TECHNIQUES TO FACILITATE THE DEBUGGING OF CONCURRENT PROGRAMS

by

HONG YAU CHUA

B. S., Kansas State University, 1983

A MASTER'S THESIS

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1986

Approved by:

R A McBride

Major Professor

TABLE OF CONTENT:

ABSTRACT:

The differences between sequential and concurrent programs are identified. These differences dictate how the design of a debugger for concurrent programs must differ from a sequential debugger. Different techniques to facilitate debugging of concurrent programs are discussed. The implementation of a tracing facility for Concurrent C programs is presented. This implementation enables all of the processes in a Concurrent C program to be presented by icons on a CRT which keeps track of their individual process state. Furthermore, the ability to breakpoint a process and at the same time suspend all the other existing processes within a Concurrent C program is implemented in this facility.

LIST OF FIGURES:

LIST OF TABLES:

ACKNOWLEDGEMENT:

The well polished and presentable form of this thesis is indebted to Dr. Richard A. McBride for the time consuming reviews of this thesis and many helpful suggestions. I also greatly appreciate the helpful guidance and suggestions from Dr. Virgil E. Wallentine and Dr. David A. Gustafson. Without the financial support from my parents, brothers and sisters during my undergraduate career, this thesis will not be a reality. Last but not least, the moral support and encouragement from my friends are greatly appreciated.

Chapter 1

Introduction

Generally, a computer program is developed through a sequence of steps. These steps can be divided into four phases namely:

1) understanding the problem,
2) designing a solution,
3) implementing the solution and
4) testing the solution.

The first phase is concerned with the specification (objective) of a solution. Understanding a problem is accomplished by studying the documentation surrounding the problem in detail so that a programmer can comprehend thoroughly what is to be solved. The design phase is concerned with how the specification can be achieved. This is normally done by laying out the logical flow or general structure of a program. Furthermore, it may involve selection of data structures to make the program more efficient so that the program executes faster or requires minimal memory space. The implementation phase deals with the conversion of the design, derived during the second phase into real programs. This third phase involves coding and compilation. The

testing phase is used to ensure that the specification is accomplished. This involves the execution, verification and debugging of a compiled program.

The debugging process during the testing phase of a program development is the focus of this thesis. In fact, this thesis presents ways to make a programming environment conducive for concurrert programming.

In order to focus on debugging, one needs to understand some of the terminology associated with debugging. When a program is said to contain a bug, then that program has an error. Debugging is the process of locating the cause(s) of an error and fixing the error(s). Debugging is necessary when the cause of errors is not apparent to a programmer. But how does one know that there is an error in the program? Errors are discovered when the behavior of a program does not conform to its specification (assuming the specification is correct).

In order to study the behavior of a program, test data are needed. With a 'proper' formulation

of test data, all of the desired behavior of a program can be reviewed. Unfortunately, the formulation of 'proper' test data which reveals all of the behavior patterns of a program is very difficult to accomplish when the program is complex.

Concurrent programs are always more complex to debug than sequential programs. This is because the behavior of a sequential program is independent of time whereas a concurrent program's behavior can be time dependent. A concurrent program is composed of two or more sequential programs that may execute concurrently as parallel processes. During their execution, these processes may occasionally interact with each other. The act of occasional interaction via synchronization and cooperation is called asynchronism. The processes involved in this interaction are called asynchronous processes. A classical example of asynchronous processes involves producer and consumer processes where a producer produces as fast as possible until it has to wait for the consumer to catch up with it or vice versa.

An example of a time dependent program is illustrated below. Let x and y be the two processes which share a common variable, s. The pseudo Concurrent C [GEHANI & ROOME 84a] code for the two processes is shown in Figure 1.0.

```
Process x:                    Process y:
{ local variables             { local variables
        X1, X2;                       Y1, Y2;

  X1 = s;                       Y1 = s;
  X2 = X1 + 10;                 Y2 = Y1 + 20;
  s  = X2;                      s  = Y2;
}                             }
```

Figure 1.0: An example of time dependent program in pseudo Concurrent C

Assume that the initial value of s is 1 and that this variable is shared by both processes. Also, assume that both processes start at the same time. Now, consider that process y executes faster than process x, i.e., y has executed s = Y2 before x executes X1 = s. In this scenario, X1 gets the value of 21 and X2 is set equal to 31. On the other hand, suppose process x executes faster than process y, i.e., x has completely executed before y is started. In this case, X1 gets the value of 1 and X2 gets 11. Thus the values of X1 and X2 are different in both scenarios because of the speed of execution of

each process. This phenomenon is called 'race condition.'

Another possibility of 'race condition' is the interleaved execution due to the lack of atomicity (see Figure 1.1). In this case, the initial value of x is 50. But the value of x in process 1 can be 40 or 20 depending how the assignment statements are executed. The value of x in process 1 can be 40, if the assignment statement is executed first, without interleaving with process 2's assignment statement.

```
                x = 50;

Process 1:                  Process 2:

  x = x - 10;                 x = x - 20;
```

Figure 1.1: An example of interleaved execution due to the lack of atomicity

The major difference between concurrent and sequential programs involves synchronization. Synchronization is the constraint imposed on autonomous processes within a concurrent program to coordinate them and to keep them from interfering with each other. This is often called the timing constraint [ANDREW & SCHNEIDER 83], [DEITEL 84].

With an understanding of the terminology of debugging and concurrent programming, the reader may now proceed to the gist of the paper. To aid the reader, here is how the remainder of this thesis has been organized. In chapter 2, a historical survey of debuggers is presented. Also, four existing debuggers are presented in order to show different features that are commonly found in debuggers. Chapter 3 covers "low-level" debugging including a discussion of hardware supports for debugging. Chapters 4 presents current techniques from different areas in computer science that might be incorporated into a debugger. In particular, applicable techniques from databases and techniques associated with artificial intelligence are covered in chapter 4. Chapter 5 deals with the difficulties of debugging a concurrent program, and includes ideas and approaches for concurrent program debugging. The last chapter, comments on future research in debugging especially for concurrent programming.

Chapter 2
History and survey of debuggers

Normally, when you talk about a subject, it would be ideal to cover the history of that subject. History is a useful tool in helping to diagnose what has gone wrong and try to teach us how to avoid the same pitfall again, and possibly providing a solution wher a researcher follows the same path as a prior one. This diagnosis may also stimulate different avenues of approaching how to solve similar problems. As a result, this chapter starts off with an account of the history of debuggers and then a survey of four existing debuggers.

2.1: History of debuggers

In the 1940's and 1950's, debuggers were nonexistent. This was partly due to the lack of hardware and software technology. Furthermore, computers were not used on a large scale and those who used them were either professionals or researchers. At that time, an assembly language was considered an easy language to use as compared

to flip-flop switches. It was only in the middle 1950's that high level languages were invented. With the understanding of constructing a compiler and more demand for 'user friendly' programming environments, debuggers started to emerge in the early 1960's and boomed in the 1970's. Initially, storage dump and output traces were the only ways to help in debugging a program. With better developed hardware, hardware interrupts were possible which permitted stepping through a program. From single-step breakpointing, a more sophisticated approach was incorporated and resulted in conditional breakpointing. Conditional breakpointing is a facility in which a breakpoint will be triggered if the associated condition(s) is met. In order to provide an interactive debugger with user friendliness, the symbolic debugger were developed. A symbolic debugger is capable of interpreting the symbols employed by a user in his/her program during a debugging session, whenever a reference to a storage location is desired. Further explanation of symbolic debugger will be provided in the next section.

In the 1970's, almost all of the debuggers

for a language were developed after the programming language itself and this created some ad noc ways of implementing a debugger. Moreover, all those debuggers were for sequential programs only. In fact, more recent debuggers which were developed in parallel with a language are more powerful than their ccunterparts which were developed after a language was already implemented. This power is due to features like: switching a debugging session to another terminal (to save important debugging information, as when the terminal is limited to a certain buffer size), scoping of debugging commands (to elimirate lengthy commands, and express commands unambiguously if variable names are duplicated in multiple procecures/functions), and adopting the same language constructs for debugging commands that are present in the associated high level language (to eliminate learning a new set of syntax). Furthermore, if a system is designed with the consideration of incorporating a debugger, then tnat debugger is more powerful because special features to facilitate the debugger have already been incorporated into the system.

Only in the late 1970's did debuggers for concurrent programs start to exist. This is partly due to the lack of hardware support before and partly because there was not much demand for concurrent programming with high-level languages.

Currently, there are many debuggers (both sequential and concurrent) tailored for specific systems, but most of them are not designed with portability in mind.

2.2: Symbolic Debugger

A symbolic debugger is one which can understand symbols employed by a user in his/her program during a debugging session instead of storage addresses. These symbols can be variables, constants, procedure or function names, etc., which are declared within a program itself. In order for the debugger to understand all those symbols, a symbol table is required in order to interpret them. Normally, this table consists of the attributes of each symbol. Attributes like type of declaration, addressing, value, range of values, upper and lower bounds of array subscripts are usually kept in the table.

Furthermore, a symbol table is generated during a compilation if it is requested by the user. The table's generation is optional since the symbol table could be very large depending on the size of the program and is useless if debugging is not needed.

2.3: Survey of Four Debuggers

The four debuggers to be discussed are UNIX's adb and dbx, Tandem Nonstop II's INSPECT and a PASCAL32 debugger. The reasons for choosing these are:

1) they reside in different environments and,
2) they exhibit different features, especially from the aspect of user-friendliness.

A comparison table for the four debuggers is shown in Figure 2.0. The features chosen for comparison are partly based on criteria for interactive debugging suggested by Seiner [SEIDNER & TINDALL 83].

Debugger	adb	dbx	INSPECT	PASCAL/32
User frienoliness		X	X	X
Easy to use		X	X	X
Descriptive Operators		X	X	X
Source level debugging		X		X
Symbolic debugging	X	X	X	X
Interactive debugging	X	X	X	X
On-line help		X	X	
Low-level debugging	X	X	X	X
ASCII display		X	X	X
Terminal independent			λ	
Concurrent Program Deougging			X	X

Figure 2.0: Comparison table of the four debuggers

2.3.1: adb

adb [TUTHILL 85b] was the first general purpose debugger on UNIX system. It is not user friendly at all and the fundamentals of its usage are quite hard to grasp. As usual, the accompanying documentation for these UNIX system programs are very brief and quite difficult to comprehend if user is not familiar with the system

[UNIX 83]. The syntax of the debugger is quite foreign to UNIX's host language, C, and other high-level languages like PASCAL, FORTRAN, LISP, etc. The syntax is closer to an assembly language with weird symbols like '.', '*', '@', '?', etc. But it allows user to set breakpoints, define action to be taken when the breakpoint is reached, display tne content of memory, display the stack which keeps track of call sequences, etc. Programs can be debugged without the insertion of any user interface routine in the program.

adb provides a controlled environment for debugging. When a program needs to be debugged, it must be compiled and the object file produced must then be submitted to aob. Symoolic debugging is available only if a program was compiled with symbol table option includeo. An analysis of a crashed program can be performed when a core image of that crashed program is available. Aob enables a user to display the contents of physical registers in a manner reminiscent of a memory dump in an I8M environment. For details about 'adb', refer to ADB(1) of UNIX Programmer's Manual [TUTHILL 85b]. NOTE: novices are strongly

advised NOT to use adb at all because it can become frustrating very quickly. Novices should use the next debugger, dbx instead.

2.3.2: dbx

dbx is a debugger capable of debugging a program at the source level under UNIX operating system. Currently, it supports Fortran 77 and C programs. It is a symbolic debugger. dbx is also a source level debugger, which means that interaction with source text during debugging is possible. For irstance, displaying source text, during debugging to see which line the program has just executed and which line is to be executed next is permitted. This kind of technique is a step towards instituting a "paperless programming" environment. For a truly "paperless programming" environment, a multiple screens capability is essential on a terminal (CRT) to permit different portions of a program to be displayed at the same time on the CRT screen. This feature would eliminate the shuffling of a program listing.

dbx uses English verbs like 'run', 'trace', etc., to name and introduce its functicns. With

this method of naming, a user can recall a function easily. Tracing of statements, variables and proceoures or functions are possible and optional predicates associated with them. Breakpointing at statement, variable and procedure/function levels are also available. The reporting of all trace and breakpoints that are set in a program can be invoked. Of course, modifying and displaying of any symbol's value is allowed and an inquiry can be made about the attribute of any symbol.

If a user has the desire to debug the program at machine instruction level, he/she is permitted to do so. User friendly commands like on-line help for a synopsis of dbx command and 'aliasing' for shortening command expression are also at user's disposal. The major difference between dbx and aob is that dbx has a higher level user interface than adb [TUTHILL 85a]. For more detail about dbx, refer to DBX(1) of UNIX Programmer's Manual.

2.3.3: Tandem Nonstop II Debugger, INSPECT

The symbolic debugger of Tandem Nonstop II

computer system, 'INSPECT' enables a user to initiate a debugging session at a terminal and then routes the input or output of that debugging session to a different terminal on the same or different Tandem system(s). This capability is accomplished by the concept of a program running as a process and 'INSPECT' running as a process also. INSPECT is as user friendly as dbx except that it does not allow source text to be displayed during the deougging session. However, it has other good features like FILES (which displays the status of opened files within the program), TERM (which can be used to switch INSPECT session to different terminal), HIGH and LOW (switches INSPECT between high-level programming language and assembly language mode), RADIX (sets the base for numeric conversion), SCOPE (sets the scope for subsequent commands i.e., allow a query about the value of variables outside the current scope of a section of code) and COMMENT (adds comments to the log file or to command files). As it is a symbolic debugger, a symbol table is generated during compilation whenever the '?SYMBOL' directive is included in tne source code. For more detail about 'INSPECT', refer to INSPECT Interactive Symbolic Deougger User's Guide [TANDEM

831.

2.3.4: PASCAL/32

PASCAL/32 is a symbolic debugger for sequential and Corcurrent Pascal. It has almost the same capabilities as dbx and INSPECT except for a few features which are unique to dbx or INSPECTS. Nevertheless, it has features for debugging concurrent programs. "SUMMARIZE" is a command which produces a summary list of the status of all active processes. The summary consists process PCB address, the current process state, the current program counter, global base, etc. Another concurrent program debugging command is "TRACEBACK" which provioes a traceback listing all of the active processes starting from the main process (which initiated all of its child processes).

2.3.5: Summary of the four debuggers

As time elapses, debuggers are becoming more user-friendly as proved by the evolution from adb to dbx. This is so because user-friendliness is demanded in all software, and more advanced technology is available now than before.

Furthermore, debuggers seem to be easier to use and a "paperless programming" approach (as evident by source level debugging) can be adcpted. However, each debugger has its own salient features. These features are system dependent. On the whole, debuggers are thought to be a piece of software which facilitates a user in debugging a program. Thus, all debuggers are built with the intention to help a user as much as possible in debugging his/her program.

2.4: Future Debuggers

Besides symbolic debuggers using the operating system capabilities of the environment, debuggers are beginning to make usage of different tools in recent years. Tools such as databases, multiple-window software/hardware, graphical hardware/software and knowledge-bases or AI have been incorporated into debuggers. With the incorporation of databases and knowledge-bases, back-tracking and English sentences for queries during debugging are now often possible. On the other hand, multiple-window layouts and graphical features enable programmers to debug their programs with little or no hard-copy. With all

these new ideas, future debuggers should be very easy to use.

Unfortunately, little work has been done on debugging concurrent program. Will the knowleoge of debuggers for sequential programs help in the construction of an adequate debugger for concurrent programs? This problem is addressed in the subsequent chapters.

Chapter 3
Low-level debugging

"Low-level" debugging is concerned with memory dumps and the interactive examination of memory [HAMLET 83]. At this level, a user is often overwhelmed with data and must be able to extract the pertinent data to fulfill his/her requirement, otherwise the data are useless. The difference between "low-level" and "high-level" debugging is the ease with which relevant data can be extracted by the user. For instance, in "low-level" debugging the user has to know the relative position of a variable declared in a program in order to locate its content from the associated memory dump of data segment. On the other hand, in "high-level" debugging all the user has to do is to specify which variable is to have its value displayed.

3.1: Software Support

Software Support is concerned with software packages available to users which can facilitate in the debugging of their programs.

3.1.1: Low-level debugger

Both the memory dump and interactive examination of memory rely on breakpointing. Breakpointing is the temporary halting of an executing program. Breakpointing is usually accomplished by the insertion of a "breakpoint" instruction, which is supported by the hardware in the appropriate segment of the executable code of a program.

In the case of a memory dump, the breakpoint is set immediately after the point where the program needs to be diagnosed and the state of the program is preserved for dumping. At that time, the program is also terminated and the state is printed (dumped). However, in the case of interactive examination of memory, conditional breakpointing is necessary because the setting of breakpoint(s) dynamically by a user during the debugging session is possible.

The approach to implementing conditional breakpointing is hardware dependent. All of the approaches use some sort of exception handling which is available on the hardware.

Unfortunately, breakpoints are extremely difficult to embed in code produced from macros because they represent predefined sets of code [GENTLEMAN & HOEKSMA 83]. The same is true for library routines called by user's program.

3.1.2: Implementation of low-level debugger

Exception detection is essential in hardware because potential disaster due to an execution error of a user's program can be avoided in advance. To preserve the integrity and to enforce the robustness of a system, some restrictions are imposed by the operating system on a user's program in the form of exception conditions. Whenever one of these conditions is fulfilled by a user's program, the hardware automatically generates an irterrupt. When this interrupt is triggered, the supervisor takes control of the CPU and executes the appropriate interrupt handling routine and a user's program is terminated.

This interrupt handling routine can be user-defined i.e., a user can specify what action(s) will be taken when an interrupt occurs. Actions

such as noting where an error occurred and orinting out error messages are possible. But note that only actions which can be performed by a user's program are alloweo; otherwise cascading interrupt handling calls can result. The system or default interrupt handling routine usually gives an exception code ano gracefully terminates the user's program. Examples of exceptional events are: illegal irstruction (i.e., an instruction unknown tc system), privileged operation (i.e., an instruction which must be executed through the operating system), addressing (i.e., the specified address of instruction or data for storing and fetching is out of bound), specification (i.e., something is wrong with the way in which an operand is specified in an instruction), data (i.e., illegal data format for certain operation), arithmetic overflow and underflow, and so forth [STRUBLE 75].

Since a memory dump is a copy of the program's state at the time that program is halted by an interrupt, the dump is normally in the form of numbers (of certain base) instead of descriptive messages with appropriate values

associated. This is because not much conversion needs to be done to convert the machine representation of a program state to a dump. Within this dump, there is a completion code (normally part of Program Status Word, PSW) which indicates the reason for halting the program. The contents of registers (general and floating-point) are also available. Through the values of these registers, a better insight of what that program was doing and what it had done can be achieved, but only with a lot of sweat. Furthermore, knowledge of the usage of each register must be known.

3.2: Hardware Support for Low-level Debugging

Since haroware is the foundation of software, it is importart to investigate how hardware supports can be employed to facilitate software support for debugging.

3.2.1: Available Hardware Tools

Some hardware supports which are available include: probes to detect signals to and from the microprocessor, a fifo buffer for logging events, comparators for matching patterns (used for

conditional breakpointing), timers for performance measurement and counters for measuring events [GENTLEMAN & HOEKSMA 83]. Most hardware tools have been used for logic level and functional unit level hardware debugging.

Commercial in-circuit emulators for microprccessors are now available [INTEL 78], [TEKTRONIX 81]. These emulators are just like probes but run unoer a secondary computer (microprocessor). Such emulators can monitor signals i.e., generate signals to the microprocesscr as if they were generated from the rest of the circuit, or generate signals to the rest of the circuit as if they come from the microprocessor, or they can generate both types of signals. These emulators have been used widely for debugging simple microprocessor applications such as a protocol for peripheral control. Nevertheless, the application of these emulators to high-level debugging is not very clear because they are not widely used. Certainly, usage of in-circuit emulators is a good candidate for concurrent debugging because it supports an independent process running on a different processor rather than the system executing the

concurrent processes that are being debugged. Thus, with the inclusion of this independent processor, the behavior of a concurrent program can be preserved. In other words, the corcurrent program can be debugged without any modification at all.

For instance, suppose a concurrent program is running on a system and an emulator is integrated (into that system), which "listens to" the transactions occurring between the target processor (the CPU which runs the concurrent program) anc other parts of the system. With this setup, every instruction executed by the system is "overheard" by the emulator. Furthermore, the emulator is an autonomous system, its listening will not affect the targeted program because the emulator is not interacting with the running program. Consequently, the concurrent program can run uninterrupted as if the emulator was not there as opposed to a conventional debugger which is embedded (i.e., the debugger is running on the same processor as the concurrent program).

This mere ability to "listen" without any interference by the emulator will preserve the

behavior of the concurrent program. This will definitely nail down the bug residing in the targeted system because there should not be any side effect produced by the emulator. Since we are interested in debugging a concurrent program, whatever tools we employ must not create any problem so that the program can be correctly debugged; otherwise the bug(s) is not eradicated. The next section covers a detailed application of such emulator in a real environment.

3.2.2: Usage of Hardware Tools

With that brief introduction to the hardware supports for debugging, we are now ready to see how they can be incorporated into a system for debugging purposes.

The use of another computer to monitor real time programs is described in [PLATTNER 84]. He uses a probe for "listening" to the transactions occurring between the target processor and other part of the system, and a dual port (phantom) memory which is a memory that can be accessed by both monitor process and target processor interface. The phantom has the same word size and

the same adcressing range as the target processor. The structure of his moritor is given in Figure 3.0.

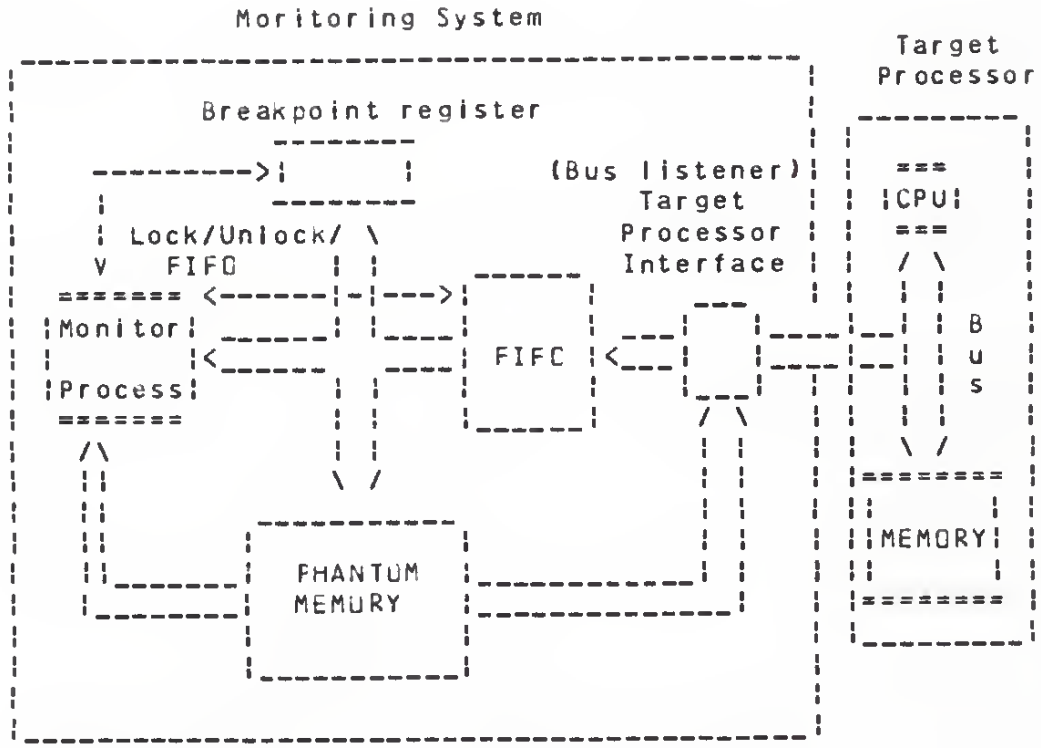

Figure 3.0: Real-time monitor hardware.

With the target processor interface in the monitoring system connected to the target processor, all memory transactions generated by the target processor's CPU are "listened to." The information involved in the memory transactions viz., its address, content ano whether the location is read/write, are queued in the FIFO. The queued data then flows into the phantom

memory. The monitor process may read the content of the phantom memory at any time. It may lock the flow of data from the FIFO to the phantom by locking the FIFO. During such a period, arriving information to the FIFO is queued. If the FIFO is unlocked then data again can flow to the phantom memory. Since the monitor process is "observing" the targeted processor state and evaluating the predicates (commands) that were submitted by an operator, the monitor process has to interpret on-line the low level (close to machine) information available in the phantom memory and perhaps at the output of the FIFO in order to construct a high level (close to target program source level) image of the target process' state. The interest of now to construct a high level image is a totally different topic from our discussion of concurrent debugger but for interested reader please refer to [PLATTNER 84]. However, to speed up the monitor process, a multiple breakpoint register is connected to the output of the FIFO. This breakpoint register reports to the monitor any memory transactions referencing a location belonging to a previously defined set of memory locations for breakpoints by a programmer who is observing the behavior of the

target processor.

After the integration of these componerts, how are we going to know that the monitoring system will work correctly? Definitely, testing of the monitoring system is also required. If there is any error encountered during the testing then debugging the system is necessary. Approaches to debugging a hardware system without having to resort to expensive commercially available cevelopment systems is described in the monograph compiled by Noordin Ghani and Edward Farrell [GHANI & FARRELL 8C].

Chapter 4
Current Techniques In Enhancing A Debugger

As time elapses, more technology is invented. Consequently this new technology may be used to enhance existing systems if the technology is suitable. With the advancement of database systems, graphics, and knowledge-based (expert) systems, radical enhancements for debuggers seem promising.

In this chapter, we investigate the prospect of such technology being used to enhance a debugger. First, we cover how the knowledge from database systems can be employed in the debugging process. Secondly, graphics technology is reviewed and thirdly, knowledge engineering is investigated.

4.1: A Database Model of Debugging

A database is a repository of highly organized information which provides for easy retrieval and maintenance of the information. The system which facilitates the retrieval and

maintenance of highly organized information is called database management system (DBMS). Retrieval and maintenance of information in a database is accomplished through the query and update facilities provided by a database management system.

In debugging a program, it is necessary to have access to information about the source program and the execution state of that program. If we could somehow translate the necessary information for debugging a program into a form that could reside in a database, then debugging could be accomplished by performing queries and updates on a database representing a program.

Currently, there exists such a programming system, called OMEGA [POWELL & LINTON 83], which stores all the information of a program (parse tree, symbol table and so forth) into a relational database system. In this environment, a program is treated as a group of objects from different classes such as variables, statements, procedures and so forth. In order to relate these classes together in an effort to represent the semantics of a program, relationships among these

classes need to be established.

For example, some of the relations could be

proceoures (name, parameters, statements)
statements (statement_category, symbols)
variaoles (name, type, value).

In this example, the 'procedure' relation has three attributes viz., name, parameters and statements. Each tuple of the relation corresponds to a procedure in a program. A procedure has a name, some parameters (possibly none) and an ordered list of statements which represent the body of a procedure. In the 'statements' relation, we have a statement_category (assignment, control, invocation and so forth) and a list of symbols making up a statement. On the other hand, the 'variables' relation has three attributes. The first attribute is "name," which can be a procedure name or a symbol in the symbol list of 'statements' relation. The second and third attributes of 'variables' relation hold the type property of a variable (such as integer, real, boolean and so forth) and the value of a variable if it has one.

A query language is used to formulate questions about information in a database. For instance, if we want to find all the statements where procedure 'bugs' is invoked, we could express the query as follows in a naive version of QUEL (the query language for INGRES) [INGRES 85] as follows:

```
range of p is procedures
range of s is statements
retrieve (s.all) where
   s.statement_category = 'INVOCATION'
        and s.symbol[1] = 'bugs'.
```

In this example, we assume the procedure name is stored in the first position of the symbol list if the statement is an invocation statement. Furthermore, a user is expected to know how the relationships are set up, especially the attributes in each relationship and how the relationships are linked together to represent the entire program. As a result, a user has to know the detail of the database like the name of each relationship, the key to each relationship. Also knowledge of the compiler might be helpful especially in understanding how the relationships are constructed out of the grammar of the

programming language.

In order for a user to debug his/her program, the information about the program's state must be incorporated into the database so that a query can be performed. However, dynamic information is NOT actually stored in the database, but accessible only from the memory.

The organization of debugging in OMEGA is depicted in Figure 4.0.

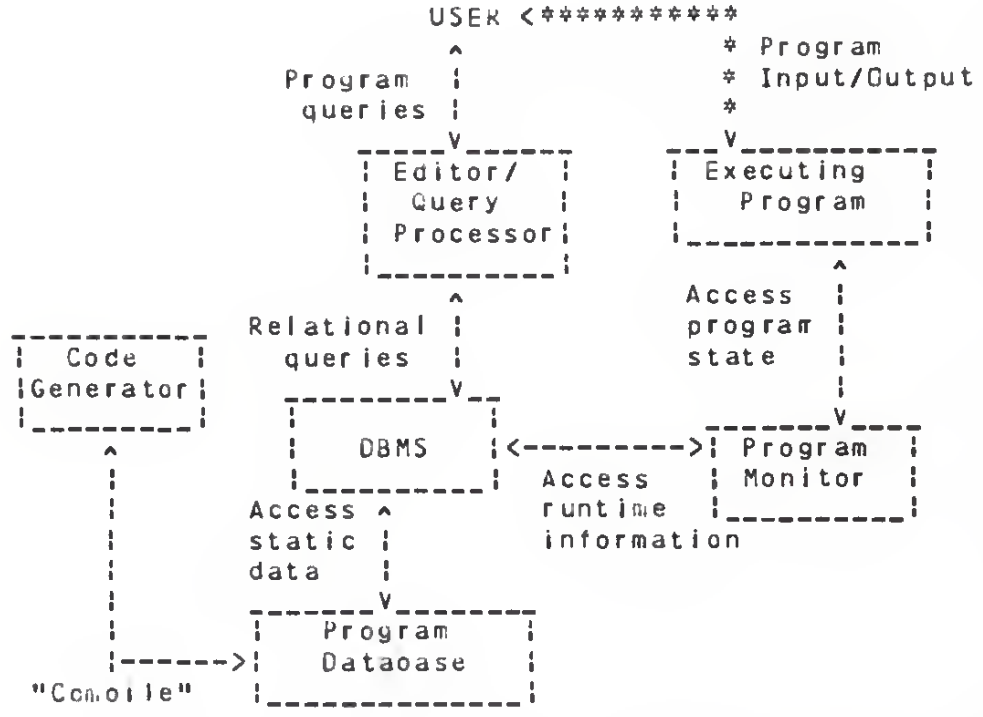

Figure 4.0: Organization of debugging in OMEGA.

In this design, the database system interacts with the program monitor (debugger) to acquire the dynamic state of an executing program. The program monitor provides relations such as valueof (variableID, value) which is transparent to a user but known to the database system as collection of relations that are physically separate from the rest of the database. For instance, to display the value of a variable during a debugging session, a user queries the system to retrieve the attribute "value" from the appropriate tuple in the 'variables' relation.

```
range of v is variables
retrieve v.value where v.name = 'var_name'
```

The DBMS then sends a request to the program monitor for the value portion of the appropriate tuple. The monitor then returns the variable's value from the executing process by looking up the value attribute associated with that variable in the 'valueof' relation in the monitor. That value is returned to the DBMS and displayed in the appropriate format for the type of variable defined in 'variables' relation.

In this environment, a user is allowed to specify a set of actions to be executed whenever

the condition (event) specified by a user is reached. An event is a condition which the user is interested in and is expressed as a relational qualification. A condition is specified in a 'when' clause. For example, to have the program's line number displayed when the procedure 'bugs' is invoked, we might have a query like

```
range of ap is active-procedures
when last(ap).procedure = "bugs" do
     display last(ap).
end.
```

Here we assume that the debugger keeps track of a relation for active procedures which is defined to be ordered in such a way that the first tuple represents the main procedure and the last tuple is for the currently active proceoure in a call chain which behaves like a stack.

With this type of query, some questions that cannot be structured in a conventional debugger can now be handled elegantly. Examples of such questions include: "When will procedure P be called from procedure Q?" and "What are the parameters associated with procedure P?".

Unfortunately, a query is quite wordy and

this approach is not very user-friendly at all. Consequently, Powell and Linton prefer a visual interface so that a user can construct a query from objects in several windows on the screen by using a mouse. In oroer to use multiple windows some graphical tool is necessary; otherwise management of windows is intractable.

4.2: Graphical Supports For Debugging

A visual interface to debugger is very attractive because it aids in creating a user-friendly environment. The user-friendliness arises from the ease of system interaction by using graphical objects on the screen (thereby hiding details) and the selection of graphical objects by using a pointing device such as a mouse. The use of graphics and a pointing device make the process of interaction simple and natural. Graphical objects can be icons, windows, graph, etc. Such graphics is made possible through the ability to allow the brightness of every discernible point in the displayed image to be independently controlled. For an introduction to computer graphics, please refer to [INGALLS 81].

With the graphics technology at our disposal, multiple windows of various sizes on a screen are attainable. Furthermore, the overlapping-window paradigm for a user interface developed by Alan Kay [KAY & GOLDBERG 77], allows information from different sources to be displayed simultaneously while, at the same time, screen space is used economically [TESLER 81].

To manage these multiple windows, windowing systems are built from the concept of multiple tasks running concurrently. Each task represents a different window and so can be switched freely between tasks by switching between windows. Such multiple windows tasking are supported by SMALLTALK [TESLER 81] and XEROX's LOOPS [BOBROW & STEFIK 83]. Consequently, this approach is of great value in debugging because the user's need of information simultaneously gathered from different sources for debugging is now available on a single CRT. This approach is used in instituting a "paperless programming" environment whereby program listings or information is no longer needed during every phase of the development and maintenance of programs.

The idea of "paperless programming" is being applied in the Blit debugger [CARGILL 83] for C programs. The Blit debugger uses a multi-processing bitmap terminal. When Blit (terminal) is connected to a port (RS232) of a UNIX system, a program in Blit's ROM may be used to communicate with the UNIX host and download binary code for execution in Blit. Furthermore, if a small multi-processing kernel, Mpx* is loaded to Blit, Mpx can run in cooperation with corresponding software invoked on the UNIX host. Mpx allows a user to define and manage a set of possibly overlapping windows. Also, Mpx runs a simple process which communicates with a copy of the standard UNIX command interpreter, SHELL. As a result, a user can treat each window as a separate terminal and engage in several simultaneous conversations with UNIX and remote machines connected to the host. Furthermore, the terminal processes may be replaced by arbitrary programs downloaded from the host to exploit Blit's graphics and local processing. Essentially, these processes execute

---

* Designed and implemented by Rob Pike.

asynchronously while Mpx arbitrates the use of the keyboard and mouse, and multiplexes communication with the host.

The debugger is divided between two communicating processes i.e., one in the Blit and the other in the host (UNIX). The process in the Blit interfaces with the keyboard, mouse and display, and has access to the process being debugged and Mpx. The communicating process of the debugger in Blit acts as a front-end, carrying out commands issued by the host process. Since Mpx manages the keyboard and display (CRT), the debugger's I/O will not interfere with that of the process being debugged even when both are active simultaneously. However, logical control of the debugging rests with the communicating process on the host. The communicating process of the debugger on the host, has access to the file system and thus the symbol tables of the program being debugged and has responsibility for almost all decisions calling for semantic interpretation of the program being debugged.

The debugger can be loaded whenever it is required and applied to any desired process for

debugging. Once it is in blit, it can be retained and applied to different processes in turn. This is possible because Mpx can pass the debugger a pointer to the process descriptor of a window picked by a user.

4.3: Knowledge-based Model of Debugging

The research in Artificial Intelligence (AI) is now being applied to other areas. This technology includes the areas of problem solving, knowledge representation, natural language understanding, perception, learning, etc. For further information about this technology, refer to [RICH 83].

Currently, the main AI technique being applied to program debugging is the concept of knowledge-basec systems. Two knowledge-based systems that deal with program bugs are: PROUST (Program Understander for Students) [JOHNSON & SOLOWAY 85] and FALOSY (Fault Localization System) [SEDLMEYER et al 83].

A knowledge-oase is a repository of declarative or procedural cefinitions of knowledge

and is dynamic [SOWA 84]. Declarative knowledge is concerned with the "what" or "knowing that", whereas procedural knowledge is concerned with the "how" or "knowing how". As an example, [SIMON 69] cited the following two specifications for a circle (p. 111).

- A circle is the locus of all points equidistant from a given point.
- To construct a circle, rotate a compass with one arm fixed until the other arm has returned to its starting point.

The first sentence is a declarative knowledge while the second is a procedural one.

Now, what is the difference between a database system [4.1] and a knowledge-based system? John Sowa [SOWA 84, 277] differentiates the two as follows:

- In database systems, "the user must know what to ask for and what to do with the results."
- In knowledge-based systems, the systems "keep track of the meaning of the data and performs inferences to determine what information is needed even when it has not been explicitly requested."

4.3.1: PROUST (Program Understander for Student)

PROUST is a knowledge-based system that attempts to find bugs in PASCAL programs written by novice programmers. When a bug is oetected, PROUST oetermines how the bug can be corrected and suggests why the bug arose. PROUST accomplishes its goal by employing a knowledge base of common programming "plans." These "plans" are selected to tackle a specific programming problem which has been defined into PROUST's problem description.

This knowleoge of a problem's definition makes the variability of novice solutions more manageable and provides important information about the programmer's intentions. To supply PROuST with descriptions of the programming problems, a problem description language is used. Each problem description is a paraphrasing of the English-language problem statement that is handed out to students. The problem description consists of programming goals and sets of data objects. Programming goals are the principal requirement that must be satisfied; sets of data objects are the data that the program must manipulate.

An example of how goals are extracted from a problem statement is demonstrated by the following

[JOHNSON & SOLOWAY 85, p 183]:

Problem statement :
Write a program that reads in a sequence of positive numbers, stopping when 99999 is read. Compute the average of these numbers. Do not include the 99999 in the average. Be sure to reject any input that is not positive.

Extracted goals :
- Read successive values of New, stopping when a sentinel value, 99999 is read.
- Make sure that the condition New <= 0 is never true.
- Compute the average of New.
- Output the average of New.

* Note : Sentinel value is a value which signals the end of input.

From these goals, a problem description is generated for PROUST. In the problem description, each data object to which the goals refer is named and declared. Also each goal, extracted from the problem statement, is recorded in the problem description. With these goals identified, plans must be selected from the knowledge base to implement these goals. PROUST uses a frame-based [MINSKY 75] programming knowledge which consists of goals and plans. Plans are stereotypic methods for implementing goals.

Once a problem description is defined and before any analysis of goals and plans takes

place, a student's PASCAL program is parsed and a parse tree is produced. This parse tree is then operated on during subsequent analysis of the program. When PROUST analyses a program, it selects goals from the problem description one at a time. As a goal is selected, PROUST retrieves from its programming knowledge base, plans that could be used to implement the goal. If the plan matches the program, then PROUST proceeds and selects the next goal; otherwise a different plan is retrieved. If none of the plans matches the program, PROUST must look for bugs that account for the mismatch in one of the plans.

When PROUST encounters plan differences, it does not give up on the plan; instead, it tries to find a way of interpreting the plan differences as bugs. Plan differences are explained by means of bug rules. Each bug rule has a test part which examines the plan differences to see whether the rule is applicable, and an action part, which explains the plan differences.

4.3.2: FALOSY (Fault Localization System)

FALOSY is a very specific fault locating

system limited to master file update. The general master file update programs involve the updating of appropriate master records to reflect the activities represented by the corresponcing transaction records from a set of master files and a set of transaction files respectively. FALOSY ioentifies the statement of master file update program statements which cause anomalous behavior. This identification of statements is accomplished by a knowledge-based model which includes the integration of prototypic and causal reasoning about faults and a recognition-based mechanism for program abstraction. The method of abstraction is done by using a recognition method in which source statements are matched to functional prototypes. Each functional prototype consists of four components namely

a) a set of recognition triggers,
b) a description of intended behavior,
c) a description of expected structure and
d) a list of constraints which must be satisfied to ensure that the candidate structure represented an instance of the prototype.

To oetect a fault, analysis of output discrepancy is needed. Through this analysis, an initial localization tactic is selected. The localization tactic is a method used to focus

attention on suspicious program statements. This analysis procuces a description of the differences between expected and observed outputs. It also generates a set of fault hypotheses, which consist of a functional prototype and an expected defect. Fault hypothesis is tested by determining if the expected defect is present. If a defect is detected through that hypothesis, then the hypothesis and the erroneous program statement(s) is output and the program stops; otherwise current localization tactic is reevaluated to select next available fault hypothesis. If a localization tactic is retained then it is used to generate the next hypothesis; otherwise a new tactic is selecteo. However, if no more plausible hypothesis can be mace then localization terminates with an appropriate message. The logic of this fault localization prccess is depicted in Figure 4.1.

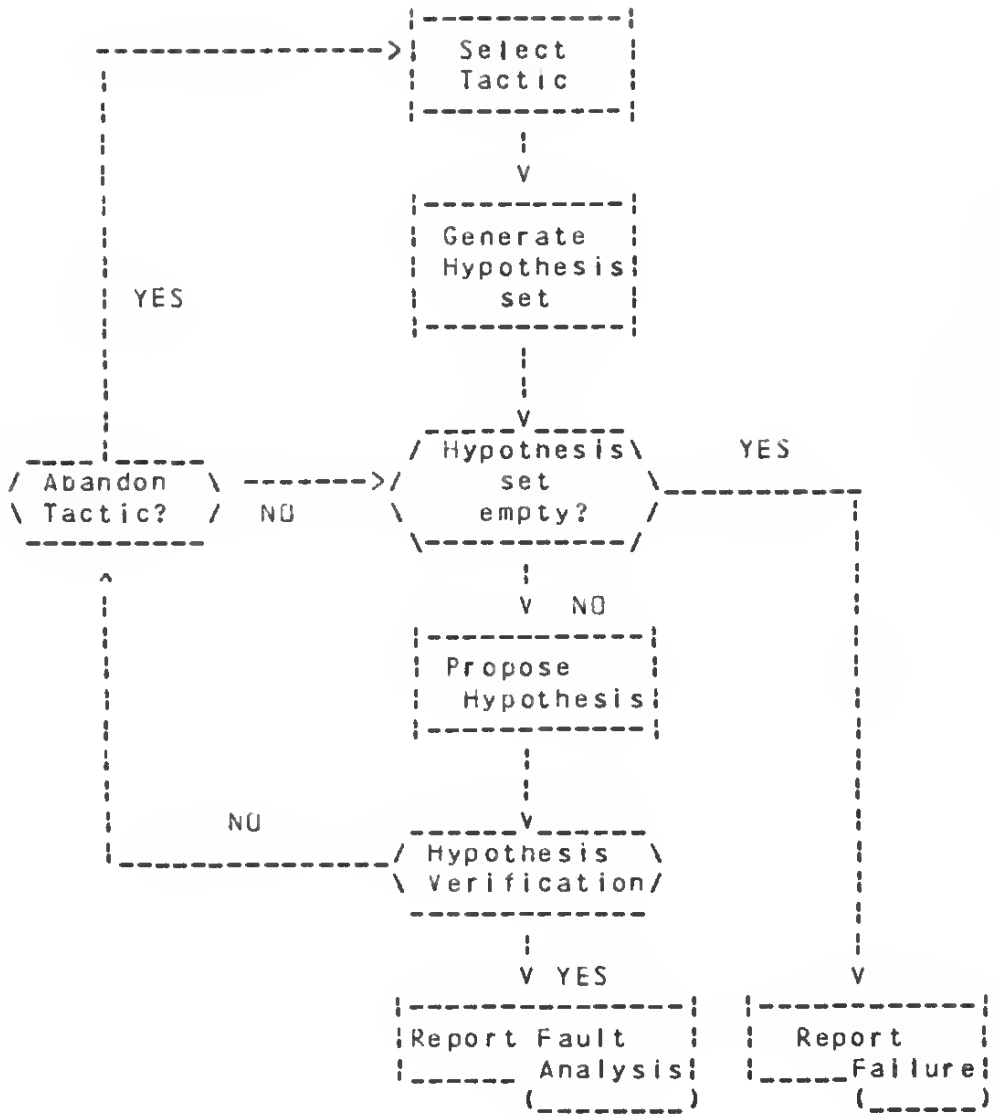

Figure 4.1: The fault localization process.

4.4: Summary and conclusion

With the database model of debugging, it is

possible to construct some powerful queries (complicated ones) that cannot be carried out by a conventional debugger because the information of a program is properly organized for easy retrieval and maintenance. Unfortunately, a query is quite woroy and a user needs to know how the database is organized so that a proper query can be constructed to retrieve pertinent information.

Graphical supports for debugging is an excellent form of interface between user and the debugger. Morecver, "paperless programming" seems promising with the help of computer graphics.

With the ability to represent some knowledge of a program in a debugger, debugging a program can be automated and suggestions can automatically be generated concerning possible remedies to correct a program's behavior. But current "automated" debuggers are tailored for only specific programs and not generic ones.

Consequently, a database model is good for constructing powerful queries and could be of great help in debugging a concurrent program

because multiple processes are involved. These multiple prccesses are often instances of a specific process type. Thus, each instance can be represented as a record in a database file for tnat specific process type with process identification as key to each record. Graphical support is definitely an asset in debugging a concurrent program because the source text of each process can be viewed simultaneously when multiple windows are possible. Furthermore, knowledge-based mooels of debugging will be of interest in debugging concurrent programs so that concurrent semantics checking can be automated to forewarn user of potential problems.

Chapter 5
Feature and implementation issues of a debugger for Concurrent C

5.1: Introduction

The differences between a concurrent program and a sequential one are: i) a concurrert program is composed of multiple sequential programs running concurrently, and ii) these multiple sequential programs typically communicate between themselves via messages (like Concurrent C) or shared memory (as with Concurrent Pascal). Consequently, errors encountered in sequential programs are also possible in concurrent programs. However, some other errors that CANNOT happen in sequential programs can occur in concurrent programs.

5.2: Categories of errors

Errors can be placed into two main categories, namely, concurrent errors and sequential errors.

Concurrent errors are errors that can occur ONLY in concurrent programs and NOT in sequential

programs. Concurrent errors can be subdivided into two groups which are race conditions and deadlocks. A race condition error is concerned with an error which arises due to a timing problem (refer to Chapter 1 for more details about timing problems). A deadlock error is concerned with the non-progress of an executing concurrent program. Non-progress means a subset or all of the multiple sequential programs within a concurrent program cannot make any progress. Essentially, these sequential programs are waiting for some events to occur before they can proceed.

Sequential errors can occur in both sequential and concurrent programs. These errors come in many forms but the common ones which have been encountered by the author are: initialization error, mismatched error, pointer error, range error, arithmetic error, and output error.

An initialization error arises due to a missing or wrong initialization of some program variables. This type of error could cause, for example, infinite looping and subscript error.

A mismatched error is concerned with an error which arises because a condition NEVER becomes true. Such error could result in the failure to exit a loop (infinite looping), or an unintended execution path to be followed.

Pointer errors arise due to incorrect addressing. A pointer error could cause, for instance, system interrupts (i.e., a type of exception handling) or code to be overwritten.

A range error is mainly concerned with some subscript being out of bound or an invalid value for a variable. Its effect is a system interrupt or corrupted data retrieval during indexing.

An arithmetic error is concerned with arithmetic overflow/underflow and division by zero. Its effect includes precision problems and a system interrupt.

Output errors arise due to a computation, logic or formatting error. Such errors could cause the program to be rewritten, personnel to point fingers at one another, etc.

5.3: Inadequacy of sequential debuggers in dealing with concurrent programs

With the identification of possible errors in concurrent and sequential programs, it is obvious that the mechanism employed in a debugger for sequential programs is NOT capable of debugging concurrent programs. This is because a sequential debugger lacks the ability to handle multiple processes (sequential programs) at the same instance. For example, dbx (a debugger for C) cannot be used to debug a Concurrent C program. However, the features exhibited in sequential debuggers can be employed to debug parts of a concurrent program. This is possible because a concurrent program is composed of multiple sequential programs. Nevertheless, additional features which are essential for debugging concurrent programs must also be incorporated.

5.4: Basic features for a Concurrent C debugger

Since a concurrent program is composed of multiple sequential programs called processes (in Concurrent C and Concurrent Pascal), the ability to visualize the state of each and every process at any one instance is essential in debugging a

program. This capability is essential in order to pinpoint which process(es) is causing the program to behave anomalously by examining the state of each process. Furthermore, these processes can communicate among themselves which makes debugging more difficult and often very tedious. The availability for examination of each process' state at any instance will facilitate identification of the culprit process by a relative comparison of the state of each process. This ability is very useful in assisting in the localization of deadlock or detection of an unwanted race condition, both of which are frequently ercountered in concurrent programming.

5.4.1: "Overall-view" approach of processes during debugging

With the state of each and every process visible at any one instance, we can deduce the cause of a deadlock or unwanted race condition. The comparison of these states will help to reveal the events causing the deadlock or unwanted race condition.

As an example, assume that we have four processes which simulate a token ring network

environment with each process representing a node of the network (see Figure 5.0a). The state of processes A, C, and D reveal that they are waiting on the token and there is a token currently on the network ring since process B is in an active state.

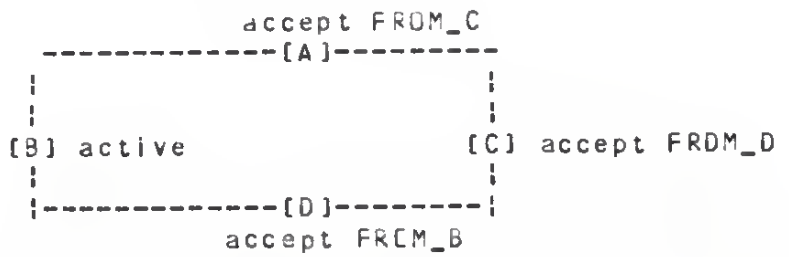

Figure 5.0a: A deadlock with all process states being visible

With the state of each and every process being visible at this time, we can infer that process B is the process responsible for the deadlock. This is because processes A, C and D are waiting, and the ONLY active process at this time is B. Thus, B is the process which never releases the token! Consequently, the other processes cannot proceed and the result is deadlock.

Suppose, for a moment, that only two processes' states are visible instead of all four.

If processes C and D are the processes with visible states (see Figure 5.0b), then we have no idea which of the other two processes (i.e., A and B) is the culprit.

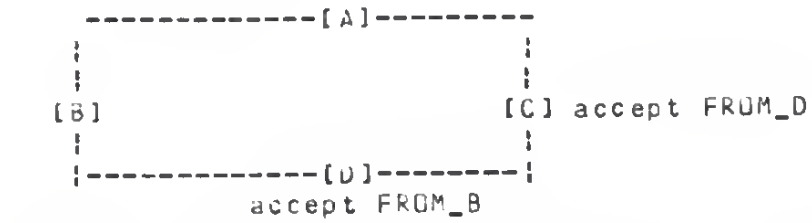

Figure 5.0b: A deadlock with partial process states being visible

5.4.2: Process states

To support the state of each and every process being visible at any one instance, we need to identify the necessary states which will highlight the status of a process. Unfortunately, the state of a process is oependent upon the context in which it is used.

5.4.2.1: Operating system's viewpoint of a state of a process

From the operating system's viewpoint, a process can be in only one of the following states, namely: ready, running, blocked or halted [DEITEL 84]. A process is in the ready

state if it is waiting for a CPU. A process is in a running state if it currently possesses a CPU. When a process is said to be in the blocked state, it is waiting for some event to happen (such as an I/O completion) before it can resume. Lastly, a process is in the halted state if it has run to completion. These four states are needed just to facilitate the scheduling of a process by the operating system.

5.4.2.2: Concurrent C's view of a process' state

On the other hand, the Concurrent C system is aware of ONLY three states for any process. These states are: active, completed, and terminated [GEHANI & ROOME 84]. An active state is attained whenever a process is created and the process remains in this state while executing the statements specified in the corresponding process body. A completed state is reached whenever a process executes a 'return' statement in its process body or when it reaches the end of its body. Finally, a terminated state is a state in which a process has completed and all the processes created by it (i.e., its child processes) have terminated, or when a process

executes a ready-to-terminate alternative in a 'select' statement within a process body. These states are employed in Concurrent C system so that transaction calls to active or terminated processes can be detected. With this detection, invalid transaction calls, deadlock, etc., can be handled gracefully.

5.4.2.3: The selected process states of a process

The states needed to support the capability of each process' state being visible to a programmer in Concurrent C at any one instance are quite similar to those mentioned earlier. Those previously mentioned states have been refined and can be placed into four categories viz.,

a) active

b) communicating (renoezvous)

c) delayed and

d) terminated

An active state is a state in which the process is executing some instructions but which may be interrupted for swapping. When a process has started up (by the 'create' primitive in

Concurrent C), it is in an active state. Once in the active state, a process could go to any of the other states depending on its source code. When it is communicating with another process (i.e., by executing a transaction call or an 'accept' primitive), its state is said to be 'communicating.' When a process executes an 'accept' primitive, it is delayed until there is a corresponding transaction call targeted to this 'accept' - a mechanism for rendezvous in Concurrent C [GEHANI & ROOME 84]. Similarly, when a process executes a transaction call (which is not a timed transaction), it is delayed until the transaction call is complete (i.e., until a renoezvous with corresponding 'accept' has taken place and a return from the rendezvous has occurred). In a timed transaction call, the process calling the transaction is allowed to withdraw the transaction call if the targeted process does not 'accept' the transaction within the specified period [GEHANI & ROOME 84]. A process is in a delayec state only when it expiicitly executes a 'delay' primitive in Concurrent C. A terminateo state is in effect whenever a process executes a 'terminate' primitive or reaches the end of its source code.

5.4.2.4: Importance of the selected process states.

The reasons why the previous four states are important in facilitating the debugging of a Concurrent C program are now described. An active state is needed so that we know for sure that a process has started and is running. Communicating and delayed states are needed just to give the programmer involved in debugging a sense of where the process is executing at a particular time - it is a kind of a sign post indicating the relative line of code being executed within a process. These two states are chosen because they are used very often in Concurrent C as means of synchronizing and communicating with other processes. Lastly, we need the terminated state just to notify the programmer that a particular process is no longer of interest to him/her because it has alreaoy terminated. With these four states, we are able to hide details which are unimportant to a programmer at the stage of locating the malicious process(es). This is an approach to state abstraction.

Since we do not know which of the processes'

states are necessary in order to facilitate in debugging, every process' state is displayed at all times. Furthermore, any one of the processes within a concurrent program has a potential to be the culprit which causes ar error. Thus, some of the processes' states may be redundant, but certainly the amount of information presented is adequate to infer the immediate cause of a cascading error. A cascading error refers to an error which propagates through one or more processes. The last process in this chain of errors is the one which could not cope with the error and so was designated as the immediate cause of the cascading error.

This approach will help pinpoint the malicious process but it does not pinpoint the specific line of code within a process that inherited the cascaded error(s).

To really nail down the error to a particular line of code, an examination of the value of variables involved in the suspected process needs to be carried out. The need to examine the values of variables is because a process is partly composed of variables and the value of each

variable determines the behavior of a process. In this fashion, we are employing two levels of abstraction - a zooming-in approach.

5.4.3: Suspension of a group of processes

Besides the four states, we still need another mechanism to help in debugging a Concurrent C program. This mechanism is concerned with how to temporarily suspend the execution of a concurrent program. The purpose of this suspension is to permit checking the value of some variables or even the state of each process in an attempt to nail down a troublesome statement.

5.4.3.1: Ways to trigger suspension of a group of processes

Fortunately, there are a several ways to trigger suspension of a group of processes or a concurrent program.

One method to trigger suspension of a group of processes is to hardcode a "breakpoint" instruction in each process body. This way is static and NOT flexible at all. Furthermore, the source code needs to be recompiled every time

there is a change in the position of any breakpoint in the code.

A better approach is by broadcasting messages to all of the existing processes involved. The process executing the breakpoint instruction has to broadcast a message, to all existing processes (within the Concurrent C program), to suspend themselves. In order to broadcast the message, there must be a global table which indicates which processes are available so that the broadcasting process knows where to send the messages. This capability of knowing where to access the global table and how to send the messages must be included in each process. This approach is a decentralized approach as opposed to a centralized approach which will be presented later.

The time delay for each process to receive and act on the message may be different. The difference in time delay is due to the configuration of the processes and the system's load. Fortunately, this approach is dynamic and so more flexible. It is dynamic because the message to be sent is determined by the global table which is updated while a concurrent program

is being executed. Moreover, it is flexible because the routing of the broadcasting messages need not be hardcoded as the routing is driven by the content of the global table. The only drawback with this approach is that its implementation will be quite hard. This is because each process within a concurrent program needs to have a way to receive the broadcasted message. Apparently this approach is not easily accomplished with the 'accept' primitive of Concurrent C as a spontaneous receipt of a message is impossible. However, there is a way to overcome that problem. The solution is that whenever a process wants to send a message related to a debugger command to other processes, that message is sent to a "mail box" process which acts on behalf of the receiving processes. In order for this solution to work correctly, a process must periodically check in with the "mail-box" process.

Yet another way to trigger the suspension of a group of processes is by using a bottleneck approach. In this case, there is a centralized process which provides a variety of services to the other processes. This centralized process

will accept a request from all the other processes which intend to execute a "breakpoint" instruction or other debugging commands handled by the centralized process. Once a "breakpoint" instruction is executed, all other requests to the centralized process are queued. Consequently, all other processes will be suspended or delayed. This method is quite flexible and easy to implement. For example, in the kernel approach to structuring an operating system, a number of operating system processes are created to serve users' processes so that the utilization of a system's resources can be improved [DEITEL 83]. Thus the centralized process can be created as one of the system processes to serve the concurrent processes of a concurrent program.

5.4.3.2: Importance of suspending all processes during a breakpoint

Since a concurrent program is composed of multiple processes, a temporary suspension of a process MUST also stop all of the other processes from executing lest suspension violate some timing factors involved in a concurrent program. Such a timing violation is sure to occur if the processes are running on a multi-processor environment, but

the violation is not so obvious in a 'quasi-parallel' environment (multiplexing of single processor).

For instance, if we have a concurrent program whose source code (pseudo Concurrent C, see Figure 5.1a) is running in a two-processor environment, then a deadlock* is SURE to happen because processes 2 and 3 in processor B are NOT progressing due to the suspension of prccess 1 in processor A.

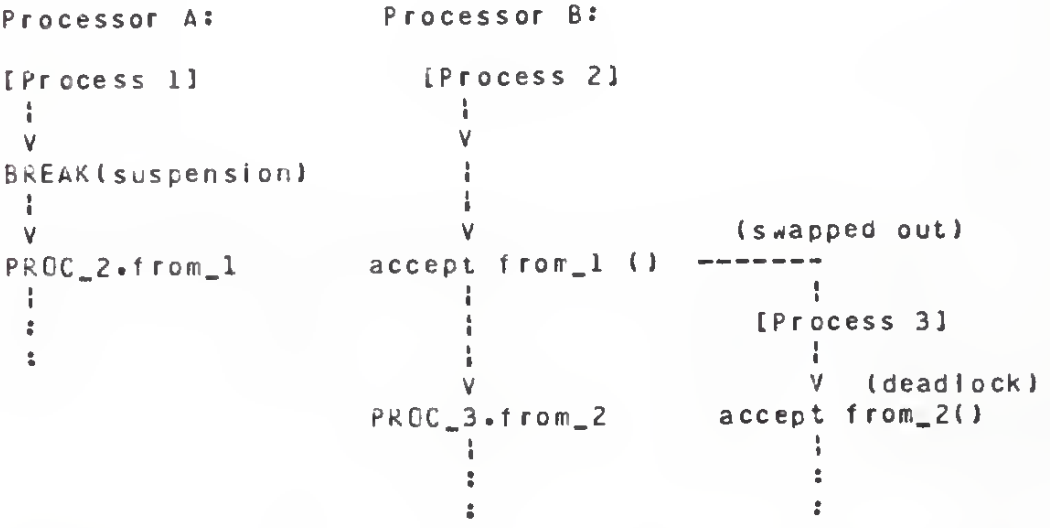

Figure 5.1a: Deadlock in a multiple processors system

---

* It is not a temporary blocking of all processes because if the breakpoint is not resumed in some finite time then a deadlock will be detected by the Concurrent C system.

Similarly, in a single-processor environment (see Figure 5.1b) the deadlock will also arise but it will take a longer time. This is so because these three processes are utilizing the same processor. Thus the deadlock is NOT obvious until sometime later. In this example, the deadlock will not occur if the breakpointing (suspension) of process 1 will also cause the rest of the processes (2 and 3) to stop through the bottleneck approach for providing suspension. However, the situation will get worse if processes can be prioritized because it complicates the analysis of swapping.

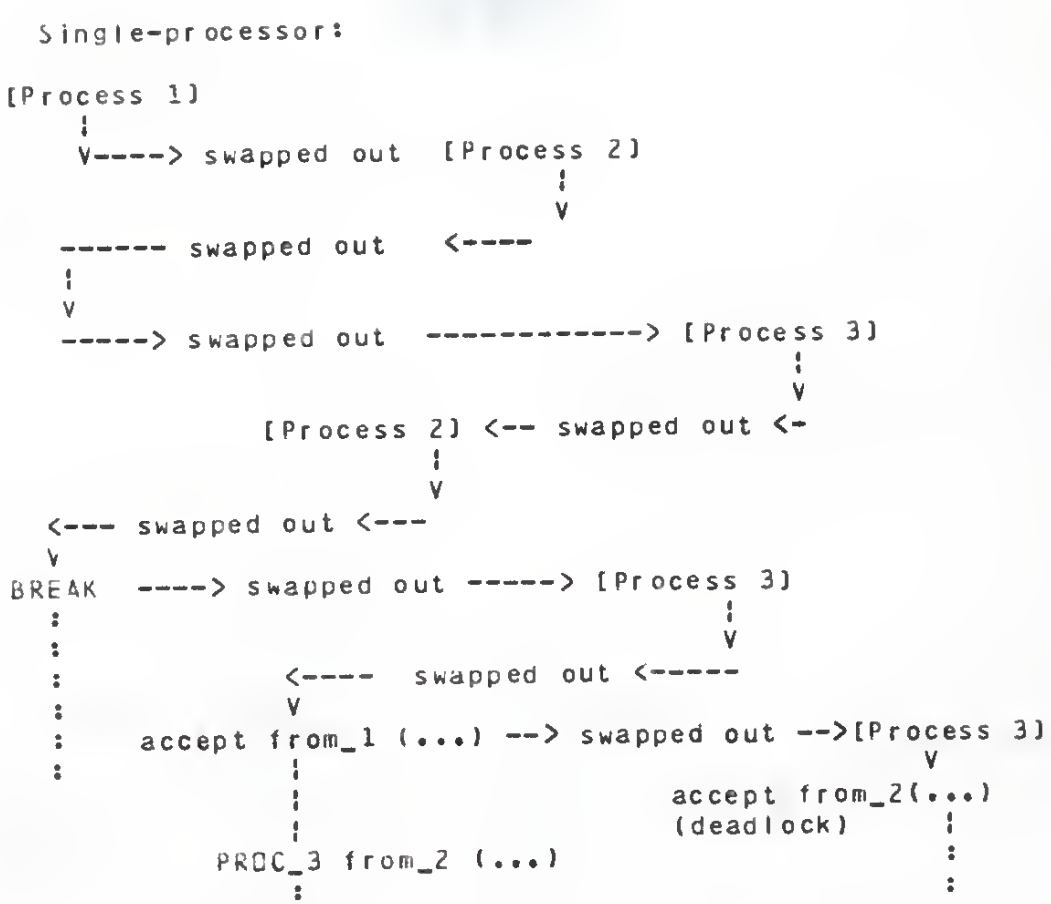

Figure 5.1b: Deadlock in a single processor system

Though these three basic features (namely, (i) overall view approach of processes during debugging, (ii) the selected process states and (iii) suspension of all processes during a breakpoint by a process, are quite simple) they are language dependent. The next section describes the implementation of these features in a trace facility or rudimentary debugging package

for Concurrent C.

5.5: Implementation of basic debugging features

To be able to visualize the overall states of a concurrent program in terms of the individual component processes, a kind of icon is employed to represent each process and its state. This implementation decision was made in order to allow all processes to be presented on the CRT screen simultaneously. Since Concurrent C is an enhancement of C, it can invoke any of the C library routines and packages [GEHANI & ROOME 84]. To implement the process "icons", a screen package [ARNOLD 83] available on UNIX system is employed. Each process can be represented by a rectangular box and each box possesses three pieces of information: process name, process type and status (state).

A process name is composed of a number which represents the position of a process in the sequence of processes that have been created in a program and the process variable name associated with that process. For instance, '1: b_mgr' in Figure 5.2 means this process is the second

process created within this concurrent program and the process variable name associated with this process is b_mgr.

A process type is one of the process types defined within a concurrent program. In this case, the word "manager" on the second line in Figure 5.2 represents the type of process, "manager", declared for this particular process b_mgr.

The status component represents a process state and any relevant information related to that state. For example, the last line of Figure 5.2 means this process is about to execute a 'select' clause. In this case, the word 'before' is a relevant piece of information related to the state S ('Select'). Another example of relevant information related to a state such as 'communicating' will be the name of a transaction call and whether the state is before or after the transaction call.

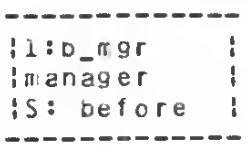

Figure 5.2: A representation of a process by an "icon"

5.5.1: Layout of a CRT screen in representing a Concurrent C program

Unfortunately, due to the size of a terminal screen (24 x 80 characters), the maximum number of boxes that can be displayed at any one time is ONLY eighteen. However, whenever a process has terminated, its box is salvaged for later use by another process if there is one.

As a common practice, a programmer should always start off with a small number of processes and increase the number once it is certain that they work in a small number. By the time one gets to a large number of processes, normally further debugging is not needed as errors have been discovered while executing a small number of processes. This argument is only valid if the small number of processes are independent of each other (i.e., there is no interaction between any of them) and the environment where these processes

are running has no limitation on the number of existing processes at any one time. However, many versions of UNIX enforce a limit on the number of processes that one user can run at a time; typically this limit is set to about 20 processes [GEHANI & ROOME 84]. Thus, the limitation of eighteen boxes should not hinder any concurrent programmer. Furthermore, if a programmer needs more than eighteen boxes, the rest of the processes' information are automatically displayed in the message area on the same screen i.e., line sixteen to twenty-one of a screen instead of an 'icon' and these six lines are scrollable.

5.5.2: Comparison of Concurrent C Window Manager with the implemented debugging features

In contrast with the Concurrent C Window Manager [THOMAS 84], which allows only up to eight processes' states to be displayed at any one instance, this implementation allows up to eighteen. With the Window Manager, output from the process(es) is displayed on the four rectangular boxes assigned to four selected processes. On the other hand, the author's implementation allows all output from the

process(es) to be displayed on the bottom section of a CRT screen. However, there are eight process states employed by the Concurrent C Window Manager whereas there are only four main process states and many substates supported in this implementation (refer to Table 1).

Abbreviation	Meaning
P:	Process is created and running.
S: before	Before 'select' statement.
S: after	After 'select' statement (impossible if 'terminate' is an alternative).
D: before	Before delay statement (impossible if 'delay' is an alternative).
D: after	After delay statement.
A:xxxxxxx	Immediately after an 'accept' statement with the first seven characters of a transaction name, xxxxxxx.
R:	Before a 'treturn' statement in an 'accept'.
TP:	Before a transaction pointer statement is used for a call.
bC:xxxxxx	Before a transaction call with the first six characters of a transaction name, xxxxxx.
aC:xxxxxx	After a transaction call with the first six characters of a transaction name, xxxxxx.
tT:	Before a timed transaction call.

Table 1: Substates

Unfortunately, the Concurrent C Window

Manager ooes not allow any stepping except by using the Control-s keystrokes for the inaccurate and temporary suspension of a concurrent program. This implementation, however, does support stepping but only at output statements to a CRT screen and at every change of any process state. It automatically breakpoints at every output line and at any change in any of the processes' state. Moreover, a user has the ability to skip as many breakpoints as he/she likes by specifying a number as prompted for after each breakpoint is executed. The prompt is displayed on line twenty-three of a screen. Refer to the following table (Table 2) for other comparisons between Concurrent C Window Manager and the author's implementation.

Concurrent C Window Manager:	The author's implemertation:
1. Uses screen-package.	1. Uses screen-package.
2. Invoked by manual insertion.	2. Invoked through a preprocessor.
3. Eignt processes are visible.	3. Eighteen processes are visible.
4. Process' output to a CRT screen is displayed in one of the four rectangular boxes.	4. Process' output to a CRT screen is displayed at the lower portion of a CRT screen.
5. Free up the process box when a process is terminated.	5. Free up the Icon when a process is terminated.
6. Freedom of switching process' output to a CRT screen be cisplayed on the four rectangular boxes.	6. All output to a CRT screen are displayed on the lower portion of a CRT screen.
7. Eight distinct states.	7. Four distinct states and many substates.
8. Break by Control-s.	8. Break by Control-s or at each screen output statement and at every change of any process state.
9. No "single" stepping.	9. Single stepping at screen output statement.
10. Capable of refreshing a CRT.	10. Unable to refresh a CRT.
11. Can be aborted by a Control-C.	11. Can be aborted by a Control-C.

Table 2: Comparison of Corcurrent C Window Manager and the author's implementation

5.5.3: Implementation of states transition

In order for the tracing facility to present the state of each process to a programmer, the tracing facility has to know the state of each process at any instance. This can be accomplished by updating the respective box of each process whenever a process changes its state. As mentioned earlier in section 5.4.2.3, we are interested ONLY in states like active, communicate, delayed, and terminated. To enable a tracing facility to reflect these states, the source code of a concurrent program has to be modified to include some updating statements at the appropriate places where a state transition takes place. Such places include the beginning of a process, communication statements (viz., transaction call and 'accept' statements), delay statements and terminate statements, respectively.

The first three states can be done easily, but the last state, "terminated", is very hard. A process in Concurrent C can terminate whenever it reaches the end of a process body or executes a "terminate" statement within a 'select' clause.

The hardest part, or an impossible one, is when it executes a "terminate" statement. This is due to the implementation of Concurrent C in which the "terminate" statement is required to be all by itself textually, i.e., it CANNOT be preceded by any other statement except "or", and consequently whatever statements follow it are ignored. Thus, there is no way to insert any update statement to reflect whether a "terminate" statement will be or has been executed. If this approach is desired, the only way out is to modify the Concurrent C compiler. Nevertheless, the terminated state can be represented by wiping out the icon representing a process which has terminated. To detect whether a process has terminated, a Concurrent C built-in routine called c_active can be invoked [GEHANI & ROOME 84].

Last but not least, the implementation of suspending processes whenever a process is breakpointed can be achieved by queueing up all processes' requests to update their respective box in a process. This is essentially a bottleneck approach. This is quite easily accomplished, but is not the most efficient approach. For example, the scenario depicted in Figure 5.1b can never

result in a deadlock. This is because the breakpoint request will have to request an access to the bottleneck process before the breakpoint is granted and the rest of the processes must also request an access to the bottleneck process before an update of a process state can take place. Thus all incoming requests will be queued and so all the involved processes will be suspended.

For this implementation, a frontend debugging process is automatically inserted into each concurrent program. This debugger process is the bottleneck process we have been talking about. It handles the update request and any other debugging request submitted by a programmer. This approach will still preserve the semantics of a concurrent program even with the debugging process included. This is because every process has to go through that debugging process to perform state update, screen IO and breakpoint. So every process will be affected in one way or another about the same amount. Obviously, the overall behavior of the concurrent program will appear slower and it will consume more cpu time when the bottleneck process is introduced.

5.5.4: User Interface

The tracing facility is implemented as a preprocessor which parses the source text of a Concurrent C program. During the parsing, it writes to an output file the parsed source text. Furthermore, whenever a state transition statement (such as transaction call, "accept", "delay", "select", etc.) is parsed, some screen update statements are appended to the output file. Consequently, a user's source program is not modified but a copy of the original program with added screen update statements is generated. The detailed instructions of how to use the tracing facility are illustrated in the Appendix - User's Manual.

5.6: The role of a compiler in building a debugger

As we know, the purpose of a compiler is to translate a programming language (source language) to executable codes (the object or target language) [AHO & ULLMAN 79]. With the existence of a compiler, we do not have to worry about the idiosyncrasies of register usage and assembly instructions; let alone about customizing

assembly language interface routines. Some interface routines are needed to allow a user's program(s) to interact with the operating system for utilizing services not supported by the high-level language in which a user writes program(s). However, if the source code of a compiler is available, then we can ennance the compiler by implementing some features which are important for a debugger [YOUNG 81]. Features like symbolic debugging use an elaborate symbol table whereas source level deougging uses line number and code mapping. Since a compiler can parse a program in sequence of lines and generate a symbol table, we need only to add some more code to the compiler to accomplish the two features (symbolic oeougging and source level debugging) just mentioned. Furthermore, we can understand how the program is organized at memory level oy studying the executable codes which are generated after a successful compilation. By analyzing the executable cudes, we will be able to know how much space (memory) is allocated for each variable, for example. Moreover, we will be able to know how tne variables are kept (addressing), and the convention on using the registers, etc. Only with such detailed

knowledge of the complier, can a modification be carried out to include some debugging features.

Unfortunately, the source code for Concurrent C's compiler is NOT available to Kansas State University at this time. Thus, it is very hard to incorporate a debugger to Concurrent C, unless one desires to re-invent the wheel for Concurrent C by writing a new compiler. However, one can still look at how the compiler of Concurrent C generates assembly code to configure a concurrent program by examining the assembly output of the compiled program (with a -S option). Once we know how they are configured, we still CANNOT insert any code into the compiler to generate the necessary additional code required to facilitate the implementation of a debugger. But there is a solution to this, i.e., we could parse the assembly output and insert into that; unfortunately the chances of making mistakes are very great and cost too much time.

Consequently, the intention of building a debugger for Concurrent C has been discouraged. However, to get a feel of how hard it is to build a debugger, a tracer has been implemented instead

which incorporates the basic features described in section 5.4. The differences between a tracer and a debugger will be discussed in the next section.

5.7: Tracer versus debugger

A tracer is essentially a highlighter which reports the necessary information a programmer needs in order to determine the behavior of a program undergoing execution. But a debugger is a powerful tool that allows a programmer to probe the program he/she is running. It has more capabilities than a tracer. It allows a programmer to manipulate the state of a program, set breakpoints to skip irrelevant information, etc. Essentially, a debugger is a friendlier tool to use than a tracer because a debugger presents ONLY information requested by a programmer. Since a tracer can assist a programmer in nailing down a malignant process, it can be used as a first step in the act of debugging a program.

Chapter 6
Future research and conclusion

After an introduction to concurrent programming and a presentation of approaches to debugging Concurrent C programs, it is now time to comment on the applications of the implemented tracing facility and to comment on future research in Concurrent C debugging.

6.1: Application of the tracing facility

Since there is no debugger for Concurrent C and the Concurrent C window Manager is hard to use, the implemented tracing facility is intended to assist a novice or experienced Concurrent C programmer to overcome the frustration of debugging a Concurrent C program. With the "overall-view" approach of presenting each and every process state of a Concurrent C program on a CRT, this approach will greatly help a programmer in identifying the two notorious errors in concurrent programming viz., deadlock and unwanted race condition. Furthermore, with the ability to breakpoint at every output statement and at every change of a process' state within a Concurrent C program, a user can trace through

his/her Concurrent C program at his/her own pace. Moreover, a user has the ability to skip as many breakpoints as he/she likes by specifying a number in response to a prompt after each breakpoint is executed. The total number of breakpoints is displayed whenever a breakpoint is executed. Thus a user can save some time and some keystrokes to arrive at a state where he/she suspects a Concurrent C program is going to misbehave or crash.

6.2: Future Research in Concurrent C debugging

Since the source code of Concurrent C compiler is not available, a debugger for Concurrent C has not been implemented. Instead, a tracing facility for Concurrent C has been implemented. In the future, if the source code is available then the tracing facility can be extended to provide all of the capabilities required by a debugger. Inasmuch as the tracing facility is incorporated into a Concurrent C program by passing that program through a parser to insert necessary screen package routines and some tracing processes, some portions of this parser's code can be added to the compiler so that

insertions can be done while compiling a Concurrent C program. Thus, the work done on the tracing facility can be reused.

Besides implementing the presented approaches, one might want to pursue the idea of how a database model of debugging would be beneficial to Concurrent C programs. At this moment, a database model of debugging seems very promising because instantiation of process types can be handled elegantly with relational database model. This way of handling is due to the fact that each process type is just a distinct database file in which each record represents an instance of the process type and each record is keyed by the process identification, which has been defined in Concurrent C. The fields in each record of the process type can be the variables local to the process, actual parameters involved, etc. The only other problem is how to set up the relationship to represent the possible interaction between different processes. One possible way, perhaps, is to setup relationships based on the access rights of each process type with regard to other processes. The rest of the relationships would be just the same as those presented in

Chapter 4 for DMEGA system except for a few relationsnips caused by concurrency constructs like "accept", "terminate", "select", "oelay", etc. Thus, it will be interesting to see how these relationships work together to represent the semantics of a Concurrent C program and how easily the program can be oebugged.

With more advanced technology for graphical tools available in the near future, graphical supports for debugging will be widely employed. This is because the cost of such supports will be greatly reduced and will stimulate more research in that avenue. If such breakthrough is achieved then "paperless programming" and graphical interface will be highly appreciated by users. Research in user interfaces using graphics may be of great interest oecause it is hard to create an interface that can easily be used without much confusion and much learning and at the same time self-explanatory.

Another future area of research is concerned with the possibility of a knowledge-based model of debugging for Concurrent C programs. As stated in

Chapter 4, the hardest part is to represent the knowledge of the interaction between the processes in a Concurrent C program. Once that is accomplished then debugging a Concurrent C program will be very easy as localization of error can be automated. To accomplish this automated task, the debugger must have a knowledge of the goal of the concurrent program. The tough part is determining how to represent the goal to the debugger. As a stepping stone, one can allow the debugger to check whether a process is performing the correct task that was intended by a programmer. This can be accomplished by supplying the specifications of a process and letting the system verify that the process behaves in accordance with its specification. Any discrepancy encountered during this verification will be reported. The specifications should include with which process(es) a process will communicate and what it will communicate. This information is needed just to verify the access right between processes. Once these processes are verified, the debugger should then verify that the goal of this concurrent program is met i.e., the net effect due to its processes interaction.

6.3: Conclusion

Constructing a debugger for concurrent programs is more difficult than for sequential programs. The difficulties arise because concurrent programs are composed of multiple sequential programs and these sequential programs can interact with each other. As a result, additional mechanisms (other than the ones found in sequential program debuggers) are needed to construct a debugger for concurrent programs.

One of the additional mechanisms is the "overall-view" capability. With this capability, the state of each and every process is visible at any instance. This mechanism is needed to solve the two most common errors which occur in concurrent programming viz., deadlock and unwanted race condition. With this mechanism, a user can identify the two notorious problems quickly. Another required mechanism is the capability to suspend all existing processes in a concurrent program if one of them executes a breakpoint. This suspension mechanism is the most crucial one in concurrent programming because a timing violation by other processes is possible if

suspension of other processes is not enforced.

With these two features supplementing the mechanisms from a sequential debugger, a powerful concurrent debugger can be constructed.

BIBLIOGRAPHY:

[AHO & ULLMAN 79] Alfred V. Aho and Jeffrey D. Ullman, Principles of Compiler Design, Addison-Wesley, 1979.

[ANDREWS & SCHNEIDER 83] Gregory R. Andrews and Fred B. Schneider, Concepts and Notations for Concurrent Programming, Computing Surveys, Vol. 15, No. 1, March 1983, pp 3-43.

[ARNOLD 83] Kenneth C. R. C. Arnold, Screen Updating and Cursor Movement Optimization: A Library Package, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, Berkeley, California, 1983.

[BAIARDI et al 86] Fabrizio Baiardi et al, Development of a debugger for a Concurrent Language, IEEE Transactions on Software Engineering, Vol. SE-12, No. 4, April 1986, pp 547-553.

[BOBROW & HAYES 85] Daniel G. Bobrow and Patrick J. Hayes, Artificial Intelligence - where are we? Artificial Intelligence Vol. 25, 1985, pp 375-415.

[BOBROW & STEFIK 83] Daniel G. Bobrow and Mark Stefik, The LOOPS Manual, Xerox PARC, December, 1983.

[BROWN & SAMPSON 73] A. R. Brown and W. A. Sampson, Program Debugging: the prevention and cure of program errors, MacDonald and American Elsevier, 1973.

[BROWN & SEDGEWICK 85] Marc H. Brown and Robert Sedgewick, Techniques for Algorithm Animation, IEEE Software, Vol. 2, No. 1, January 1985, pp 28-39.

[CARGILL 83] Thomas A. Cargill, The BLIT Debugger, The Journal of Systems and Software, Vol. 3, No. 4, December 1983, pp 277-284.

[DEITEL 83] Harvey M. Deitel An introduction to Operating Systems, Addison-Wesley, 1984.

[DIONNE & MACKWORTH 78] Mark S. Dionne and Alan K. Mackworth, ANTICS: A system for animating LISP programs, Computer graphics and image processing, Vol. 7, 1978, pp 105-119.

[GARCIA-MOLINA et al 85] Hector Garcia-Molina et al, Debugging a distributed Computing System, IEEE Transactions on Software Engineering, Vol. SE-10, No. 2, March 1984, pp 210-219.

[GARMAN 81] John R. Garman, The "BUG" heard 'round the world, ACM SIGSOFT, Software Engineering Notes, Vol. 6, No. 5, October 1981, pp 3-10

[GEHANI & ROOME 84a] N. H. Gehani and W. D. Roome, Concurrent C, AT&T Bell Laboratories, 1984.

[GEHANI & ROOME 84b] N. H. Gehani and W. D. Roome, Concurrent C - Programming Examples, AT&T Bell Laboratories, 1984.

[GENTLEMAN and HOEKSMA 83] W. Morven Gentleman and Henry Hoeksma, Hardware Assisted High-level Debugging, ACM SIGSOFT/SIGPLAN Notices: Symposium on Debugging, Vol. 3 (March, 1983), pp 140-144.

[GHANI and FARRELL 80] Noordin Ghani and Edward Farrell, Microprocessor System Debugging, Research Studies Press, 1980.

[HAMLET] Rich G. Hamlet, Debugging "Level": Step-Wise Debugging, ACM SIGSOFT/SIGPLAN Notices: Symposium on Debugging, Vol. 3 (March, 1983), pp 4-8.

[HANSEN 73] Brinch Hansen, Testing a multiprogramming system, Software-Practice and experience, Vol. 3, 1973, pp 145-150.

[HECHT 77] Matthew S. Hecht, Flow Analysis of Computer Programs, North-Holland.New York, 1977.

[HOARE et al 76] C. A. R. Hoare et al, Quasiparallel programming, Software-Practice and experience, Vol. 6, 1976, pp 341-356.

[INGALLS 81] Daniel H. Ingalls, The smalltalk graphics kernel, BYTE, Vol. 6, No. 8, August 1981, pp 168-194.

[INGRES 85] Ingres, Reference manual, version 7, 1985.

[INTEL 78] Intel Corp., ICE-85 In-Circuit Emulator Operating Instructions for ISIS-II Users, Intel Corporation, 1978.

[JACOB 85] Robert J. K. Jacob, A state transition diagram language for visual programming, Computer, Vol. 18, No. 8, August 1985, pp 51-59.

[JANSON 85] Philippe A. Janson, Operating Systems: structures and mechanisms, Academic, 1985.

[JOHNSON & SOLOWAY 85] W. Lewis Johnson and Elliot Soloway, PROUST, Artificial Intelligence, Vol. 10, No. 4, April 1985, pp 179-190.

[KAY & GOLDBERG 77] A. Kay and A. Goldberg, Personal Dynamic Media, Computer, March, 1977.

[LEDOUX 85] Carol Helfgott LeDoux, A knowledge-based System for debugging Concurrent software, PhD thesis, Computer Science Department, School of Engineering and Applied Science, University of California, Los Angeles, CA 90024, 1985.

[LEDOUX & PARKER 85] Carol H. LeDoux and D. Stott Parker, JR., Saving Traces for ADA debugging, ADA in use, Proceedings of the ADA International Conference, Paris, 14-16, May 1985, pp 97-108.

[MAIO et al 85] A. Di Maio et al, Execution monitoring and debugging tool for ADA using Relational Algebra, ADA in use, Proceedings of the ADA International Conference, Paris, 14-16, May 1985, pp 109-123.

[MAUGER & PAMMETT 85] Claude Mauger and Kevin Pammett, An evert-driven debugger for ADA, ADA in use, Proceedings of the ADA International Conference, Paris, 14-16. May 1985, pp 124-135.

[MINSKY 75] Marvin Minsky, A framework for representing knowledge, The Psychology of Computer Vision, McGraw-Hill, 1975.

[NEWSTED et al 81] Peter R. Newsted et al, The impact of programming styles on debugging efficiency, ACM SIGSOFT, Software Engineering Notes, Vol. 6, No. 5, October 1981, pp 14-18

[PERROTT & RAJA 77] R. H. Perrott and A. K. Raja, Quasiparallel tracing, Software-Practice and experience, Vol. 7, 1977, pp 483-492.

[PLATTNER 84] Bernhard Plattner, Real-Time Execution Monitor, IEEE Transaction on Software Engineering SE-10 (November, 1984), pp 756-764.

[POWELL & LINTON 83a] Michael L. Powell and Mark A. Linton, Database Support for Programming Environments, 1983 Database Week: Engineering Design Application, May 23-26, 1983, San Jose, pp 63-70.

[POWELL & LINTON 83b] Michael L. Powell and Mark A. Linton, A Database Model of Debugging, The Journal of Systems and Software, Vol. 3, No. 4, December 1983, pp 295-300.

[RAEDER 85] Georg Raeder, A survey of current graphical programming techniques, Computer, Vol. 18, No. 8, August 1985, pp 11-25.

[RICH 83] Elaine Rich, Artificial Intelligence, McGraw-Hill, 1983.

[RUSTIN 71] Randall Rustin, Debugging Techniques in Large Systems, Courant Computer Science Symposium 1, Prentice Hall, 1971.

[SIMON 69] Herbert A. Simon, The sciences of the Artificial, MIT Press, Cambridge, MA., 1969.

[SEDLMEYER et al 83] Robert L. Sedlmeyer, Knowledge-based Fault Localization in Debugging, The Journal of Systems and Software, Vol. 3, No. 4, December 1983, pp 301-306.

[SEIDNER and TINDALL 83] Rich Seidner and Nick Tindall, Interactive Debug Requirements, ACM SIGSOFT/SIGPLAN Notices: Symposium on Debugging, Vol. 3 (March, 1983), pp 9-22.

[SOWA 84] John F. Sowa, Conceptual Structures: Information processing in mind and machine, Addison-Wesley, 1984.

[STRUBLE 75] George W. Struble, Assembler Language Programming with the IBM System/360 and 370, Addison-Wesley, 1975.

[TANDEM 83] Tandem Non-Stop II Systems, Guardian Operating System Programming Manual, Volume 1 and 2, 1983.

[TAYLOR 83] Richard N. Taylor, A general-purpose Algorithm for Analyzing Concurrent Programs, Communications of ACM, Vol. 26, No. 5, May 1983, pp 362-376.

[TEKTRONIX 81] Tektronix, Inc., 8550 Microcomputer

Development Lab - System User's Manual, User's Guide (DOS/50 Version 1.x), Tektronix Inc., 1981.

[TESLER 81] Larry Tesler, The smalltalk environment, Byte, Vol. 6, No. 8, August 1981, pp 90-147.

[THEAKER & BROOKES 83] Colin J. Theaker and Graham R. Brookes, A Practical Course on Operating Systems, The MacMillan, 1983.

[THEBAUT et al 85] Stephen M. Thebaut et al, Identifying error-prone software - An empirical study, IEEE Transactions on Software Engineering, Vol. SE-11, No. 4, April 1985, pp 317-324.

[TUTHILL 85a] Bill Tuthill, Getting the bugs out with dbx, UNIX Review, Vol. 4, No. 1, January 1986, pp 78-85.

[TUTHILL 85b] Bill Tuthill, Debugging with adb, UNIX Review, Vol. 3, No. 12, December 1985, pp 66-71.

[UNIX 83] 4.2 Berkeley Software Distribution, Virtual VAX-11 Version, Unix programmer's manual, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, Berkeley, California, August 1983.

[WEGNER & SMOLKA 83] Peter Wegner and Scott A. Smolka, Processes, tasks, and monitors: A comparative study of Concurrent Programming Primitives, IEEE Transactions on Software Engineering, Vol. SE-9, No. 4, July 1983, pp

446-462.

[YOUNG 81a] Robert A. Young, Enhancements to the Perkin-Elmer Pascal Compiler to support the Symbolic Debugger, Department of Computer Science, Kansas State University, March 1, 1981.

[YOUNG 81b] Robert A. Young, Program logic Manual for the Pascal Symbolic Debugger, Department of Computer Science, Kansas State University, March 1, 1981.

Appendix - USER'S MANUAL

This tracing facility is intended to assist Concurrent C programmers to debug their Concurrent C programs. It enables a user to visualize the overall states of a Concurrent C program in terms of the individual component processes. Each process and its state are represented by an icon. This implementation decision was made in order to allow all processes to be presented on the CRT screen simultaneously. Since Concurrent C is an enhancement of C, it can invoke any of the C library routines and packages [GEHANI & ROOME 84]. To implement the process "icons", a screen package [ARNOLD 83] available on UNIX system is employed. Each process can be represented by a rectangular box and each box possesses three pieces of information: process name, process type and status (state).

A process name is composed of a number which represents the position of a process in the sequence of processes that have been created in a program and the process variable name associated with that process. For instance, '1: b_mgr' in Figure 1 means this process is the second process

created within this concurrent program and the process variable name associated with this process is b_mgr.

Figure 1: A representation of a process by an "icon"

A process type is one of the process types defined within a concurrent program. In this case, the word "manager" on the second line in Figure 1 represents the type of process, "manager", declared for this particular process b_mgr.

The status componert represents a process state and any relevant information related to that state. For example, tne last line of Figure 1 means this process is about to execute a 'select' clause. In this case, the word 'before' is a relevant piece of information related to the state S ('Select').

The details of all the implementated process states for tnis facility are depicted in Table 1.

Abbreviation	Meaning
P:	Process is created and running.
S: before	Before 'select' statement.
S: after	After 'select' statement (impossible if 'terminate' is an alternative).
D: before	Before delay statement (impossible if 'delay' is an alternative).
D: after	After delay statement.
A:xxxxxxx	Immediately after an 'accept' statement with the first seven characters of a transaction name, xxxxxxx.
R:	Before a 'treturn' statement in an 'accept'.
TP:	Before a transaction pointer statement is used for a call.
bC:xxxxxx	Before a transaction call with the first six characters of a transaction name, xxxxxx.
aC:xxxxxx	After a transaction call with the first six characters of a transaction name, xxxxxx.
tT:	Before a timed transaction call.

Table 1: Substates

Unfortunately, due to the size of a terminal screen (24 x 80 characters), the maximum number of boxes that can be displayed at any one time is ONLY eighteen.

To use the facility, the following steps MUST be followed:

1. Be sure that your Concurrent C program

does not have any compilation error, but warnings are tolerable.

2. To enable the tracing capability be included in your Concurrent C program, you have to invoke a preprocesscr. To invoke the preprocessor, use the following command:

/usrb/chua/beta/TPP filename

where 'filename' is the file name of your Concurrent C program (BE SURE that it has the '.cc' extension).

3. The preprocessor will produce two files called 'filenameT' and 'filenameTRACE'. 'FilenameT' is actually the name of your Concurrent C program with a 'T' inserted just before the '.cc' extension whereas 'filenameTRACE' is a file name of your Concurrent C appended with the word 'TRACE'. For example the name of your Concurrent C program is 'test.cc' then the file name for 'filenameT' is 'testT.cc' and the file name for 'filenameTRACE' is 'test.ccTRACE'.

'FilenameT' is a file which has the tracing

information annotated into your original Concurrent C program. On the other hand, 'filenameTRACE' may contain an error message which is resulted from an error encountered during the conversion and halted the conversion. This error message always specifies which line of your Concurrent C program that the error was detected. This error arises because the Concurrent C compiler is very liberal on syntax checking. However, a report about the number of static processes within this Concurrent C program is always included in the 'filenameTRACE.'

4. If there is an error message reported in the 'filenameTRACE' then the output file, 'filenameT', is incomplete! Thus you have to correct the proolem to eliminate the error. If you think tne problem is in the preprocessor, report the error to the personnel in-charged.

5. If there is no error at all in the 'filenameTRACE' then you may now proceed to compile the outputted file, 'filenameT'. To compile 'filenameT', type the following command:

/usrb/chua/beta/COMPILE 'filenameT'

(Note: Do not include the single quote in the command)

6. If there is any compilation error, try to fix it yourself. Please IGNORE any compilation warning especially "warning: g_temp_pid not used."

7. If the compilation was successful then an object file, 'a.out', would have been produceo.

8. To run the object file, just type the object file name, 'a.out'.

9. During the execution of your object file, a suspension will be taken whenever an output to the CRT or a change in any one of the processes' state is exeucted. During this suspension, a prompt for a number to represent how many output statements to the CRT and how many changes of process' state before the next suspension is to be taken, will be prompted. Furthemore, a number

representing the number of output statements and the number of changes in processes' states executed so far is displayed. If you respond with a '-999' then the program will be executed to completion or to wherever it cannot proceed. However, if you respond with a positive number then the next suspension will not occur until the sum of the number of output statements and the total number of processes' state changes equal to that number you submitted. Any other respond will be treated as a positive one.

10. If your object file does not run to completion with the below message displayed,

"****** NORMAL TERMINATION ******"

you have to reset your terminal by typing the command:

'/usrb/chua/beta/UNDOCRT'

before any other commands; otherwise your terminal will behave strangely!

User Restrictions

a) User's "include" files CANNOT be included in a user's Concurrent C program.

b) All transaction calls issueo by global processes do not have any tracing information annotated with them in order to oiscourage usage of global processes.

c) Timed transaction calls may not work very well because most of them will be expired before rendezvous can occur as processes will be slow down by any "breakpoint."

d) If a user's orogram is too large (i.e., about 36 Kbytes of compiled code), compilation error is very likely with the error message "internal /usr/local/lib/ccpp error: input buffer overflow."

e) Use the built-in constant, null_pid, for null process ic instead of the value zero.

f) DO NOT use a "#define" to define any reserved symbols (like "{", "}", etc) or words (like 'select', 'accept', etc).

g) DO NOT use a "typedef" to define any process types but a "typedef" may be used to define any types of transaction pointers.

h) All screen I/O statements (like "printf", "scanf", etc) must be in a process body and NOT in a function.

Reference:

[ARNOLD 83] Kenneth C. R. C. Arnold, Screen Updating and Cursor Movement Optimization: A Library Package, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, Berkeley, California, 1983.

[GEHANI & ROOME 84] N. H. Gehani and W. D. Roome, Concurrent C, AT&T Bell Laboratories, 1984.

TECHNIQUES TO FACILITATE THE DEBUGGING OF CONCURRENT PROGRAMS

by

HONG YAU CHUA

B. S., Kansas State University, 1983

AN ABSTRACT OF A MASTER'S THESIS

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1986

ABSTRACT:

The differences between sequential and concurrent programs are identified. These differences dictate how the design of a debugger for concurrent programs must differ from a sequential debugger. Different techniques to facilitate debugging of concurrent programs are discussed. The implementation of a tracing facility for Concurrent C programs is presented. This implementation enables all of the processes in a Concurrent C program to be presented by icons on a CRT which keeps track of their individual process state. Furthermore, the ability to breakpoint a process and at the same time suspend all the other existing processes within a Concurrent C program is implemented in this facility.