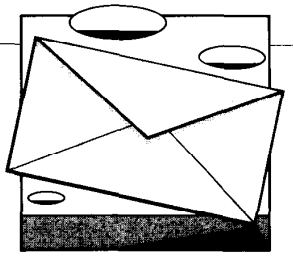

Connecting Tools Using Message Passing in the Field Environment

Steven P. Reiss, *Brown University*

Field connects tools with selective broadcasting, which follows the Unix philosophy of letting independent tools cooperate through simple conventions. Field demonstrates that this simple approach is feasible and desirable.

Workstations that offer a host of graphical capabilities, powerful processors, and mouse-based input to support integrated environments have been available for five years. However, programmers still write code in classical languages like C and Pascal with collections of unintegrated tools.

There are two exceptions to this rule. The first is the class of special-purpose, controlled environments for languages like Lisp and Smalltalk. Here, where the whole system is based on a single, semi-interpretive language, many graphical environments exist. Most of them, however, are specific to their underlying language and do not apply to programming in more common languages.

The other exception is the blossoming class of simple but powerful PC-based programming environments for traditional languages. Vendors have understood the potential for mouse input and graphical output as programmer aids. Integrated environments like Lightspeed Pascal are a powerful framework for teaching programming and for writing and debugging simple Pascal programs. PC-based environments, however, are highly integrated and suited for small programs only.

I undertook the Field project to show that you can implement on workstations the highly integrated, interactive environments like those on PCs and that you can use them for classical-language and large-scale programming. I also wanted to show how to use a workstation's more advanced capabilities to attain a more productive and powerful environment by providing functionality not found on PCs or in standard software-engineering environments.

Field was designed both to accomplish these ends and to provide a production environment for research and instructional programming at Brown University. I designed Field to be simple and inexpensive so it would be operational as soon as possible and to ensure that it

0740-7459/90/0700/0057/$01.00 © 1990 IEEE

Related research

Integrated programming environments have been widely touted as a way to increase productivity. Also, because they offer a controlled and understandable environment, they are ideal for instruction. In the past 20 years, a wide range of integrated environments, using many ways to integrate their tools, have been developed.

File level. Most of today's environments are integrated at the file level. Unix is an example of such an environment. In Unix, programmers edit source files. They use the SCCS or RCS tools to check files in and out and to manage file versions. They run the make program that takes a file describing how the system is to be built and runs the appropriate compilers and loaders on the source files to produce object files and the resulting systems. The debugger runs on the binary file and accesses the source files as appropriate.

These Unix tools are integrated only in that they operate on a common set of files. But each tool has a different interface and obeys different conventions. Programmers are free to choose the set of tools they want to use and ignore the others. None of the tools takes advantage of the others or interacts except through their eventual outputs in the file system.

Others have tried to extend Unix's limited integration. Sun's DBX tool provides a Make command to run the make program and an Edit command to run an editor on the appropriate source file. The Gnu Emacs editor can parse compiler output and go to lines containing errors. Others have extended the default make rules to do many SCCS functions automatically.

However, all these extensions are ad hoc, simple extensions to one tool. They do not provide a common interface nor a general, extensible integration mechanism.

Single system. More integration can be found in single-language environments that combine all the tools in a single system. The tools generally share a common interface and operate on a single program representation. Such environments exist for languages like Lisp and Smalltalk and on small-scale PC systems.

These systems can be highly integrated. For example, Pecan[1] lets multiple editors access the source code and updates them all as the source code changes. It uses the source-code views to animate program execution and provides an incremental compiler that runs as the editor detects a source-code change. In turn, the compiler updates a set of semantic views such as the symbol table, detects errors, and highlights them in the source-code views. You can set breakpoints in any of the source-code views.

However, these systems tend to be complex. They also can't take advantage of existing tools. Languages like Lisp and Smalltalk do support an extensible environment and give you direct access to the execution framework. This lets you add tools incrementally, so the system gains functionality and complexity over time. But to integrate a new tool typically means integrating it separately with each of the other tools, because there is no guarantee that the interfaces or conventions are similar.

Systems that incorporate traditional languages like C or Pascal are difficult to extend, especially given their size and complexity. Most of the environments developed for these languages use an interpreter to achieve a high degree of integration between an executing program and the rest of the system, a strategy that limits these systems to handling small programs.

Program database. Another way to integrate tools is to combine them into a program database. In this approach, a single database stores all relevant information about a system. These systems achieve integration by having the tools share common data structures that represent different aspects of the program and its execution.

A program database also lets independent tools have controlled access to a specific set of common data structures. In effect, the shared data structures of the single system are placed under the control of a separate database system that provides consistency and integrity between processes, and the tools access these structures through this new system.

There are two ways to implement an environment that uses a program database. The first is to have all the tools use the database directly. In this approach, the tools are designed with the database in mind and use representations that are either stored in the database or can easily be derived from the database.

This approach, which underlies the Ada Programming Support Environment, is efficient and consistent. The APSE environments store an attributed, abstract syntax representation in a common database. The compiler, debugger, loader, and other tools all access the program as Diana trees by going through the common database system. The principal disadvantage of this approach is that you must rewrite any existing tools so they can use the database. A secondary disadvantage is that you must determine the database representation before you implement the tools, so it can be a problem to add tools you did not anticipate.

The second way to implement a program database is to treat it as a software backplane. Here, the tools can use whatever representation is most appropriate — existing tools use their current representations and new tools are written to use the most efficient representation for the application. The database then stores a single, extensible representation of the data. This representation is mapped by the database system back and forth from the form needed for a particular application when it is run.

This approach lets you use existing tools and makes it easier to incorporate new tools. However, the mappings from the database representation to the application representation can be complex and indirect.

Program databases in general have disadvantages. You need an additional system to maintain the database, which adds complexity. Database systems are usually large and complex — and a program database that handles multiple clients and maintains consistent program information is no exception. Finally, this integration strategy requires you to understand well the program representation before you write most of the tools, which can make it difficult to add tools that do not fit well with your original definition.

Message facility. Field provides a third alternative, a loosely coupled message facility to integrate tools. This approach has several advantages, primarily simplicity, ease of reuse of existing tools, and ease of extensibility. This, combined with the annotation editor to provide consistent access to the source throughout the environment, leads to a powerful programming environment that can effectively use existing and future tools. While this approach cannot provide the complete integration of the other approaches, my experience is that the level of integration is high enough for almost all applications and that complete integration is not necessary.

Reference

1. S.P. Reiss, "Pecan: Program-Development Systems That Support Multiple Views," *IEEE Trans. Software Eng.*, March 1985, pp. 276-284.

could be maintained and that it would work in an educational environment. I used existing tools wherever possible, including standard Unix tools, workstation software previously developed at Brown, and other available software. I also wanted to use Field as a testbed environment in which new tools such as program-animation systems can be incorporated easily.

Integration framework. Field achieves all these goals by providing a consistent graphical front end and a simple integration framework that lets existing and new Unix tools cooperate. The front end, based on a tool set called the Brown Workstation Environment,[1] includes several input interfaces that incorporate static, pull-down and pop-up menus, dialogue boxes, and scroll bars; a powerful, extensible editor; a geometry package; drawing packages, including one for the automatic layout and display of structured diagrams; an integrated help facility; and an application window manager.

Field's principal contribution is its integration framework, which lets me tie many tools together with minimum effort. The framework combines a communications mechanism that I call *selective broadcasting*, an annotation editor that provides consistent access to the source code in multiple contexts, and a set of specialized interactive analysis tools.

In selective broadcasting, all tools talk to a central message server. Each tool registers a set of message patterns with the server. Tools communicate by sending messages to the server and receiving those messages that match their registered patterns. This approach is easy to implement and extend. It has several advantages over the more traditional integration mechanisms that involve program databases or the development of a single massive system. Field demonstrates that my simpler approach is both feasible and desirable.

Field overview

Field serves three purposes: It is the principal programming environment for teaching undergraduates, it is a research programming environment, and it is a testbed for developing new tools. The first purpose requires that the environment be easy to use, the second that it handle moderate-sized (100,000-line) systems spanning multiple files in multiple directories, the third that it be flexible and easily extensible.

The current set of Field tools includes:

- An annotation editor, a full-functioned, mouse-oriented, extensible editor for C and Pascal. The editor is augmented with an annotation window that lets you associate annotations with each line of code. You can create, remove, and query annotations through the editor. Field uses these annotations to relate the code to the rest of the environment. Field lets you have multiple annotation editors active at one time, so you can view and annotate several files simultaneously.
- A cross-referencer, which collects a relational database about a system. This database is generated the first time that cross-referencing is done on a system and incrementally thereafter, rescanning only those source files that have changed. You can specify a system as a set of source files, a binary file, or a directory hierarchy. A relational-calculus query language provides access to the database. Current relations include references (name, file, line, and assignment), declarations (name, scope, file, type, class, and line), calls (from, call, file, and line), functions (name, file, line, scope, argct, and args), files (name), and scopes (class, start_line, end_line, and file).
- A cross-reference interface, a menu-oriented interface that lets you make most simple database queries by filling in a dialogue box. This interface lets you select a listed reference to look at in the editor and handles cross-reference queries from other system tools, thus integrating the cross-referencer to the rest of the system. The editor uses this facility to provide commands based on program contents such as "find and display the declaration of this procedure." The debugger uses it to provide high-level commands such as "set breakpoints at all assignments to this variable."
- A data-structure display, adapted from the Garden environment[2] and incorporated as a pair of tools. The first tool displays a data structure graphically, letting you pan and zoom to show more or less detail. The second tool lets you describe quickly how the data structure should be displayed by the first tool. These two tools let Field display complex data structures that are similar to the diagrams a programmer draws.
- A debugger, which is comparable to DBX on a Sun. It provides an extended, DBX-like user interface and an internal, message-based interface to other Field tools. Using a separate debugger makes it easier to port Field to other systems, provide a consistent debugger language across systems, and incorporate new debugger commands. Using a message-based interface lets us incorporate different, machine-independent debugging languages later. In Field, you can run multiple debuggers on separate processes simultaneously.
- A graphical, button-oriented debugger interface. The debugger interface lets you easily create new buttons to represent common debugger commands, provides a full transcript of the debugging session, and can display program I/O.
- A flow-graph viewer, which displays a hierarchical flow graph. This tool lets you interactively select interest areas and set up the display accordingly. It interacts with the rest of the system, so you can use the flow graph, obtained from the cross-referencer, to locate routines and calls and to highlight execution.
- A menu-driven interface to the Unix make program.[3] By building my interface on top of make, I could offer many extensions from different versions of make and incorporate tools such as an automatic-dependency analyzer. When requested, this interface performs compilations and in-

Field's principal contribution is its integration framework, which combines the selective broadcasting communication mechanism, an annotation editor, and a set of specialized interactive analysis tools.

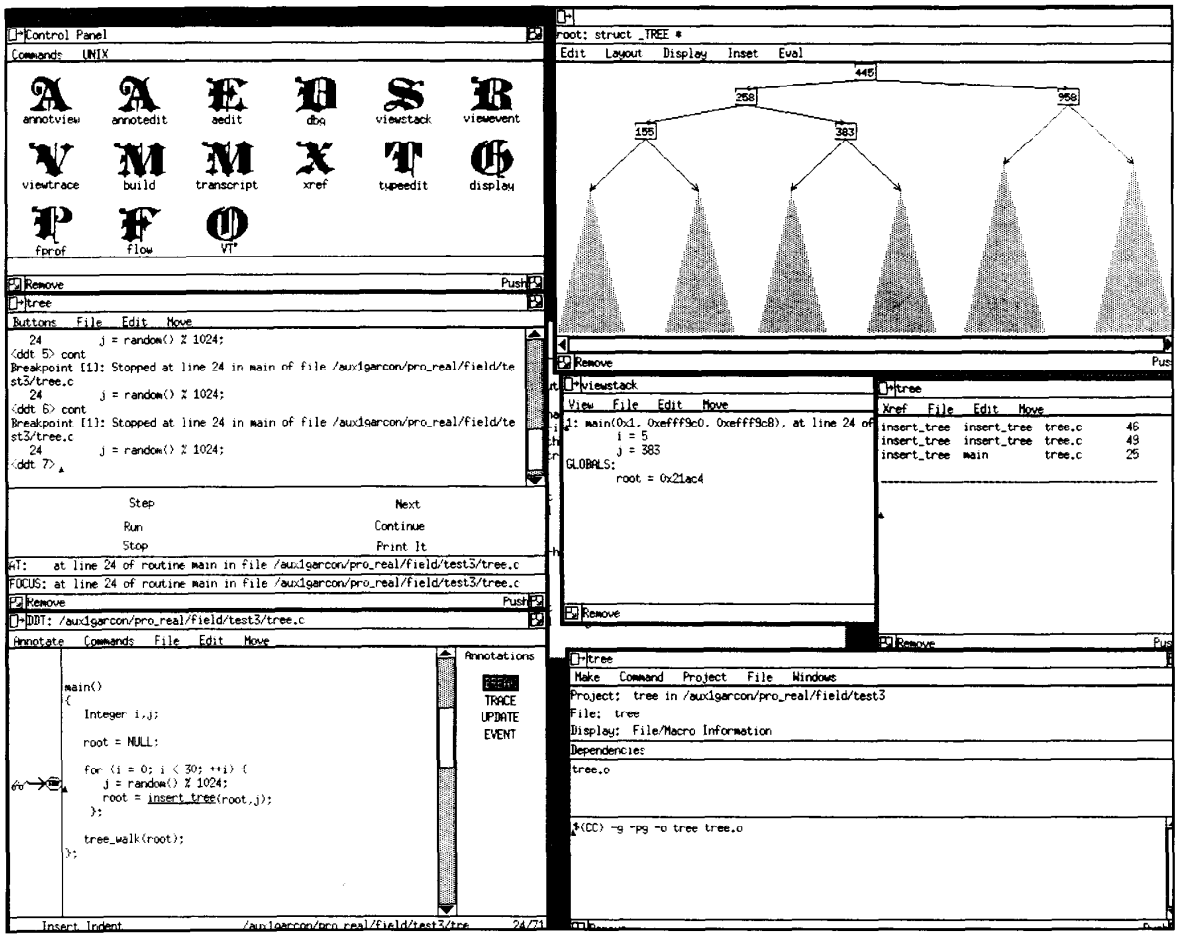

Figure 1. The Field environment.

forms editors of errors.

- A profiler interface, a graphical interface to a slightly extended version of the Berkeley Unix Gprof command.[4] I extended Gprof to make it more interactive and to provide timing information about files and lines as well as functions. This graphical interface lets you selectively view the large amount of information the profiler produces.
- A viewer, a general facility that lets you view different system aspects. I have developed viewers for the runtime stack, variables and expressions being traced, and debugger events such as breakpoints.

Figure 1 shows a snapshot of the Field environment. The window in the upper left is the control panel. It contains icons (Old English letters) for the views and windows you can define. Below it is the debugger-interface window, which shows both a transcript of a debugging session and, below the transcript, the debugger-command buttons.

Underneath the debugger-interface window is the annotation editor, which is displaying the source-code file. To the left of the line of code are three annotations: An arrow indicates the line currently executing, eyeglasses represent the current debugger focus, and a stop sign indicates a breakpoint.

The window in the upper right shows a view of the program's data structure. The program shown here does tree insertion; the tree is displayed in its current state (the dark triangles represent empty subtrees).

Below this window, on the left, is a stack viewer. This displays the current function and line being executed as well as the contents of the local variables at this point. The window next to this is the cross-reference interface displaying the result of a query asking for all calls to the function-insert tree.

Finally, the window at the bottom right is the make interface, which is displaying information about building the system being run.

Integration framework

Field's constraints and goals made it inappropriate for me to adopt existing strategies for integrating tools into a programming environment. As the box on p. 58 describes, other integration approaches are more complex, require more effort than I wanted to devote to the project, make it difficult to use Unix tools, and do not offer the extensibility I wanted. I needed a simpler mechanism that would achieve a high degree of integration while being easily extensible.

I established four criteria for my integration framework, based on an analysis of the desired interactions between the tools:

- The tools must be able to interact with each other directly;
- the tools must share dynamic information;

• the programmer must access the source code though a common editor; and

• the environment must make static, specialized information available to all the tools that need it.

Message facility. The integration mechanism I developed for Field is a selective-broadcast message facility, called Msg. Tools register patterns with Msg to describe the messages that interest them. Tools interact by sending messages to Msg, which rebroadcasts them selectively to those tools that have expressed an interest in them through a previously registered pattern.

This simple concept — a central message server and distributed message handling — is sufficient to put together an integrated programming environment. Its power comes from its flexibility:

• Field passes all messages and patterns as text strings of arbitrary length.

• Msg incorporates and extracts string and numeric arguments according to patterns.

• Messages can be sent either asynchronously or synchronously. Asynchronous messages let the sender continue immediately; synchronous messages generate a string reply to the sender once all eligible servers have acknowledged the message.

Tool interaction. My first criterion is that tools must be able to interact with each other directly. If you want to set a breakpoint in the editor, the editor must be able to issue the corresponding debugger command. If you want to force a recompilation from the editor, the editor must inform the make interface. If the compiler detects errors, Field should change the current editor focus to the erroneous context. If you want to find all occurrences of a variable in your system, you must be able to make a request of a cross-referencing utility. If a variable display needs information about the type or contents of the value it is to display, it must be able to query the debugger.

Field supports such interactions by using the message facility as a command interface to its tools. The most widely used command interface is to the debugger.

```
DDT ACTION system action
  action = INIT | QUIT | KILL | STOP
DDT ASSIGN system var expr
DDT CALL system rtn args
DDT DUMP system from to length format
DDT EVAL system expr
DDT EVENT ADD system file func line expr cond addr act fgs
  act = TRACE | BREAK | CALL | WATCH | MONITOR | event_name
  fgs = 1:internal, 2:external, 4:event
DDT EVENT SHOW system file func line expr cond addr act id
  act = TRACE | BREAK | CALL | EVENT | MONITOR | TRIGGER
DDT EVENT REMOVE system file func line expr cond addr act id
  act = TRACE | BREAK | CALL | EVENT | MONITOR | TRIGGER
DDT RUN system args in out new
DDT SET system what value
  what = INFILE | OUTFILE | USE | WHERE | NEWSYS | DDTOUT | WD
       PRINTIGN | PRINTUSE | CATCH | IGNORE
       PROG | ENDPROG | FORCE_RUN | RUN_ARGS | USER_TTY | ENV
       STACK_TOP | STACK_BOTTOM | STACK_DUMP | STACK_SHOW
STOP_UPDATE
DDT SHOW system what
  what = SIGNAL | RUN | USE | SYSTEM | LOCATION | FOCUS | ENV
DDT STACK system from to dump
DDT STEP system count sig unit
  unit = STEP | NEXT | CONT | SEPTI | NEXTI
DDT SYMINFO system what file func line name
  what = WHICH | WHAT | WHERE | VARINFO | TYPEINFO | LIST_FILES |
LIST_FCTS
DDT VIEW system file func line count stack_delta
```

Figure 2. Debugger command messages.

The debugger's full functionality is available through message-based commands, which lets various tools interact with system execution. For example, the editor sends messages to insert and remove breakpoints, the variable viewers send messages to turn tracing on or off for a given expression, the flow-graph viewer sends tracing requests for the functions it is displaying, and the data-structure display sends messages to query types and values from the running program.

Figure 2 shows the current set of debugger commands. Each command has an alternative form, where the prefix Ddt is replaced by Ddtr. The Ddtr command form is used when output to the user is not needed; for example, when the request is made by a tool other than the debugger interface. Many of these commands serve multiple functions depending on their arguments. I represent unnecessary or omitted arguments with asterisks for strings and zeros for numbers.

Other command interfaces are provided by the make interface, the cross-referencer, and the editor. The make facility's command interface lets any other system component request that a command be executed or a file be compiled. This is used by the editor to request that a file be recompiled and by the debugger to request that the current system be rebuilt.

The cross-referencer's command interface lets any other component query its database. The editor's command interface lets another tool request that the current file be saved, that selected annotations be cleared, or that all annotations be implicitly recreated. It uses the first two requests when externally preparing a source file for compilation; the debugger uses the third to reestablish breakpoints after recompilation.

It is easy to integrate a tool into the environment so other tools can invoke it through the message server when you do it with a new tool interface. Both in adding

```
DEBUG VALUE <system> <file> <line> <var> <value>
DEBUG ENTER <system> <file> <func> <line> <value>
DEBUG EXIT <system> <file> <func> <line> <value>
EVENT ADD <system> <id#> <event_type> <file> <line> <text>
EVENT REMOVE <system> <id# ><event_type> <file> <line> <text>
STOP-ERROR <signal ><file> <line>
DEBUG AT <system ><file> <func> <line>
DEBUG FOCUS <system> <file> <func> <line>
IE <type> <system> <file> <line> <value> ...
DEBUG CLEAR <system>
DEBUG RESET <system>
WHERE <system> <level><file><func> <line> <addr> <args>
WHERE_DUMP <system> <level> <name> <value>
WHERE_BEGIN <system>
WHERE_END <system ><level>
DEBUG SYSTEM <system>
DEBUG NO SYSTEM <system>
DEBUG FINISH <system>
DEBUG START <system>
DEBUG STOP <system> <signal_name>
DEBUG STOP <system> OK
UPDATE <system ><file> <line>
```

Figure 3. Informative messages sent by the debugger.

a new user interface and in integrating the message server, functions representing the various commands exist or are written as part of the expanded tool, and it is straightforward to register message patterns for message-based commands and to call these functions as appropriate when a message occurs.

This is the approach I have taken in the make interface, the cross-referencer, and the profiling interfaces, each of which took about two days of work. In the debugger interface, I took this a step further and at first provided only a message-based interface. I then developed an independent front end that generated message-based commands. This approach will let me develop different debugger front ends with additional functionality and sophistication in the future.

Sharing dynamic information. My second criterion is that the integration framework must let tools share dynamic information. First, different environment components need to know the current execution context. For example, the editor may want to highlight the current line of execution and the line last selected in cross-referencing.

Different components also want to know something about the state of the other components. For example, the editor wants to know where the debugger has set breakpoints so it can inform the user; the make interface wants to know when the editor saves a file so it can initiate an automatic recompilation if requested; the display tools need to know the new values of variables being traced whenever they change; and error messages generated by the compiler need to be associated with the appropriate source code.

Msg is designed to handle such dynamic information. Each tool defines a set of events it deems might be of interest to other tools and sends messages about these events as they occur. Other tools register with Msg those event messages they want to handle and are duly informed when the events occur.

Figure 3 shows the debugger's event messages. The principal messages are sent out whenever the debugger knows — either from the program stopping or from a trace request — the current line of execution, whenever a traced function is entered or exited, whenever a traced value changes, and whenever an event such as a breakpoint is added or removed. It sends out messages describing the content of the stack if requested to do so by, say, the stack viewer. It also lets tools define their own messages to be tied to a program's trace points, a facility used by the algorithm-animation package Tango,[5] a separate Brown research project.

This event-message facility is also used by other tools. The make interface sends out messages for each error or warning the compiler detects so the editor can associate the errors with the program source code. The cross-referencer and flow-graph viewer send out messages when the user clicks on an output reference so the editor can shift its focus to the referenced location. The editor sends out a message whenever a file is opened or closed.

Source access. My third requirement for an integration framework is that it must provide consistent access to the program's source code. Programmers access the source code for many reasons. They edit it to create or change it. They view it to correlate compiler-generated error messages, to see where they are during execution, and to see what program portions the profiler has identified as bottlenecks. They set breakpoints at source statements, trace variables and expressions defined in the source code, and designate source-code components to cross-reference. A fully integrated environment should provide a single access mechanism that can accommodate all these needs and any others that arise.

Field provides such a consistent interface through the annotation editor, which is closely tied to the message facility. The annotation editor starts with a powerful, extensible base editor that provides full editing capabilities. It augments each source line with a set of associated annotations. These serve both as commands that you can invoke for that line and as markers for the line. The current annotation set includes:

- Break, which lets you set breakpoints and shows where breakpoints are set.
- Trace, which lets you set trace points and shows where trace points are set.
- Watch, which lets you trace variables and shows where variable trace points are set.

- Focus, which shows the line the debugger is looking at.
- Current, which shows the current execution line.
- Def, which shows the line last referred to by the cross-referencing tool or the flow-graph viewer.
- Update, which lets you set an update point and shows where such points exist. An update point synchronizes displays such as the stack viewer and the data-structure display with program execution.
- Event, which inserts an interesting event for program animation.
- Error, which flags a line containing a compiler-detected error.
- Warning, which flags a line containing a compiler-detected warning message.

It is easy to add new annotations because they are defined by a text file that is read at start-up, not coded as part of the editor.

Annotations interact with the message system in many ways. How an annotation interacts is specified by string values that represent messages and patterns associated with an annotation. For example, Figure 4 shows the messages associated with the Break annotation. The Msg_Add string corresponds to a message that is broadcast when the user adds an annotation; the Msg_Remove string corresponds to one that is sent out when the user deletes one. The Msg_Set string and the Msg_Unset string correspond to patterns for messages that will cause the editor to add and remove the annotation.

The strings associated with annotations contain escape sequences that the editor fills in to form the message or pattern. These escape sequences can refer to the current line (%L), character position (%C), file name (%F), or to strings associated with this annotation (%V and %T). These strings are defined by their occurrences in the set pattern. Other escape sequences refer to arbitrary strings (%s) and numbers (%d) in a pattern that are to be scanned and ignored.

In Figure 4, the Msg_Add string causes a breakpoint to be added in the file at the given line, using a debugger command from Figure 2. The asterisk in the system field indicates that the breakpoint should apply to all systems that include this source file. The editor fills the file field in with the name of the file being edited. The asterisk in the function field means to ignore the function. The line-number field contains the line where the annotation was requested. The asterisks in the expression and conditional fields mean the breakpoint is unconditional whenever execution reaches this line. Similarly, the Msg_Remove string sends a debugger command to remove the breakpoint.

```
MSG_ADD = "DDTR EVENT ADD * %F * %L * * 0 BREAK 3"
MSG_REMOVE = "DDTR EVENT REMOVE * %F * 0 * * 0 BREAK %V"
MSG_SET = "EVENT ADD %s %V BREAK %F %L %T"
MSG_UNSET = "EVENT REMOVE %s %V BREAK %F %L %T"
```

Figure 4. Messages associated with a Break annotation.

You can define more complex annotations, such as a conditional breakpoint, with %T arguments (%T1, %T2, and %T3). In this case, the user is prompted for the condition. I now use this annotation type to define the interesting events that drive the Tango algorithm-animation package.

An integration mechanism must make static, specialized information available to all tools. To do this, Field uses active servers, which receive requests through the message server and fulfill them.

The patterns associated with an annotation contain the same escape sequences as messages. In this case, however, the escape sequences refer to arguments that the editor interprets. If the pattern provides the file name, it must match the file being edited. If it doesn't, the message will generally be ignored. However, you can make the editor sensitive to this annotation type by telling it that if the file name differs, it should close the current file, open the specified file, and process the message. You do this to keep an editor synchronized with the debugger and to view compiler error messages and cross-reference locations.

An annotation escape sequence must include a line number to identify where the annotation should be set (added) or removed. If the editor is sensitive to the annotation type, it will scroll so this line is visible. The %V and %T escape sequences define the values associated with the new annotation on a set message and ignored on a remove message. The %T values contain text information that you can view later. For breakpoints, this is a string describing the breakpoint; for error messages, it is the text of the message. The %V field is a number. The patterns in Figure 4 correspond to the event messages in Figure 3.

The annotation editor lets different annotations, each with its own icon and color, behave appropriately by letting them have different properties. Annotations can be exclusive to all source files, to a given line, or not at all.

Annotations that are exclusive to all source files let the editor remove the annotation indicating the current line of execution when a new current line is broadcast, even if this new one is for a different source file. Annotations that are exclusive to a given line let the editor discard or merge previous annotations on that line. For example, each line can have only one Error annotation, but it contains all the information about all the passed error messages for that line.

You can directly set annotations like Break annotations, while others, like Error annotations, can be set only through outside messages.

Sharing static information. The fourth requirement for an integration framework is that it make static, specialized in-

```
EVENT ADD %s %3d BREAK %1s %2d %4r
```

(a)

```
static void
handle_add_msg(file,line,value,text)
  String file;
  Integer line;
  Integer value;
  String text;
```

(b)

```
EVENT ADD tree 4 BREAK ./tree.c 24 [4] BREAK at line 24 of file ./tree.c
```

(c)

```
handle add msg("./tree.c",24,4,"[4] BREAK at line 24 of file ./tree.c",-1)
```

(d)

Figure 5. Example of **(a)** message pattern, **(b)** routine declaration, **(c)** message, and **(d)** resulting call.

formation available to all tools. Static information includes system-building rules, cross-reference information, profiling data, and program and execution information. Program information includes information about and descriptions of variable types. Execution information includes the current set of breakpoints and other runtime events. All this information must be available on demand to various system components, and it must be managed so it is kept current.

To do this, I could have stored all the necessary information in a central database. Instead, Field uses active servers. Active servers are Field components that receive information requests through the message server and fulfill them either by dynamically computing the information or by caching it in a local database.

This message interface lets the various tools get necessary information without needing to know how it is stored. Also, by compartmentalizing specialized information, Field simplifies individual servers so it can use existing or slightly modified tools in many cases.

Four Field tools can share static information to some extent. The debugger provides information about program variables, program types, and the runtime environment. The cross-reference server handles queries to a relational database that represents program information. The profiling interface supplies information about functions, files, and lines and supports queries about program bottlenecks. The make interface stores information about dependencies and how to build the system.

Message facility

Field's message facility, Msg, is based on selective broadcasting. Each tool registers with the message server a set of patterns that describe the messages it is interested in. Any tool can send a message to the server, which rebroadcasts it to all the tools that have registered a pattern matching the message. The message facility allows synchronous and asynchronous broadcasting.

I implementated this facility in two parts. The actual Msg server runs as a separate Unix process and communicates via sockets. Each tool includes a client interface that talks to the server and distributes the message from the server to the tool. When a tool sends a message, the client interface passes it to the server. The server passes the message back to the client interface for each process with a pattern matching the message. The client interface then scans the message and takes the appropriate action to send it to the receiver.

I used TCP-domain sockets so Field's tools can reside on different machines and still share a common message server. This lets me integrate the debugging of a distributed system and lets you use one machine to debug a program running on a second.

The current Msg implementation is a 2,000-line C program that is divided equally among the server, the client interface, and a pattern matcher.

Message passing. All Field messages are passed as strings. While this introduces some inefficiencies, it greatly simplifies pattern matching and message decoding and eliminates machine dependencies like byte order and floating-point representation. Similarly, strings represent message patterns.

While it is easy to encode a message into a string, it can be complex to decode it. Therefore, the Msg client interface — not the tools — decodes messages. When a tool registers a message pattern, it provides an entry point to a routine that will handle the message. The pattern identifies not only the message but also those parts of the message that correspond to arguments for that routine and the format of those arguments. A side effect of matching a message to a pattern is that the arguments are decoded into the proper internal form. If the pattern matches the message, the client interface calls the routine associated with the pattern and passes it the decoded arguments. This simplifies the tools' message-handling routines. For example, Figure 5 shows a message pattern, the associated routine declaration, a sample message, and the resulting call.

Message patterns are literal characters that must match the corresponding characters in the message and escape sequences that represent either arguments or generic strings. The format of these patterns is based on the Unix Scanf facility.

Escape sequences consist of a percent sign, an optional description, and an alphabetic character denoting the escape sequence. Escape sequences representing

arguments have the form

```
% [argument_number] [. length]
  type_character
```

Argument_number lets the message give the routine arguments in any order in the message; length allows for fixed-length fields where appropriate; and type_character can be one of

- d, decimal integer;
- o, octal integer;
- x, hexadecimal integer;
- e, floating point;
- c, character;
- s, string;
- r, string representing the remainder of the message;
- q, string in quotes;
- [characters], string consisting only of given characters; and
- [^characters], string consisting of anything but given characters.

The call to define a pattern can also set default values for arguments the pattern does not define directly, so a common routine can handle many messages. For example, the message pattern in Figure 5 accepts the sample message. The first escape sequence in the message pattern, %s, causes the system name tree to be discarded. The second escape sequence, %3d, causes the number 4 to become the third parameter to the function. The third escape sequence, %1s, matches the file name ./tree.c and causes the corresponding string to be passed as the first parameter. The fourth escape sequence, %2d, causes line 24 to be scanned and passed as the second parameter. Finally, the fifth escape sequence, %4r, causes the rest of the breakpoint message containing a definition string to be passed to the routine as the fourth parameter.

Message types. Most of the messages sent in Field are asynchronous and are broadcast to provide potential clients with information. Messages that represent commands must be synchronous and must provide the caller with a reply. Field clients send synchronous messages in much the same way they send asynchronous messages. However, once the message is sent, the client partially blocks until a reply is received.

When it receives a synchronous message request, the message server first determines how many receivers exist for the given message and then sends the message to all these clients. Each client is responsible for telling the server that it has processed the message by returning a string reply. The message server and the associated client interface return control to the initial caller only when they have received replies from all recipients. When this occurs, the client server returns to the sender the first nonnull string reply it received. I designed Field to ignore all but the first reply because it is simpler than accumulating all replies and because I haven't found an occasion where more than one reply was necessary or expected.

While it waits for a reply, Msg will still send new messages to the tool for processing. Field tools must be able to accept and process new messages while waiting for a reply because, while the update and event messages the debugger sends are synchronous (so the tools run in step), the typical reaction of a data viewer when it gets an update message is to request the debugger to get the information it needs to be up to date.

While most of the current Field implementation involves programming-in-the-small, the concepts and integration mechanisms can easily be extended to programming-in-the-large, which I plan to do.

To allow synchronous messages, the message server appends an argument to the end of the argument list for each message. This new value is either negative or the identity of the message to reply to. The message recipient will check this value and, if it is not negative, will use it when issuing a reply. This lets tools like the debugger receive commands that don't necessarily require a reply. By using a specific reply identifier, Msg also lets many synchronous messages be outstanding and be processed by a variety of tools.

This facility is robust as long as the tools do not ignore a reply request. The message facility immediately returns control to the caller if it could not match the pattern and therefore could not send the message. The server also keeps track of what replies are outstanding by what tools. If a tool goes away, normally or abnormally, the message server detects this and simulates a null response to all its pending messages.

I decided against imposing a time-out feature on message replies because the action associated with a synchronous message may require user interaction with another tool and thus can take an arbitrary amount of time.

Field demonstrates a simple but effective way to unite many existing tools in an integrated programming environment. It runs on Sun 3 and Sun 4 workstations; I have ported parts of it to a Digital Equipment Corp. MicroVAX and to an Encore Multimax.

In the Computer Science Department at Brown, Field is being used in undergraduate courses ranging from an algorithms and data-structures course to a software-project course. For the elementary courses, I created a restricted form of Field that provides a limited tool set and that automatically selects the tools of interest. In the advanced courses, students use the full environment.

Field is also being used in research as a debugging aid and as a testbed for developing additional programming aids. So far, our experiences have been generally positive, but it is too early to evaluate Field's effectiveness.

I am continuing to do research with Field. Tango, a project undertaken by John Stasko, now at Georgia Tech, is an algorithm-animation system that uses Field to specify and generate interesting events. Tango is designed to provide access to Balsa-like[6] animations that are easy to create and easy to tie into the user's program.

Over the past six months, I have added to Field support for C++. This support includes full-name mangling and demangling in the debugger and cross-refer-

encer, additional debugger commands, and a graphical class browser. More recently, I replaced the original make interface with a more general one that can handle the intricacies of the newer make version and interact with the source-code context system and SCCS or RCS. I have also been working on improving the layout algorithms used in the flow-graph viewer, the data-structure display tool, and the new class browser and make interface.

I intend to develop more tools for programming-in-the-small. Among the tools I am considering are variable displays that would graphically show variables as dials or gauges, program views that display the program and let you visualize and control program execution, a testing view that would automatically run test cases and display the results, better profiling tools to get a finer level of granularity, and better debugging interfaces based on new debugging languages.

Field could also be extended to handle parallel programming. Field already can debug multiprocess programs and has hooks to handle multithreaded programs. To use these hooks effectively, the debugger has to understand multiple threads of control. Also, I could add many monitoring tools, depending on the parallelism model I choose.

While most of the current implementation involves programming-in-the-small, the concepts and integration mechanisms can easily be extended to programming-in-the-large. I plan to incorporate into Field new and existing Unix-based tools for programming-in-the-large. These will include tools for version control, better configuration management, interface checking, history recording, and bug reporting.

In my experience, Field's performance and degree of integration is more that adequate for most users and most applications. I believe Field's simple mechanisms are a practical alternative for producing a highly integrated programming environment. ❖

Acknowledgments

This research was supported in part by grants from the US Defense Dept.'s Advanced Research Projects Agency, the National Science Foundation, and Digital Equipment Corp. The NSF also provided equipment support.

References

1. J.N. Pato, S.P. Reiss, and M.K. Brown, "An Environment for Workstations," *Proc. IEEE Conf. Software Tools*, IEEE, New York, 1985, pp. 112-117.
2. S.P. Reiss and J.N. Pato, "Displaying Program and Data Structures," *Proc. 20th Hawaii Int'l Conf. System Sciences*, CS Press, Los Alamitos, Calif., 1987.
3. S.I. Feldman, "Make: A Program for Maintaining Computer Programs," *Software Practice and Experience*, 1979, pp. 255-265.
4. S.L. Graham, P.B. Kessler, and M.K. McKusick, "Gprof: A Call-Graph Execution Profiler," *SIGPlan Notices*, June 1982, pp. 120-126.
5. J. Stasko, "The Tango Algorithm-Animation System," Tech. Report CS-88-Z0, Computer Science Dept., Providence, R.I., 1988.
6. M.H. Brown and R. Sedgewick, "Techniques for Algorithm Animation," *IEEE Software*, Jan. 1985, pp. 28-39.

Steven P. Reiss is a professor of computer science at Brown University. His research interests include the use of workstations for programming, programming environments, and visual programming. Before working on Field, he developed the Pecan program-development system and the Garden object-oriented programming environment.

Reiss received a BA in mathematics from Dartmouth College and a PhD in computer science from Yale University.

Address questions about this article to Reiss at Computer Science Dept., Brown University, Providence, RI 02912; CSnet sps@cs.brown.edu.