Richard O. Hart
Glenn Lupton

DEC FUSE: Building a Graphical Software Development Environment from UNIX Tools

DEC FUSE is an integrated programming environment for UNIX systems. It is an evolution of the FIELD environment developed at Brown University. To take advantage of the features of workstations developed during the 1980s, these environments were designed to provide graphical user interfaces for commands commonly used by UNIX software developers. DEC FUSE uses two methods to create an environment from smaller and simpler software components. These methods are sending messages between components and layering graphical interfaces on top of UNIX commands. DEC FUSE uses these methods to create an easy-to-use, integrated environment with more features than its individual components.

The UNIX operating system originated at Bell Laboratories in 1969 and rapidly grew more popular, first within Bell Labs, then at universities and, since the early 1980s, at commercial enterprises. One reason cited for its success is that it is a good operating system for programmers.[1] The wealth of simple tools and the ability to combine them easily into new tools provides an attractive environment for software development. Projects organize their development processes around the capabilities of UNIX tools like sccs for version control and make for application building. Developers build project-specific tools using UNIX commands in shell scripts and have become proficient in the use of tools like the dbx debugger and the emacs and vi editors.[2] Developers have also become accustomed to commands for text manipulation (sed, awk), searching (grep), and comparing (diff), and the use of these in combination with other commands to do special tasks.

In the late 1980s, workstations came into common use for software development. Workstations provided additional compute power and were capable of displaying complex graphics and providing point-and-click interfaces. The UNIX tools and shell environment, designed around character-cell video terminals and hard-copy devices, did not make effective use of these workstation capabilities. Different tools and a different approach to combining them were needed to provide an effective workstation-based development environment that would take advantage of the additional compute power available to workstation users and the graphical interfaces available using the X Window System.[3]

In this paper, we define the characteristics of some integrated software development environments designed to take advantage of modern UNIX workstations. We describe the DEC FUSE product as an example of one of these environments and present two methods used to create the DEC FUSE product. With the first method, we show how tools are built as graphical user interfaces (GUIs) on top of existing UNIX commands. Then, we show how messaging enables these tools to work together. We present trade-offs and design alternatives for each method.

Integrated Software Development Environments

Integrated software development environments are collections of software programs, or tools, that are used together to accomplish one or more phases of software development. DEC FUSE and other integrated software development environments, including HP SoftBench from Hewlett-Packard and SPARCworks from Sun Microsystems, are based on a control integration model.[4–7] Control integration enables tools to make requests of other tools for information or to do required tasks.[8]

The DEC FUSE, HP SoftBench, and SPARCworks environments were strongly influenced by work done at Brown University on the FIELD programming environment by Steven P. Reiss.[8,9] DEC FUSE, in fact, continues to use some code originally written as part of FIELD. These environments share the following features with FIELD:

- Environments are collections of cooperating tools. Each tool addresses a single aspect of the software development process such as editing, searching, debugging, or building. This follows the UNIX philosophy of making tools or commands simple and focused on a single problem. As a result, they are easier to build, maintain, and use. The tools cooperate with each other by performing operations at the request of other tools. For example, the builder tool can request that the source code corresponding to an error be displayed, and the text editor will present the code.
- Tools use a selective broadcasting communications method. Tools send simple, usually textual, messages to communicate with other tools.[10] A message may be either a request for a service or a notification of the occurrence of an event. Tools register their interest in receiving particular messages. A message is then broadcast without requiring the sender to specify who will receive it. Since requests are not directed to a particular tool, a tool can be replaced with a similar tool that responds to the same messages without making changes to the sender. Because messages are broadcast, multiple tools can receive a notification and each can take appropriate action.
- Source files and annotations are viewed using a single text editor. Each tool that needs to present source text to the user does so by sending request messages that are processed by a single source text editor. The text editor displays the desired source files, and it may also place annotations next to source lines of interest. Annotations are used to link the sources with other parts of the environment. For example, the location of breakpoints is provided by the debugger, the location of build errors by the builder, and the location of strings matching a pattern by the search tool. Each of these locations is identified with an annotation symbol next to a line of source code in the editor display.
- GUIs are built on top of UNIX tools. Many of the tools in the environment are GUIs fitted to existing UNIX commands such as `make`, `grep`, and `dbx`. These interfaces provide menu and button access to these commands and their options; they also interpret the results of the commands, presenting them in formatted, interactive displays.
- Program information is presented pictorially. The graphical display capabilities of the workstation are used to pictorially present information that may be complex or extensive. For DEC FUSE, this includes a program's function call graph, the dependencies in a makefile, or the execution times of each function in a program. This issue of the *Digital Technical Journal* presents another example of displaying information pictorially with DEC FUSE in the paper "Adding a Data Visualization Tool to DEC FUSE."[11]
- Users continue to use familiar tools and methods. Because the FIELD and DEC FUSE environments are built using existing tools such as `make`, `sccs`, and `dbx`, users can continue to use tools with which they are familiar. They can also use existing makefiles and source libraries in the environment. In addition, users can make a gradual switch to an environment such as DEC FUSE. They can use DEC FUSE when it is most advantageous and continue to use older tools and methods when that is preferable.

DEC FUSE Overview

The primary goal of the DEC FUSE product was to create a commercially useful, integrated software development environment supporting a variety of programming languages, including C, C++, and Fortran. The DEC FUSE environment takes advantage of the capabilities of the UNIX workstation, while allowing software developers to preserve their investment in familiar UNIX tools. DEC FUSE designers adopted some FIELD components, which were converted to use Motif. Extensions were also made to the FIELD environment to create the DEC FUSE product. These extensions are described in the next sections. Several tools have been added to the environment through successive releases of DEC FUSE. The tools supplied with DEC FUSE version 2.1 are listed in Table 1 and are described in subsequent sections.

Selective Broadcasting Mechanism

The messaging used by DEC FUSE, called the multicast messaging system, has been extended in two ways beyond its FIELD origins. First, messages have been made more functional in nature. In the FIELD environment, messages are strings that are assembled by

Table 1
Tools Supplied with Digital's DEC FUSE Version 2.1

DEC FUSE Tool	UNIX Commands Used
Editors	emacs, vi (and a Motif-based editor)
Debugger	dbx or DECladebug (on Digital platforms)
Search	grep, fgrep, egrep
Builder	make, gnumake
Code manager	sccs, rcs
Man page browser	man
Cross-referencer Call graph browser C++ class browser	Use common data from compilers or other source scanners.
Profiler	prof, gprof, pixie
Compare	diff
Help	HyperHelp
DEC FUSE shell	sh, csh, ksh, ...

the sending tool and delivered to receiving tools. The receiving tools have registered an interest in particular messages by describing them using a pattern string. DEC FUSE uses a more functional interface that more closely resembles a remote procedure calling mechanism. Each tool defines the messages that it can send and receive as function definitions using the DEC FUSE tool integration language (TIL). Second, a set of components called the DEC FUSE EnCASE facility has been developed to support the integration of new tools and new messages into the DEC FUSE environment.[5] These components include the TIL compiler and the Message Monitor tool, described later in this paper.

Choice of Source Code Editor

Instead of having a different editor as part of each tool, the FIELD environment provided a single GUI-based editor. Because most users have strong preferences about which text editor they use, DEC FUSE extended the environment to allow each user to choose from three different editors: emacs, vi, and the DEC FUSE editor.[2] Both emacs and the DEC FUSE editor support use of annotations supplied through interactions with other tools. Users of the vi editor do not see annotations, but other tools can still position vi on source lines of interest.

DEC FUSE Tools

The tools described in this section are currently available in DEC FUSE. Figure 1 shows the DEC FUSE C++ class browser, builder, code manager, and profiler tools

- The search tool searches files for strings matching a literal string or regular expression using grep. Options available through the user interface allow for specifying whether the search should be case-sensitive, whether lines matching or not matching should be displayed, and whether the search should be limited to a single directory or an entire directory tree.
- The builder builds applications using the make or gnumake commands and existing makefiles or makefiles generated by the builder. A scrollable results window shows the output for the build operation, including diagnostic messages. The build dependencies between the files for the application that are described in the makefile are displayed graphically. The builder also distributes build actions across hosts on a local area network (LAN) and provides a user interface for specifying those hosts and for monitoring the progress of the build.
- The debugger provides a GUI to command line debuggers. This interface provides a source display with annotations for breaks, conditional breaks, and the current execution point. Debugging commands can be executed using buttons, menus, and a command line interface. Special windows provide for viewing and changing variables, breakpoints, and machine registers, and for monitoring the values of expressions.
- The compare tool displays the differences between two text files in a side-by-side display with related areas highlighted and graphically connected. The analysis of the differences is provided by the diff command.
- The code manager provides a GUI to the version management tools rcs and sccs. The code manager displays the revision history of the managed files. Details such as author, date created, and comment can be displayed for each version. In addition, the code manager uses the compare tool to display differences between revisions or revisions and files.
- The man page browser displays the reference pages for commands, system calls, subroutines, and special files. References to other manual pages in the text are hot links, and the user can click on a reference to display the other page. The man page browser can also display an index of selected reference pages. Users control the index content by specifying a keyword to match in the reference page description or a prefix to match to the reference page name. These allow users to find reference pages when they are unsure of the function or command name.
- The profiler runs an application to collect run-time statistics and displays the results at the function and line level. Statistics include the CPU time used by functions or source lines, function-call counts, line-execution counts, and function and line test coverage.
- The cross-referencer displays source locations for declarations, references, and function calls whose

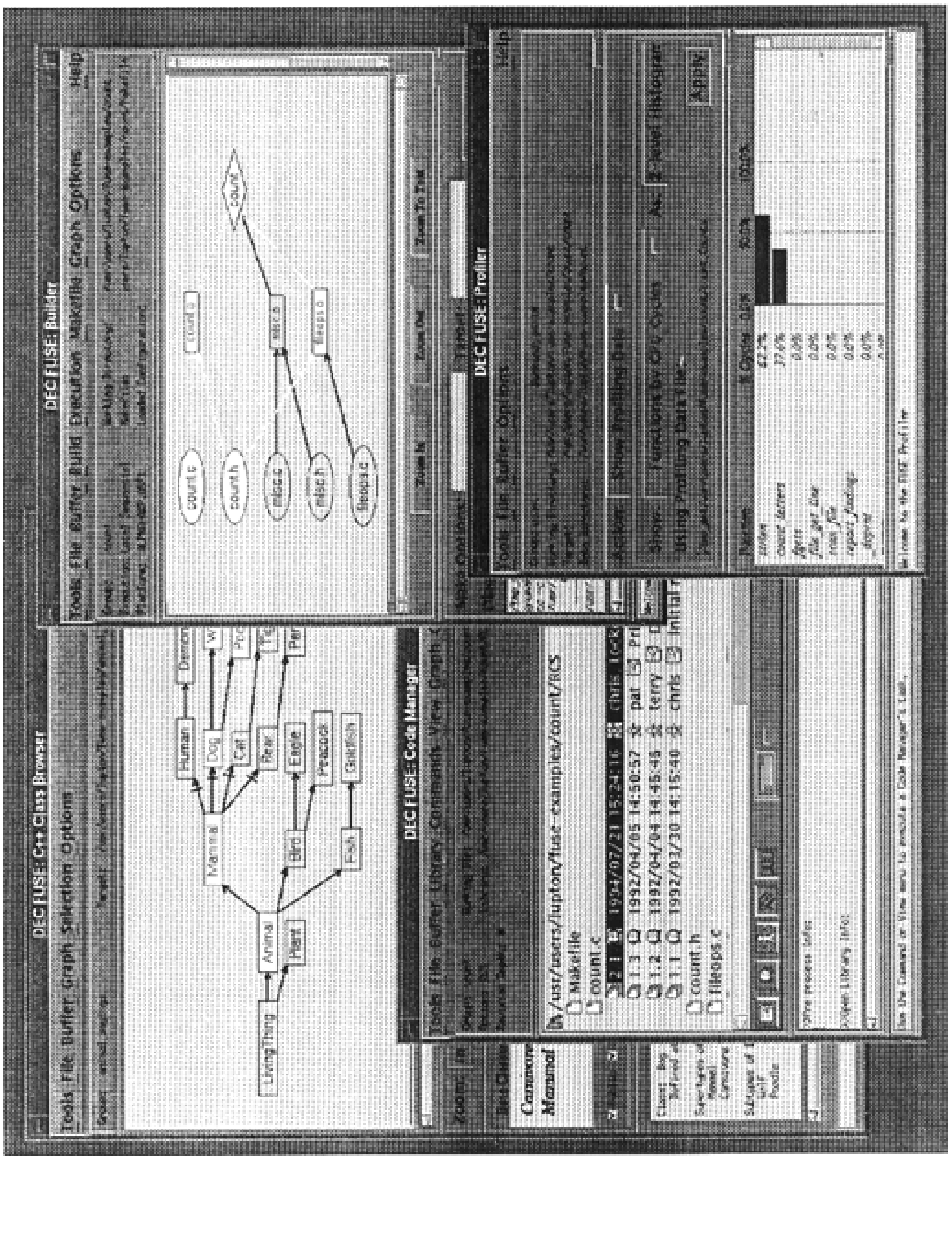


Figure 1
DEC FUSE C++ Class Browser, Builder, Code Manager, and Profiler

names match a regular expression. Queries can be constrained by declaration types and locations among other things.

- The call graph browser graphically displays the call relationships within a program. Relationships between functions, source files, and source directories can be shown. The user can constrain the display to selected parts of the program.
- The C++ class browser displays the C++-class hierarchy graphically. Inheritance paths and detailed information about each member and class can be displayed.
- Editors include the DEC FUSE text editor, emacs, and vi. The DEC FUSE and emacs editors allow other DEC FUSE tools to supply annotations on source text lines of interest. In addition, other DEC FUSE tools can be invoked from the editor, including the builder, the code manager, and the man page browser. The DEC FUSE emacs editor is a standard emacs, with additional keys defined for DEC FUSE functions.
- The help tool works with the HyperHelp tool from Bristol Technology, Inc. to display on-line help and training.
- The DEC FUSE shell supplies a terminal emulator window running a standard UNIX shell in the context of the user's DEC FUSE development environment.

In addition to the tools listed above, DEC FUSE includes a control panel tool that starts tools and manages their environment.

Using the DEC FUSE Tools Together

The messaging mechanism allows each of the tools to make selected operations available to other tools. For example, the editor makes its ability to open and display a source file and to position to a specific line available to the other DEC FUSE tools through messages. The man page browser accepts a message that causes it to display a manual page for a specified topic. The following scenario, summarized in Figure 2, shows how messaging ties together DEC FUSE tools into an integrated environment.

1. To locate places in an application that need to be changed, the developer starts the DEC FUSE search tool and looks through C source files for occurrences of a particular name. The files and lines containing a match are displayed in the search tool. By double-clicking on a line, the corresponding file is loaded into the DEC FUSE editor, and the line is displayed with an annotation that the search tool provided the location. (The search tool is used in this scenario, but the cross-referencer can also be used to do this task.)

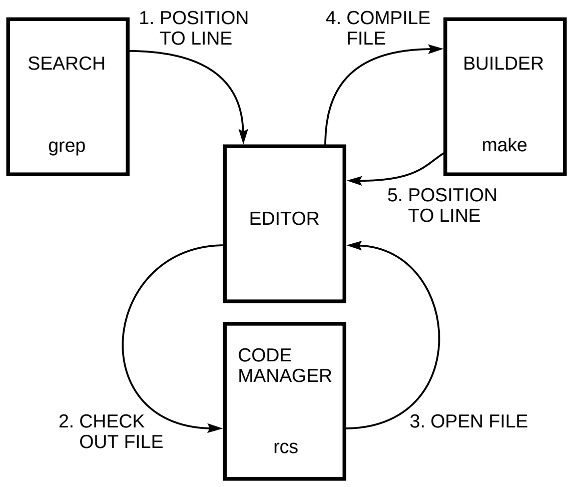


Figure 2
DEC FUSE Tool Communications

2. After inspecting the source, the user decides to modify the code, but must first check it out using rcs. By choosing the "check out" menu item in the editor, the user starts the DEC FUSE code manager, which shows the user the revision being checked out and allows the user to browse the library before confirming the check-out operation.
3. The code manager sends a message to the editor telling it to load the file to ensure that the user is editing the latest version.
4. The user edits the file and then starts a compilation using the "compile file" menu item in the editor. This starts the DEC FUSE builder, which runs make and displays compiler diagnostics.
5. By double-clicking on a diagnostic, the user gets back into the editor on the line containing the error.

The messaging mechanism allows for automated switching between the tools. Information is passed between the tools, thus eliminating retyping or cutting and pasting. Other features also contribute to the feeling of an integrated environment in DEC FUSE. These include consistent GUIs for all tools, global preference setting, saving and restoring of state information, and centralized help and training. However, it is the messaging that ties tools together, making DEC FUSE an integrated environment rather than a simple collection of tools.

We have now examined the features of integrated software development environments in general and the DEC FUSE environment as an example of these environments. In the next two sections, we examine two important aspects of the design of DEC FUSE. First, we discuss the mechanisms used to add graphical interfaces to existing UNIX commands. Then we present the design of DEC FUSE messaging.

Building Graphical Interfaces for Existing UNIX Commands

Most DEC FUSE tools consist of a graphical program that provides a point-and-click interface for invoking UNIX commands. This program interprets the results from the execution of the commands and presents these results graphically. This approach has several advantages over building a completely new tool. These are examined in this section, along with the implementation techniques used.

Rationale for Building a Graphical Interface for Existing Commands

Using an existing command to perform functions needed by a new command is a technique that is often used on UNIX systems. DEC FUSE tools use existing commands for the following reasons:

User Investment Protection Two types of investments must be made in software development environments. One investment is training: software developers have learned the concepts and capabilities of the underlying tools. Since the graphical interfaces of an integrated environment are built on tools that are familiar to users, they can be learned in considerably less time. For example, the concept of revisions, the semantics of revision numbers, and the capabilities of rcs are the same whether rcs is invoked from the command line or selected from the DEC FUSE code manager.

Second, a project may have invested in procedures and software that depend on project tools such as make and sccs. Users often use many makefiles that have been tailored to meet the needs of their project. Likewise, most projects use sccs and rcs in ways that must be supported by scripts. By building the code manager and builder on the existing rcs, sccs, and make utilities, this investment is preserved. (The DEC FUSE code manager provides mechanisms to support user-written scripts used in combination with sccs and rcs.)

Easier to Invoke Operations Although the UNIX command line environment is extremely flexible, most users find themselves frequently referring to reference pages to check command syntax and option flags. By replacing commands with menu items and buttons and by replacing flags with toggle buttons and fill-in-the-blank dialog boxes, users interact with the tools faster with less typing and less browsing through reference pages. This is especially true for novices who have not defined their own collection of aliases and scripts.

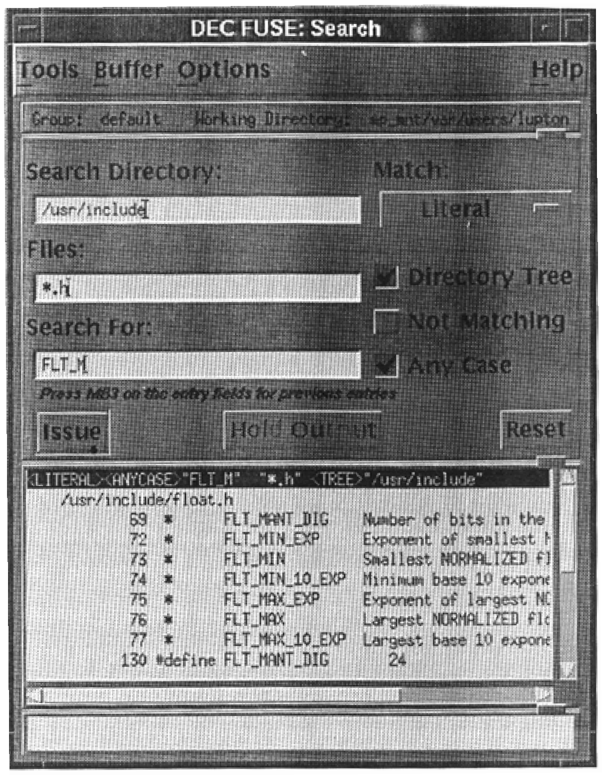
For example, searching all the header files in a directory hierarchy for the occurrence of a string requires a command like the following:

```
find /usr/include -name "*.h"
  -exec grep -i FLT_M {} /dev/null \;
```

This is a typical example of a command that a software developer might need to use from time to time. The command would be entered on one line. A first-time user, however, might not correctly input all the details of the command for the following reasons:

- The "*.h" designation includes quotation marks so that it is not immediately expanded by the shell in the user's current directory, but instead expanded by find in all the subdirectories in the /usr/include tree.
- If the search is to be case-insensitive, the -i switch must be used with the grep command.
- The grep command supplies the name of the file where the string is found only if more than one file is supplied in the grep argument list. /dev/null is added to make grep include the file names in the output.
- The find command requires that subcommands that it will execute be terminated with a semicolon. Because a semicolon is also recognized by the shell, it must be preceded with a backslash (escaped), so that find will see it.

To do the same operation from the DEC FUSE search tool, the user fills in some fields and sets a toggle (see Figure 3). This can be done easily and correctly the first time by both novice and experienced UNIX users.


DEC FUSE: Search
Tools Buffer Options Help
Group: default Working Directory: ~/usr/users/lupton
Search Directory:
Match:
Literal
Files:
Directory Tree
Not Matching
Any Case
Search For:
Press MB3 on the entry fields for previous entries
Issue
Hold Output
Reset

Search Directory: /usr/include
Files: *.h
Search For: FLT_M
Directory Tree [✓]
Not Matching [ ]
Any Case [✓]

```
<LITERAL><ANYCASE>"FLT_M"  "*.h" <TREE>"/usr/include"
   /usr/include/float.h
           69  *       FLT_MANT_DIG    Number of bits in the
           72  *       FLT_MIN_EXP     Exponent of smallest N
           73  *       FLT_MIN         Smallest NORMALIZED fl
           74  *       FLT_MIN_10_EXP  Minimum base 10 expone
           75  *       FLT_MAX_EXP     Exponent of largest NC
           76  *       FLT_MAX         Largest NORMALIZED flo
           77  *       FLT_MAX_10_EXP  Largest base 10 expone
          130 #define FLT_MANT_DIG       24
```

Figure 3
DEC FUSE Search Tool

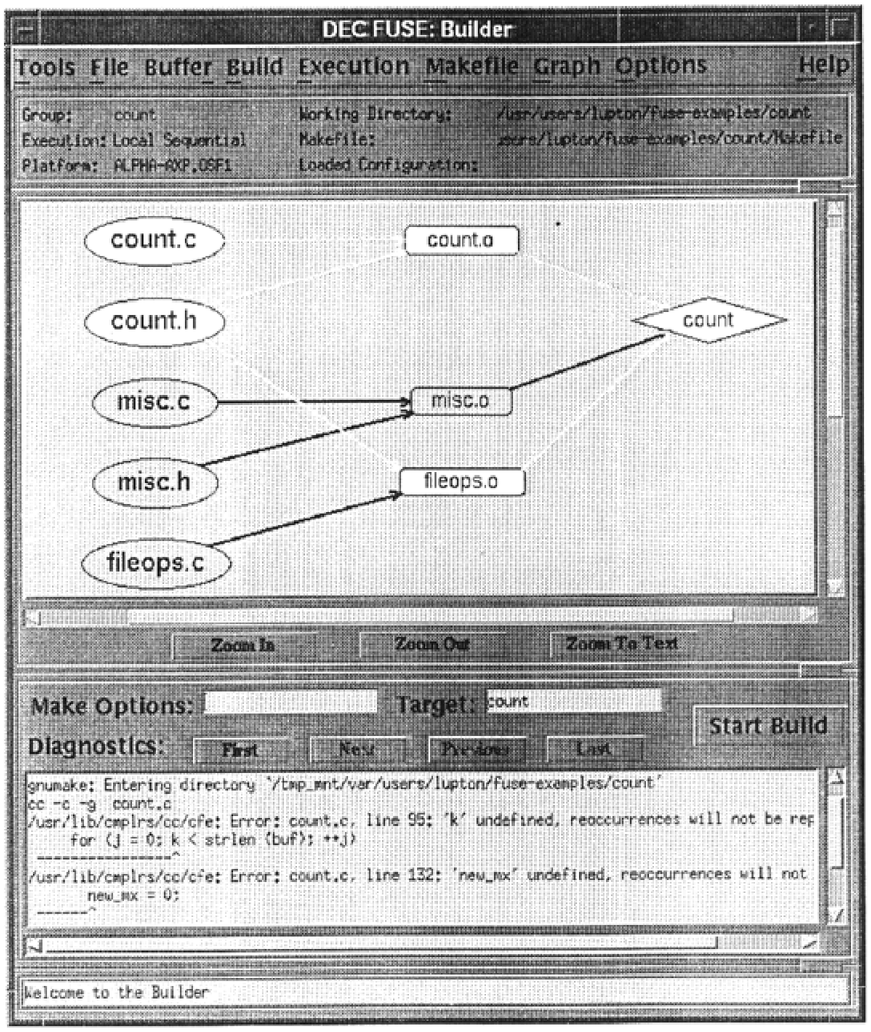
When the user spots an interesting occurrence in the output from a grep command and wants to edit the file, a command line interface requires the user to enter the command to edit the file and to type the file name and line number. Using the DEC FUSE search interface, the user double-clicks on the interesting line in the search tool and the editor automatically loads the file and sets the position to the desired line, saving typing and eliminating the possibility of errors.

Hiding Details Another advantage of graphical interfaces on underlying commands is the ability to hide details of particular commands. For example, theDEC FUSE code manager supports bothsccs and rcs with the same graphical interface. A user does not need to know the differences between rcs and sccs; by using the graphical interfaces, the user can see similar version history information from either underlying library format.

Graphical Presentation One advantage of a workstation is its ability to present information graphically. A GUI layered on a command line tool can analyze the output of the tool and present it to the user graphically, making the information in the output easier to understand.

An example of this is the dependency graph in the DEC FUSE builder, as shown in Figure 4. The graph displays the build dependencies for the user's application as specified explicitly or implicitly in the application's makefile. This display is an analysis and presentation of the output provided by make when run with options that produce debugging information about makefiles. Nodes designated orange in the graph represent the files that have changed. Nodes designated red in the graph represent the files that need to be rebuilt because of their dependency on the changed files.

Another example of using the graphical capabilities of the workstation is the DEC FUSE compare tool, which is built on the UNIX diff utility. The output of the UNIX diff utility is textual; an example is shown in Figure 5. In contrast, Figure 6 shows how the DEC



Group: count
Execution: Local Sequential
Platform: ALPHA-AXP.OSF1
Working Directory: /usr/users/lupton/fuse-examples/count
Makefile: sers/lupton/fuse-examples/count/Makefile
Loaded Configuration:

Make Options:
Target: count

Diagnostics:

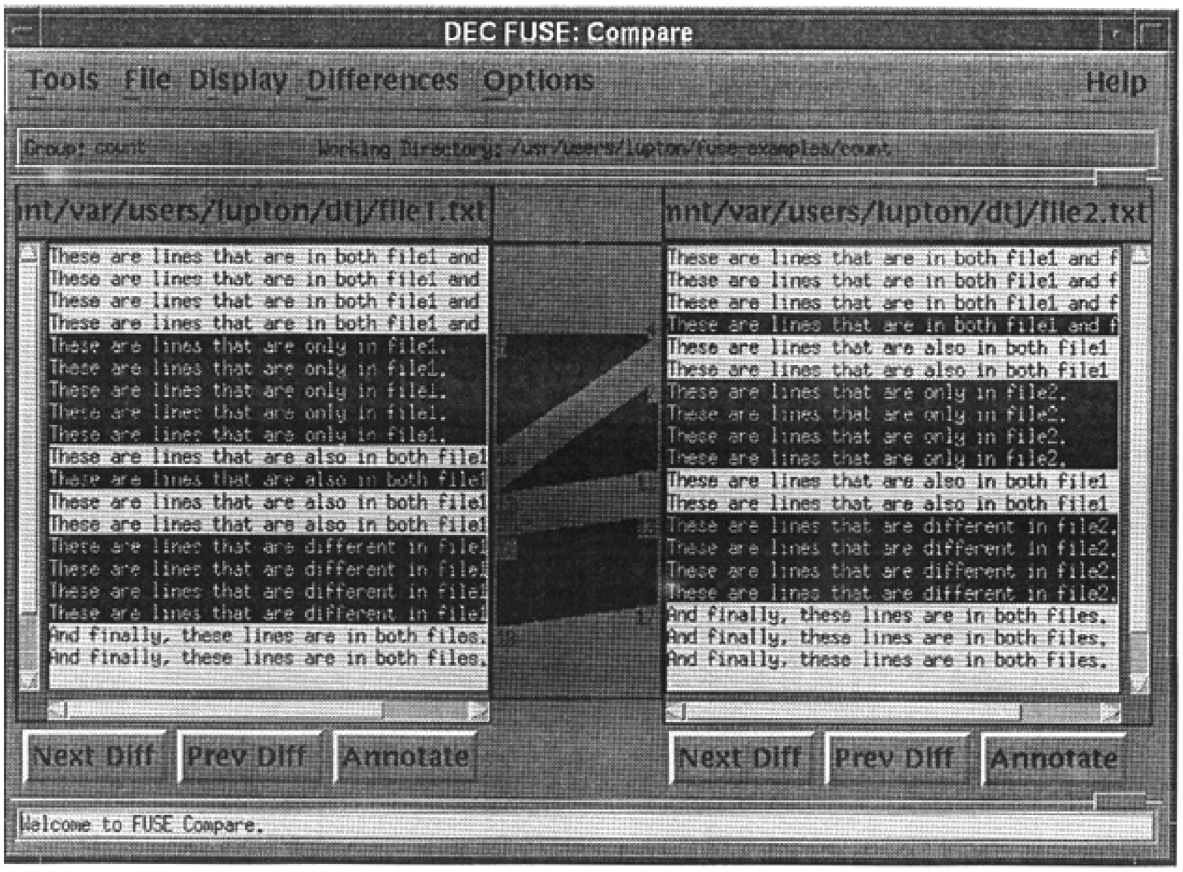
```
gnumake: Entering directory `/tmp_mnt/var/users/lupton/fuse-examples/count'
cc -c -g  count.c
/usr/lib/cmplrs/cc/cfe: Error: count.c, line 95: 'k' undefined, reoccurrences will not be rep
     for (j = 0; k < strlen (buf); ++j)
 ----------------^
/usr/lib/cmplrs/cc/cfe: Error: count.c, line 132: 'new_mx' undefined, reoccurrences will not
       new_mx = 0;
 ------^
```

```
Welcome to the Builder
```

Figure 4
DEC FUSE Builder Tool with Dependency Graph

```
csh# diff file1.txt file2.txt
5,9d4
< These are lines that are only in file1.
< These are lines that are only in file1.
< These are lines that are only in file1.
< These are lines that are only in file1.
< These are lines that are only in file1.
11a7,10
> These are lines that are only in file2.
> These are lines that are only in file2.
> These are lines that are only in file2.
> These are lines that are only in file2.
14,17c13,16
< These are lines that are different in file1.
< These are lines that are different in file1.
< These are lines that are different in file1.
< These are lines that are different in file1.
---
> These are lines that are different in file2.
> These are lines that are different in file2.
> These are lines that are different in file2.
> These are lines that are different in file2.
```

Figure 5
Sample diff Output



```
Group: count          Working Directory: /usr/users/lupton/fuse-examples/count
```

```
nnt/var/users/lupton/dtj/file1.txt
These are lines that are in both file1 and
These are lines that are in both file1 and
These are lines that are in both file1 and
These are lines that are in both file1 and
These are lines that are only in file1.
These are lines that are only in file1.
These are lines that are only in file1.
These are lines that are only in file1.
These are lines that are only in file1.
These are lines that are also in both file1
These are lines that are also in both file1
These are lines that are also in both file1
These are lines that are also in both file1
These are lines that are different in file1
These are lines that are different in file1
These are lines that are different in file1
These are lines that are different in file1
And finally, these lines are in both files.
And finally, these lines are in both files.
```

```
nnt/var/users/lupton/dtj/file2.txt
These are lines that are in both file1 and f
These are lines that are in both file1 and f
These are lines that are in both file1 and f
These are lines that are in both file1 and f
These are lines that are also in both file1
These are lines that are also in both file1
These are lines that are only in file2.
These are lines that are only in file2.
These are lines that are only in file2.
These are lines that are only in file2.
These are lines that are also in both file1
These are lines that are also in both file1
These are lines that are different in file2.
These are lines that are different in file2.
These are lines that are different in file2.
These are lines that are different in file2.
And finally, these lines are in both files.
And finally, these lines are in both files.
And finally, these lines are in both files.
```

```
Welcome to FUSE Compare.
```

Figure 6
DEC FUSE Compare Tool

FUSE compare utility displays these differences graphically, using highlighting to indicate the differences and shapes to connect regions in the two files that relate. The display allows differences to be viewed in the context of the lines before and after them and the lines that correspond to them in the other file.

Reduced Tool Development Work An obvious advantage for the developers of the interface is that building on a command line tool may involve considerably less work than designing and implementing a new tool that includes all the capabilities of the command line tool. Furthermore, not every capability needs to be provided through the user interface of the tool, because users have access to less-used capabilities through the command line. For example, the seldom-used administrative features of sccs and rcs can be omitted from the user interface. Thus, with a minimum amount of effort, it is possible to provide a convenient interface to the most important underlying capabilities.

Managing Command Interfaces

It is common on UNIX systems to use the output of one tool as input to another. In the case of DEC FUSE, the output of command-line tools is being used as input to DEC FUSE tools. The DEC FUSE tools construct commands and pass them to a separate process for execution. The results of these commands are then interpreted by the DEC FUSE tools so that desired information can be presented to the user. The methods used to issue commands and to analyze their results vary from one DEC FUSE tool to another.

One method used by DEC FUSE tools is to directly issue commands using the popen library function, which both starts execution of the command and creates a pipe to the process running the command. This is done by tools like the man page browser and search. Output from the man or grep commands that they issue is parsed by the DEC FUSE tool, often using a simple mechanism such as the standard C library function fscanf, which applies a format string to a line to parse it. Some tools also make use of lex with or without yacc to aid in parsing the output of the commands.[12,13]

Other tools use PMAT (pattern matching) routines for examining command output for desired patterns. The PMAT functions were developed by Steven Reiss as part of the FIELD environment. They are used in FIELD both for managing messaging as well as for interpreting the output of UNIX commands. For DEC FUSE interfaces to UNIX commands, the patterns used by the PMAT routines are organized in tables. Portions of two of these tables are shown in Figure 7. These examples are for the output of gnumake and a make program supplied with Digital UNIX.[14] For this analysis, there are two significant parts of each pattern table entry: a text pattern that may be found in the command output, and the name of a routine to be called if the associated pattern is found. For example, when the error message “Failed to remake target file ‘%1s’” is recognized, the function named make_giving_up is called with arguments that match specifications in the pattern string.

Additional values from the table (omitted in the figure) are also passed as arguments to the routine. The string ‘%1s’ in the pattern is similar to the conversion specifications used by scanf. It represents a field in the output that will be passed to the recognition routine when a pattern is recognized. Some of the field specification characters used are given in Table 2. The number preceding most field specification characters tells the pattern match what position this field should hold in the argument list passed to the recognition routine. When there is no number with a field specification character, that field is not passed to the recognition routine.

Choosing the Appropriate Command Interface Method

The DEC FUSE product was designed to be portable across several hardware platforms and many operating system versions. DEC FUSE was developed on the ULTRIX system and has been ported to SunOS, AIX, HP-UX, and Digital UNIX operating systems. It was released to customers on all these platforms, except AIX. Since portability across platforms and versions is a goal, interfaces for different command implementations and versions need to be considered. The choice of interface method is made based on the complexity of the interface (the number of commands and expected responses), the number of different interfaces needed because of system differences, and the rate at which the interfaces are evolving.

Most common UNIX commands, such as grep, man, and diff, have regular output that seldom changes. The versions of these commands on the desired platforms and operating systems have few differences, so it is not difficult to write portable code that can issue these commands and interpret the output using the lex, yacc, or the scanf functions.

In cases in which the output is less regular and varies across commands and platforms, the PMAT facilities are more appropriate. This includes the DEC FUSE builder, which must support several different make programs on the supported platforms. The PMAT facilities allow for interpreting a large number of different format lines and for selecting tables of patterns appropriate to the underlying command. This makes it easier for the builder to accommodate a variety of make programs and interpret both output from make and output from compilers.

```
/****** Pattern table for gnu make ******/
static MAKE_PAT gnu_pattern_table[] = {
   {"Reading makefiles...",        gnuscan_makefile, ...},
   {"Considering target file'%1s'",  gnuscan_consider, ...},
   {"Found an implicit rule for'%1s'", gnuscan_flags,    ...},
   {"Updating goal targets....",  gnuscan_makefile, ...},
   {"File'%1s' was considered already",  gnuscan_done,     ...},
   {"Must remake target '%1s'",        gnuscan_flags,    ...},
   {"Failed to remake target file'%1s'", make_giving_up,   ...},
   {"No need to remake target '%1s'",     gnuscan_flags,    ...},
   {"# Files",                   gnuscan_files,    ...},
   {"# Not a target:",             gnuscan_notarget, ...},
   {"#  commands to execute",          gnuscan_setrules, ...},
   {"#  Phony target",              gnuscan_defflags, ...},
   {"#  Precious file",            gnuscan_defflags, ...},
   {"# VPATH Search Paths",             gnuscan_files,    ...},
   {"# gnumake: Entering directory'%1s'", gnuscan_proj,     ...},
   {"# gnumake: Leaving directory'%1s'",  gnuscan_proj,     ...},
   {"%1s: %2r",                   gnuscan_def,      ...},
   {"%1s:",                     gnuscan_def,      ...},
   ....
};


/****** Pattern table for dec make ******/
static MAKE_PAT dec_pattern_table[] = {
   {"doname(%1s,%2d)",               decscan_consider, ...},
   {"setvar: @ = %1s noreset",       decscan_flags,    ...},
   {"setvar: ? = %1r",            decscan_flags,    ...},
   {"! = %1r",                   decscan_adjust,   ...},
   {"look for explicit deps. %1d",   decscan_flags,    ...},
   {"look for implicit rules. %1d",  decscan_flags,    ...},
   {"Current working directory for make is %1s",
                                 decscan_proj,     ...},
   {"%1s: %2r",                     makescan_def,     ...},
   {"%1s:",                       makescan_def,     ...},
   {"Reading %1s",                  decscan_makefile, ...},
   .....
};
```

Figure 7
make PMAT Patterns

Table 2
Some PMAT Field Specification Characters

Field Character	Data Type
d	Decimal number
x	Hexadecimal number
c	A single character
s	A string, delimited by white space
q	A string, delimited by quotation marks
r	A string, from the current location to the end of the line
e,f,g	Floating-point numbers

The tool with the most complex command interface is the debugger. The debugger shares the following issues with other tools, but demonstrates them most forcefully:

1. Debuggers are big and complex. Debuggers are more complex than the commands used in other DEC FUSE tools. Each debugger engine accepts many commands, all of which have their own output that must be parsed. The debugger engine also continues to run while the user works. Unlike most other tools, the debugger engine is not restarted every time the user wants more information, so the debugger process must be managed over a long period of time.
2. Debuggers are evolving more quickly. Debuggers frequently change to support new needs (for example, new languages like C++, threads, or hardware architectures), so new debugger commands or new output from old commands can be expected often.
3. Synchronizing the front end and the debugger engine is a complex task. The graphical front end

must remain synchronized with the debugger engine it is running. Preserving this synchronization is made more difficult for three reasons. First, users can enter debugger commands directly as text, making it difficult for the front end to determine their effect. These commands may require updates to the graphical displays or the internal state information used by the front end. Second, the debugger may not be in a state where it can accept commands (when the user program is running for example), so the front end cannot update displays. Third, spontaneous and unexpected debugger engine output may occur as the result of traces or certain breakpoints.

4. Different debuggers use different commands. Commands on different debuggers can be different in both name and design. For example, with the `dbx` debugger available on SunOS, AIX, and Digital UNIX, the commands `func` and `file` can be used to find the currently active function and the name of the source file where that function is defined. The `xdb` debugger used on HP-UX, however, uses the `L` command to present both the current function and the name of the file where it is defined, as well as to display the current source code line.
5. The same debugger commands have different output. Other commands, although similar in name and design, can produce output that is different enough to cause problems. One example is the `where` command used in `dbx` on both Digital UNIX and SunOS platforms. This command returns the current stack information. The Digital version includes a pointer character (>) to show which stack entry is the current scope; however, the SunOS version does not supply this scope information. Therefore, a debugger GUI program must be carefully designed to get needed scope information if it must support both debugger engines.
6. The output of some debugger commands is complex, and the results of some debugger commands are difficult to parse. For example, in the display of the content of a data structure, the format of the output will vary depending on the source language used in the application.

Experiences with DEC FUSE suggest that there is no easy solution. Addressing these issues results in many specialized routines in the DEC FUSE debugger tool to both construct debugger commands and interpret the results. Techniques that help to make the problems more manageable include the following:

- Cleanly separate generic-GUI and command-specific code. The design of the debugger GUI identifies the operations that it requires of the debugger engine and the data that it must get from the engine. These are provided by a set of functions whose implementation will vary from one engine to another. These functions will be modified over time to accommodate the evolution of the engines. Another method being designed now is to use C++ classes to encapsulate code for each supported debugger engine.
- Limit the details that the GUI depends on. One way to limit the dependency of the GUI on the details of the engine is to provide GUI support for only the most frequently used debugger operations, while providing a command interface for the remaining operations. Another technique is to avoid interpreting the output of the engine when possible and simply display the output of the command in a text window.
- Implement special interface commands in the engine. When it is possible to change the underlying debugger, special commands and output can be implemented by the debugger designed exclusively for use by the GUI front end. For example, the DECladebug debugger engine has been modified with the introduction of two new commands for use by the graphical interface that simplify the task of displaying data structures in the GUI. Although other commands display data structures for the user, the format of the output of these commands is designed to be easily interpreted by the GUI. These commands are designed for the exclusive use of the GUI. They need not be changed for the user, for example, to improve readability; thus the evolution is controlled.

Fortunately, most UNIX tools are not as complex as the debugger. In fact, building a GUI for commands with output that seldom changes and is consistent across implementations is a straightforward task.

Using Messaging to Make Independent Tools Work Together

As described earlier, each DEC FUSE tool focuses on a single, separate software development task. This design philosophy, sometimes called "divide and conquer," combined with the DEC FUSE multicast messaging system (MCMS) makes it easier to maintain or replace tools. DEC FUSE tools can therefore be easily replaced with alternative tools that provide the same function.

MCMS is the key to making independent tools work together. Any message sent by a tool is delivered to all tools that express an interest in receiving the message. Some messages, called notifications, are defined to have no response. Other messages, called requests, have responses for which the sending tool usually waits. A tool can also eavesdrop on requests that will be handled by other tools. A DEC FUSE component called the DEC FUSE message server keeps track of the active tools and which messages each can send and receive.

Messaging with MCMS

Messages used by tools are easily defined in aTIL file, written in the DEC FUSE tool integration language. An example is the manager.til file used by the DEC FUSE code manager. Part of manager.til is shown in Figure 8. Each TIL file can define one or more tool classes. Each class definition describes how a single DEC FUSE tool will be integrated with the rest of DEC FUSE. A class definition contains three parts:

1. Attributes: This is a collection of tool attributes such as the string to be used in theDEC FUSE tools menu and the command to invoke the tool.
2. Messages: This section lists definitions for all messages sent and received by the tool, including their arguments and return values. Messages that have return values defined are called requests, and the returned value is expected by both the message switch and the tool that sent the request. Messages with no return value (the type is void) are called notifications. The keyword trigger is used if the message should automatically start the tool.
3. States: This section describes when each message may be used during the execution of the tool. This section defines one or more states in which the tool

```
class MANAGER = {
      Attributes {
            label    = "Code Manager";
            accel    = "Meta+M";
            path     = "$(FUSE_SH_BIN)/manager";
            .... };

      Messages {
      /* messages accepted by the FUSE code manager */
      char *ToolReconfigure(char *working_directory,
            char *target_directory, char *target, char *other);

      trigger char *CheckIn (char *libraryname, char *filename,
            char *revision, char *comment, int  keepfile,
            int  filemode);
      ....

      /* messages sent by the FUSE code manager */
      void   CheckInNotification ( int   instance_id,
            char *libraryname, char *workdir, char *filename,
            char *revision, int status);
      ....  };


      States {
            start {
                  receives {
                  ToolReconfigure,
                  ....  };
                  sends {
                  ....  };
            };
            running {
                  receives {
                  ToolReconfigure,
                  CheckIn,
                  CheckOut,
                  .... };
                  sends {
                  ToolReconfigure,
                  CheckInNotification,
                  .... };
};    };    };
```

Figure 8
DEC FUSE Tool Integration Language File

may exist. Tools can change their state, and within each state only the listed messages may be used. Most DEC FUSE tools need only two states: an initialization or start state used during tool start-up and a running state. Other states may be needed by some tools. For example, the builder uses a building state to advise the message server that a build is in progress and that some requests (like another build request) are not allowed.

A TIL compiler translates the TIL files of DEC FUSE tools into the data files needed to run DEC FUSE. Figure 9 summarizes how the files generated by the TIL compiler for a DEC FUSE tool (named fuse_tool) fit into the architecture ofDEC FUSE.

The TIL compiler combines information from the fuse_tool TIL file with TIL files for tools already installed on a system. The TIL compiler generates three files:

1. fuseschema.msl – This file tells the message server which tools wish to receive which messages.
2. tools.rc – This file tells the control panel how to start each tool. Tools may be started in response to a trigger message or manually from the Tools menu found in each DEC FUSE tool.
3. FUSE_fuse_tool.c – This file contains functions for each of the messages that the tool wishes to send. This file is compiled and linked with fuse_tool along with libfuse.a. Messages are sent by simply calling these functions. This file also contains an initialization function in which callback functions for messages that the tool receives are registered.

The use of the TIL compiler in DEC FUSE provides a mechanism similar to a remote procedure call facility. This allows tools to send a message using a single function call. This contrasts with the messaging mechanisms used in the HP SoftBench and Sun SPARCworks products, which require a number of calls to the messaging application programming interface (API) to allocate, assemble, send, and free a message. These mechanisms also require tools to assemble and register patterns corresponding to the messages that they want to receive, a function handled by the initialization function in the C source file generated by the TIL compiler.

To simplify the task of integrating tools, DEC FUSE also supplies a DEC FUSE message monitor. This tool monitors and debugs messages sent by tools and provides a mechanism for integrating shell scripts as tools that can send and receive messages.

Simplified Tool Replacement

MCMS does not require the user to specify the tool that does the work. When a tool sends a message using MCMS, it does not specify what tool should service the message. This allows for replacement of the tool that services the messages with an equivalent tool, without making any change to the sender. This mechanism is used in DEC FUSE to allow users to select which of three editors they want to use and whether they want to use a GUI debugger based on `dbx` or DECladebug.

This mechanism also facilitates upgrading the DEC FUSE environment. Recently, the Motif help widget in DEC FUSE was replaced with the HyperHelp tool. The replacement was facilitated by continuing to use the existing messages. This isolated all changes to the DEC FUSE help tool. The help tool continues to receive messages of the form

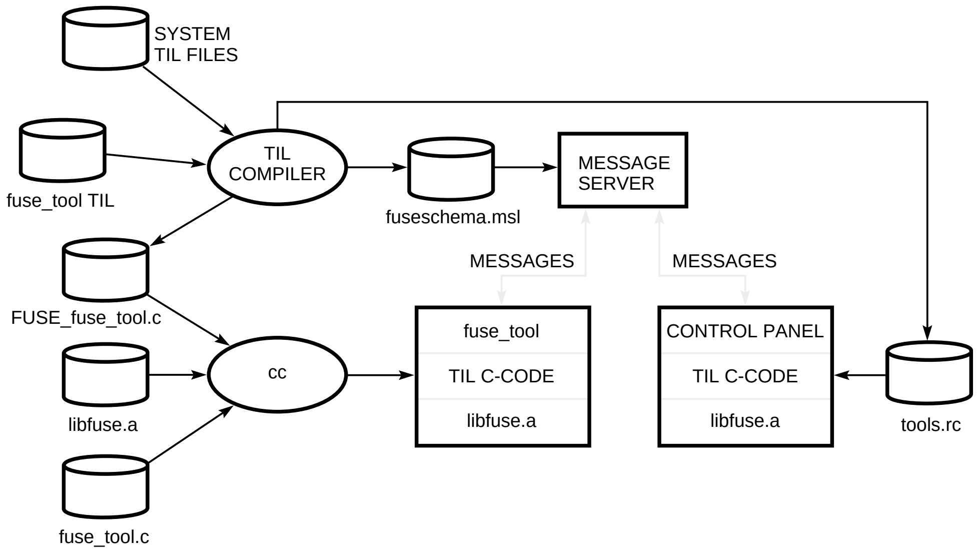


Figure 9
Use of TIL-generated Files in the DEC FUSE Architecture

```
trigger void HelpShowTopic(char *product,
                           char *mode,
                           char *topic);
```

In the previous version, the message argument, topic, was a string that identified what kind of help was desired. The new help tool uses numbers instead of names to identify help topics. Consequently, a simple mechanism was designed to translate the strings received in the HelpShowTopic messages to the desired HyperHelp topic number.

Conclusion

DEC FUSE provides an integrated programming environment for UNIX software development that takes advantage of the graphical capabilities of workstations. Two key techniques are used to implement DEC FUSE:

- The layering of GUIs on existing UNIX command line tools
- A multicast messaging mechanism that permits tools to interoperate without limiting the environment to specific tools

The GUIs provide point-and-click interfaces for invoking operations and specifying options and use pictures and diagrams in addition to text to display information. At the same time, the use of traditional UNIX commands to perform programming tasks preserves the user's investments in those underlying tools.

The GUIs interpret the output of UNIX commands and present the information in pictorial and interactive displays. A variety of techniques can be used to process the output of a command line tool, depending on the complexity of the tool output. Simple text-processing techniques are usually adequate for interpreting the output of command line tools. When the underlying tool output is syntactically complex or evolving, or when considerable state information is frequently needed from the underlying tool, it becomes difficult to apply these techniques. Under these conditions, designs that avoid the processing of human readable output are preferred.

The use of messaging is consistent with the UNIX philosophy of creating simple tools and letting the user combine them in any way that might be useful. The messaging mechanism ties the individual tools together into an integrated environment by allowing tools to invoke operations in other tools on the user's behalf. This eliminates steps for the user, and it also eliminates the potential for errors. Because the tools are still autonomous and interface solely by means of the messaging, equivalent tools that accept the same messages can be substituted, allowing for user and project preferences.

Acknowledgments

We would like to thank the many past and present members of the DEC FUSE team who contributed to the design and implementation of the DEC FUSE product. We also want to acknowledge the work on FIELD done at Brown University by Steven P. Reiss and his students that laid the groundwork for DEC FUSE and other software development environments.

References

1. B. Kernighan and R. Pike, *The UNIX Programming Environment* (Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1984).
2. R. Stallman, *GNU Emacs Manual* (Cambridge, Mass.: Free Software Foundation, 1988).
3. R. Scheifler and J. Gettys, "The X Window System," *ACM Transactions on Graphics,* vol. 5, no. 2 (April 1986).
4. *DEC FUSE Handbook* (Maynard, Mass.: Digital Equipment Corporation, Order No. AA-Q8ZMA-TE, 1994).
5. *DEC FUSE EnCASE Manual* (Maynard, Mass.: Digital Equipment Corporation, Order No. AA-Q8ZPA-TE, 1994).
6. M. Cagan, "The HP SoftBench Environment: An Architecture for a New Generation of Software Tools," *Hewlett-Packard Journal* (June 1990): 36–47.
7. *Common Desktop Environment: Getting Started Using ToolTalk Messaging* (Mountain View, Calif.: Sun Microsystems, Inc., 1994).
8. S. Reiss, *The Field Programming Environment: A Friendly Integrated Environment for Learning and Development* (Boston: Kluwer Academic Publishers, 1995).
9. S. Reiss, "Interacting with the FIELD Environment," *Software—Practise and Experience,* vol. 20 (June 1990): 89–115.
10. S. Reiss, "Connecting Tools Using Message Passing in the Field Environment," *IEEE Software,* July 1990: 57–66.
11. D. Zaremba, "Adding a Data Visualization Tool to DEC FUSE," *Digital Technical Journal,* vol. 7, no. 2 (1995, this issue): 20–33.
12. M. Lesk and E. Schmidt, "Lex—A Lexical Analyzer Generator," *Computer Science Technical Report No. 39* (Murray Hill, N.J.: Bell Laboratories, 1975).
13. S. C. Johnson, "Yacc: Yet Another Compiler-Compiler" (Murray Hill, N.J.: Bell Laboratories).
14. R. Stallman and R. McGrath, *GNU Make—A Program for Directing Recompilation* (Cambridge, Mass.: Free Software Foundation, 1993).

Biographies

Richard O. Hart
Rich Hart joined Digital in 1980 and is currently a member of the FUSE Group. Prior to his work on the DEC FUSE programming environment, Rich was a member of UEG (ULTRIX Engineering Group) and led the first version of the Palladium distributed printing project at MIT's Project Athena. As one of Digital's representatives to the X/Open, POSIX, and ANSI standards groups, Rich has contributed to the development of software standards for transaction processing, printing, and CASE environments. He earned a Ph.D. from the University of Connecticut and is a member of ACM and IEEE.

Glenn Lupton
Glenn Lupton is a consulting software engineer and has been with Digital for 20 years. During this time, he has worked primarily on programming environments and tools, including Bliss compilers and DECset. For the last two years, he has been the technical director of the DEC FUSE project with responsibility for the overall technical content of DEC FUSE. Glenn received B.S.E.E. and M.E.E.E. degrees from Rensselaer Polytechnic Institute.