The System Architecture for CORE:
A Tolerant Program Development Environment

James E. Archer, Jr.
Richard Conway
Andrew I. Shore
Leonard S. Silver

TR79-399

Department of Computer Science
Cornell University
Ithaca, New York 14853

This work was supported in part by a grant from the NSF MCS77-08198.

The System Architecture for CORE:

A Tolerant Program Development Environment

James E. Archer, Jr.
Richard Conway
Andrew I. Shore
Leonard S. Silver

Department of Computer Science
Cornell University

CORE is a program development environment intended primarily to explore a highly tolerant user interface. In some respects the internal architecture is also novel. It permits a highly interactive and supportive user interface to be implemented with processing routines which are essentially oblivious to any user interaction.

Interactive systems typically detect but do not repair errors made by the user in entering and executing programs. Rather than attempt to repair an error, they return to the user and insist that he remedy the problem before proceeding. This opportunity was not available in batch systems, and in that context automatic repair has been found useful. The question is whether automatic repair might not still be of value in interactive systems. Although the benefit may not be as great, it might still be helpful to attempt a repair before returning to the user. Assuming the user has a convenient way to accept or reject the proposed repair this might save time and effort, as well as provide a an effective means of explaining the nature of the error and a plausible remedy. CORE is intended to test this hypothesis.

Automatic repair requires an unusual parser but does not in itself require a particularly unusual overall architecture. Neverthless, the construction of a new system offered the opportunity to simultaneously explore several ideas concerning architecture. These are entirely invisible to the user, and will not confound experiments concerned with the tolerant interface. Other papers describe the user interface and the unique parser employed in CORE; this paper is concerned with the overall architecture. This has the following characteristics:

1. The interactive user interface is completely masked from all but one module of the system. Similarly, the repair capability is localized in a single module.

2. The user is granted generous ability to execute "immediate" statements, as well as stored programs. While the user must be aware of the distinction, the system is not. Both forms of execution are handled by exactly the same internal mechanism.

3. The architecture allows the usual "modes" or "levels" to be eliminated from the user interface. Every user command is available at all times.

4. The architecture is designed around a universal file system. The system itself uses the same file system that is available to the user.

We believe that this architecture is unusually flexible, and at the same time is exceptionally clean and simple. This has facilitated construction and will presumably simplify maintenance.

Neither the repair strategy nor the architecture is peculiar to a particular source language. The initial version of CORE employs a carefully disciplined subset of PL/I called PL/CS [5], but PASCAL [6] or an appropriate subset of Ada [1, 2] could just as well have been used. This PL/I subset is the instructional language used at Cornell, so this choice will make the experimental evaluation of CORE in Cornell classes much easier.

The Command Interpeter

A user views a session with CORE as a sequence of "commands", each followed by the appropriate system response. The user must understand the difference between commands and statements, and the difference between storing a statement and executing it immediately.

Statements are the objects of ultimate concern to the user. They are constructions in the host programming language. The system assists in the development, storage, and execution of statements.

Commands, on the other hand, are the imperatives by which the user controls the actions of the system -- that is, its actions in constructing, saving, and executing statements.

The distinction between statements and commands is emphasized by having entirely different forms for them. The number of commands has been kept very small, and each command is assigned to a special key. Statements are sequences of words generated on the normal alpha-numeric portion of the keyboard.

The distinction between immediate and stored statements is entirely separate from the statement-command dichotomy.

The user interface is perhaps simpler than with many other interactive systems by reason of the elimination of modes or levels. The set of commands has not been partitioned: all commands are accessible at any time.

Many commands allow (or require) an argument, which is a string of characters. A user action consists of the entry of a string of characters, and then the entry of a command -- in effect, specifying how that string is to be treated. For example, the SAVE command causes it to be stored; the EXECUTE command causes it to be executed. It is not our intent here to detail the command set, but this much explanation is necessary to understand that the top level of the system directly reflects this uniform view of commands, and the complete absence of modes.

The top level control is simply the following loop:

```
DO UNTIL (command = QUIT);
    Get user entry ( [text-line], command ).
    Perform action specified by command.
    END;
```

Each iteration of this loop selects one of a set of parallel routines, one for each user command.

Each of these command routines can have an "editing phase" and an "excecution phase". Some have only an edit phase, some only execution, but some have both. For example, immediate execution is accomplished by editing the given statement(s) into a file, and then calling on the execution supervisor to execute that file. The execution supervisor is entirely unaware of the distinction between "immediate" and "stored program" execution.

While the command set is not partitioned by modes, the display screen does change form to reflect the nature of the last command given. There is an "edit screen" displayed when the command is explicitly altering some file, and an "execution screen" displayed when the command caused execution of some statement. But the shift between these different displays is completely automatic and unobtrusive. Each command determines what will be displayed, rather than have the current display determine what commands are accessible.

The error-repair capability of CORE is entirely localized to the module that edits files whose type is "procedure". No other module is directly affected by the fact that the text-line submitted by the user may not be what is actually inserted in the file.

Execution Supervisor

The execution supervisor is described in detail in [4]. It is called by many of the command routines, in some cases as a second phase after a file has been edited. Parameters to the call specify the particular procedure to be executed, and the environment in which to execute. The procedure, the environment, and the input are all given in separate files; the output resulting from execution is placed in other files. As execution proceeds, the environment file is updated to reflect the current state of execution.

The basic structure of the Execution Supervisor is:

```
DO UNTIL (execution is terminated);
    Fetch next statement from procedure file
    Execute instruction (modifying output and diagnostic files as req'd)
    END;
```

Five types of situation terminate the execution interval:

1. Normal program completion.
2. Any input entered at the terminal.
3. Unable to write to file because window is full; waiting for page turn.
4. Input file contains insufficient data; waiting for terminal input.
5. Error encountered in execution.

Operation of Execution Supervisor is synchronous in the sense that it can never be "surprised" in the middle of a statement. The interrupt command simply sets an interrupt flag, which is interrogated only after excecution of the current statement is completed. The last three of these situation do involve aborting execution in the middle of a statement -- for page-turn, data-supply, and execution error, respectively. For data-supply and execution error, the statement causing the interruption is executed again from the beginning when execution is resumed. Only in the case of page-turn does resumption actually continue from the point of interruption.

An interesting characteristic of the Execution Supervisor is that it is essentially unaware of the distinction between "restart from the beginning" and "resume from the point of interruption". It is simply given a procedure, a position, an environment, etc. This gives the system maximum flexibility in the execution of immediate statements (which are executed in the environment of the interrupted procedure), and in the alteration of a procedure while execution is interrupted. Of course, some alterations make resumption impossible, but these are treated like any other run-time error during execution. The interesting problems of interleaving editing and execution are discussed in [3].

The Universal File System

An important characteristic of the CORE architecture is the construction of the system around a comprehensive file system. The same file system is used by the system for internal purposes, by the user to save programs, by the user to supply input data, by the user for auxiliary file operations (PUT FILE and GET FILE), and by the system to store the results of execution.

The file system is also fundamental to the architecture in that <u>every module operates with file objects as both input and output</u>. For example, the Execution Supervisor is called with parameters that specify a certain procedure file, a certain environment file, and one or more input files. The Execution Supervisor modifies the environment file, and adds execution output to one or more serial files. In addition, the information destined for the execution screen (tracing and checking information) is also directed to separate serial files. What portion of this information is displayed on the screen, and in what arrangement, is the function of the Screen Control module, which uses these files as input. The Execution Supervisor has no direct involvement in screen display.

The result is a highly interactive system, but one in which the consequences of user interaction are carefully isolated. None of the main processing modules is complicated by the intricacies of user interaction; all operate strictly on a file-to-file basis.

The only exception to this structure arises in the fact that certain actions (for example "execute") are of indefinite duration and are terminated only by the beginning of the next action (by user "interrupt"). To accomplish this, the main action control performs the execution phase of certain actions as a sequence of short executions, after each of which the "user input" file can be queried to see if a priority action is waiting.

In many cases within the system, transformations are required in order to use information in a file. For example, the internal representation of a program in CORE is a compromise between the form needed for textual display to the user, the parse-tree form required for constructing programs, and an efficient code structure for interpretive execution. Consequently, there are a rich collection of utility routines to translate input to, and output from, files of different types. In aggregate, these utility routines represent a substantial fraction of the implementing code. Logically, these routines give the system's files a uniform appearance and access pattern, making it possible to deal with all of them as if they were a single function.

Even the table-of-contents of the file system is itself a file in the system, and is displayed and edited much like any other file. It is a file of a special type, with translation utilities to display it in intelligible form. This eliminates the necessity of having additional special commands to manage the file system.

The price of this uniformity of view is undoubtedly relatively slow execution. As yet we do not know how large this price might be, but do not regard machine time as the critical resource in the program development process. A substantial increase in machine time could be justified by even a small increase in the programmer's effectiveness.

Controlling the Display Screen

It remains to be shown how the system creates the appearance (and hence the reality) of interaction. As explained above, each processing unit is designed to process individual transactions which cause file contents to be read, interpreted and modified. The burden of making this transaction-oriented process seem both interactive and cooperative is born largely by the Communication Control which is responsible for the contents of the user's CRT display. To effectively provide an interactive facility, the contents of the user's display must be kept current with the files that are represented.

Most user actions modify one or more files. The editing actions obviously modify a file -- under the user's explicit control. However, by far the largest number of file modifications are made implicitly and automatically by the Execution Supervisor in the course of executing statements. The effect on the DATA and OUTPUT files is obvious, but the Execution Supervisor also modifies files that contain trace, check and environment information. Each of the system's file types has both an internal and an external representation. The external representation is designed to communicate to the user the current state of the file and to allow the user to make changes in a logical way. For programs, the external representation is the programming language source; for other file types, appropriate representations are chosen for user communication.

Whenever a file is modified, its internal representation is changed and the Communication Control is informed of the change. If the particular file is being monitored by the user on the screen, the Communication Control must modify the contents of the screen to reflect the file change. If the user makes a direct editing modification to the representation on the screen, the Communication Control must pass the text string which is the modified user representation into the system as the argument of a command, along with specific operational instructions arising in the user's choice of command. The final form of the user's modification is then reflected on the screen when the processing routine receiving the command modifies the affected file. In appearance, the system has responded to the user's action directly, though in fact, it was the file monitoring mechanism which simply made it appear to do so.

The Communication Control deals entirely with the external form of the files and the actual processing routines deal with the internal form. The following diagram depicts the conceptual stages of a file transformation and the routines performing the transformations.

```
--------- ----------------------------------- ---------------------- ----------
        | |                                 | |                    | |
--------| | --------   --------   ---------| | --------   --------| | --------
|       || | |      |  |      |  |        || | |       |  |      || | |       |
|       ||-|-|      |--|      |--|        ||-|-|       |--|      ||-|-|       |
|       || | |      |  |      |  |        || | |       |  |      || | |       |
--------| | --------   --------   ---------| | --------   --------| | --------
  file  | | virtual  represented  display| | window     virtual| | actual
        | |  text       text       text  | |            screen | | screen
        | |  file     segment    segment| |                    | |
--------- ----------------------------------- ---------------------- ----------

       ==========>  Decoder  ========>                    ====> real CRT
File System                          Communication Control
       <==========  Editor   <========                    <==== user entry
```

Consider the file interactions involved in changing a line of the user's program. The program is stored in a file in a form that facilitates interpretive execution, but is sufficient for displaying the program. To display the program on the screen, the Communication Control must have access to sufficient program text to display the appropriate segment. This is called the "display text segment", and is created by "decoding" (translating from internal to external form) each of the relevant program lines. Depending on a number of factors (including recent usage history and available memory), the Communication Control may already have text representations of some of the program text beyond the lines required for the current display. In the extreme case, the entire program might exist as text (in addition to the internal representation). The "virtual text file" is what the user thinks the file looks like, and is equivalent to the "represented text segment" when the entire program is represented as text.

To perform a direct, screen-edit modification of the program, the user moves the cursor to the appropriate line and inserts or deletes characters to change the line to the desired form. When the user completes the change and indicates that the change should be entered (or executed or inserted elsewhere in the program), the Communication Control merely appends the appropriate command character to the line as modified and passes it on to the Command Supervisor just as if the user had entered the entire line as an explicit command. The Command Supervisor routes the request to the editor for the appropriate file type. The editor converts the "new" line into internal form, incrementally parsing the new text, converting it to internal form, and updating the program file. As part of the file update, the file system notifies the Communication Control which line(s) of have been modified. The Communication Control then requests text versions of the lines for display on the screen. The parsed, formatted and corrected segment then appears on the screen. As part of the history of the modification, sufficient information is kept to allow the user to back out of the change with a single "reject" key. The original text entry is then available for modification.

By following this uniform access path, it is possible to allow any system command to be entered at any time. The automatic update notification from the file system to the Communication Control greatly simplifies the processing routines. Processing routines only need to read and write files and can ignore the form and content of the display screen. Since each system routine processes a series of independent transactions, the order of function usage is not enforced.

If this system is to appear natural to the user, the Communication Control must have the ability to allow the user to move around on the screen and make changes at will. These immediate modifications to the screen are written to a file which which is repeatedly read by the Command Supervisor. The Command Supervisor is responsible for correctly routing transactions to processing routines. In order to request information from the user, a processing routine must return to the Command Supervisor and wait for the next transaction. Note that the removal of command context (in conjunction with the ability to stop sessions in the middle) from the processing routines requires that the files used to represent system objects be complete and consistent whenever the processing routine is not in control.

The Communication Control assembles input command lines and displays output window lines. The characteristics of the actual terminal are known only to Communication Control, which is solely responsible for handling differences between the logical and physical terminal devices. The implementation of all other system modules is insensitive to changes in terminal characteristics.

The logical terminal is a simple input device and a powerful output device. Logical terminal input lines are considered to be character strings of indefinite length, each representing a complete user input. All line editing features are handled by Communication Control, invisible to the user in the same way that they would be on an intelligent terminal. The logical output screen consists of a (potentially very large) set of file windows. Communication Control maintains a formatted screen consisting of those windows that "should" be displayed at any particular time. The selection of windows is determined by the current system state, user-selected parameters, and available screen space. The logical screen consists of lines of indefinite length made up of a wide variety of characters in multiple fonts. The communication control is responsible for displaying these characters in the best manner for each terminal.

The isolation of all terminal communication in the Communication Control makes it possible for the other modules to operate in transaction-processing mode, with almost no provision for the interactive nature of the system. This not only simplifies the overall system implementation, it makes it possible for the system to easily adapt to available hardware. The bandwidth of the terminal connection is important for a system which keeps a relatively full screen image up-to-date. For slower terminals, it is possible for the Communication Control to show parts of the current window set only as they are needed, filling the screen as "needed" instead of automatically. For intelligent terminals, some or most of the Communication Control can be moved to the terminal. In the extreme case, the text of all currently accurate window segments could be retained in the terminal, requiring only requests for new or changed line contents to redraw the screen.

References

1. "Preliminary Ada Reference Manual", SIGPLAN Notices 14, 6, June 1979.

2. Archer, J., "An Instructional Subset of the Programming Language Ada", Technical Report 79-395, Department of Computer Science, Cornell University.

3. Archer, J., "Interleaved Execution and Modification", Technical Report 79-398, Department of Computer Science, Cornell University.

4. Archer, J.E., and A. Shore, "A Program Development System Execution Supervisor", Technical Report 79-397, Department of Computer Science, Cornell University.

5. Constable, R. L. and R. W. Conway, "PL/CS -- A Disciplined Subset of PL/I", Technical Report 76-293, Department of Computer Science, Cornell University.

6. Jensen, K. and N. Wirth, PASCAL User Manual and Report, 2nd ed. Springer-Verlag, 1975.