The Pennsylvania State University
The Graduate School
Capital College

BACI Debugger: A GUI Debugger for the BACI System

A Master’s Paper in
Computer Science
by
David Strite

©2001 David Strite

Submitted in Partial Fulfillment
of the Requirements
for the Degree of
Master of Science

November 2001
Abstract

Due to the increasing importance of concurrent programming and cluster systems, good understanding of concurrency and its impact on process synchronization is more important than ever. Since concurrency introduces design and execution issues not found in sequential programming, it is important that students gain hands on experience doing concurrent programming. The best way to get this experience is by using a system developed specifically for teaching concurrent programming. The Ben-Ari Concurrent Interpreter (BACI) is such a system. Unfortunately, when errors are encountered in BACI programs, they are cumbersome to debug because there had been no satisfactory debugger for the system. The purpose of this project was to develop a graphical user interface debugger for the BACI language to make debugging concurrent BACI programs easier. The debugger replicates the functionality of the BACI interpreter and adds the functionality of a debugger. The debugger allows students to see what is happening while their programs are executing, which helps in understanding concurrent programming.
# Table of Contents

Abstract

Acknowledgement

List of Figures

1 Introduction

2 Concurrent Programming and Process Synchronization
   2.1 Definition
   2.2 Examples
   2.3 Semaphores
   2.4 Monitors

3 Teaching Concurrent Programming and Process Synchronization

4 Ben-Ari Concurrent Interpreter

5 Requirements for Debuggers

6 BACI Debugger

7 Functionality

8 BACI Debugger User’s Guide
   8.1 Introduction
   8.2 Screen Layout
   8.3 Using BACI Debugger
   8.4 Process Sub-Window
   8.5 Program Control
   8.6 Console Sub-Window
   8.7 Globals Sub-Window
8.8 History Sub-Window 21
8.9 Options 22
8.10 Input 23
9 Design Choices 23
10 Conclusion 25
References 26
Appendix A: Sample Program 28
Acknowledgement

I would like to thank Dr. L. Null for being the advisor for this project. She provided many suggestions and valuable insights. I also want to thank the members of the committee, Dr. T. Bui, Dr. S. Kingan, and Dr. P. Naumov, for reviewing this paper. I am grateful to the IBM Corporation for its financial support. I also wish to thank my wife, Lisa, for her support while working on this project.
List of Figures

1  Dining Philosophers’ Table 3
2  Using a semaphore to enforce mutual exclusion 5
3  Using a semaphore to enforce a process ordering 6
4  Simple Monitor Declaration 6
5  COBEGIN block example 9
6  BACI Debugger screen shot 14
7  Main window, before P-code file is selected 16
8  File chooser dialog 17
9  Process sub-window for the main process 18
10 Details Tab 19
11 Execution Control Buttons 20
12 Console Sub-Window 20
13 Globals Sub-Window 21
14 History Sub-Window 22
15 Options Menu 23
16 Input Dialog 23
1 Introduction

Concurrent programming and process synchronization are topics that are a part of every computer science curriculum. For students to really understand these topics, they must have an opportunity to try these concepts for themselves. BACI is a system designed to give students this experience. Although BACI is good for teaching concurrent programming, there had been no satisfactory environment for debugging these concurrent programs. The purpose of this project was to create a GUI debugger for BACI that performs the functionality of the BACI interpreter and adds the functionality of a debugger. This further assists students in learning about concurrent programming and process synchronization.

2 Concurrent Programming and Process Synchronization

2.1 Definition

Some time-shared computers consist of a single CPU while others have multiple CPUs. With more than one CPU, programs can be executed in parallel. Although this is not possible with a single CPU, processes can share the CPU in such a way that the programs appear to execute in parallel by using an interleaved scheduler. Concurrent programming is the creation of programs that consist of segments that have the potential for parallel execution. It involves the necessary mechanisms for expressing potential parallelism as well as the necessary control structures for the resulting synchronization and communication problems. The two fundamental concepts in concurrent programming are processes and resources. A process is a sequential computation with its own thread of control. Concurrent programs allow multiple processes to share resources.

There are several reasons why a programmer should be interested in concurrency:
• The execution time for a particular calculation can be decreased if the calculation is broken into separate parts that can run in parallel on multiple processors.
• The overall throughput of a system can be increased, because the CPU can be executing a process while another process is waiting for a resource.
• Some problems are more naturally solved with concurrent processes, especially simulations and real-time systems. A few examples are simulation of customer flow in a supermarket and air traffic control systems.
• With multiple threads of control, it is easier to design a reactive system, where an action is triggered when a certain event occurs.
• Concurrent programs give greater control to the programmer than sequential programs, since various programs can be suspended and resumed.
• Understanding concurrency is important for understanding many topics related to computer architecture, especially pipelining and super scalar computers.

Process synchronization is coordination between cooperating concurrent processes. This addresses two important issues: mutual exclusion and process ordering. Process ordering deals with processes that must be run in a specific order. Mutual exclusion addresses the issue of potential corruption of data when concurrent processes share access to data. Because the relative speeds of processes cannot be predicted, the resulting interleaved schedule cannot be determined. This means that when a process reads and writes a variable while another process is doing the same, a race condition can occur where the resulting value of the variable is inconsistent. Therefore this data must be put in a critical section and protected by some type of process synchronization tool.
The nondeterminism exhibited in interleaved schedules also makes locating programming errors difficult since subsequent runs of a program may produce different results. The problems encountered with the radiation therapy machine Therac-25 are an example of the importance of mutual exclusion and the problems that can be experienced by nondeterministic results. Problems with the machine caused several patients to receive massive overdoses of radiation, resulting in severe injuries or death. While the cause of the problems was a complex set of factors, one of the main contributing factors was an error in the coding of a critical section involving multiple shared variables [14].

2.2 Examples

A classic example of process synchronization is the Dining Philosophers problem. In this problem, there are five philosophers seated around a circular table. Each of these philosophers divides his time between thinking and eating. Between each pair of philosophers there is a chopstick. Figure 1 shows how the philosophers’ table is configured. When a philosopher gets hungry, he tries to pick up the closest two chopsticks, one at a time. When a philosopher is holding two chopsticks, he is able to eat. When he has finished eating, he puts down the chopsticks. If one of the chopsticks needed by the philosopher is in use by his neighbor, he must
wait for the chopstick to be available. This is a simple model for a process synchronization problem. Each philosopher is a concurrent process and the chopsticks are shared resources that the processes need to access in a mutually exclusive manner [15].

A simple example of how data can become inconsistent can be seen by considering the following two processes that share a variable, $x$:

<table>
<thead>
<tr>
<th>Process A</th>
<th>Process B</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x = x+1$</td>
<td>$x = x \times 2$</td>
</tr>
</tbody>
</table>

These single source statements are translated into multiple instructions at the machine level:

<table>
<thead>
<tr>
<th>Process A</th>
<th>Process B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register1 = $x$</td>
<td>Register2 = $x$</td>
</tr>
<tr>
<td>Register1 = Register1 + 1</td>
<td>Register2 = Register2 $\times$ 2</td>
</tr>
<tr>
<td>$x = \text{Register1}$</td>
<td>$x = \text{Register2}$</td>
</tr>
</tbody>
</table>

Suppose $x$ has an initial value of 10. Then $x$ should have a value of 21 or 22 after process A and B have been run, depending on the ordering of the processes. Either value is acceptable, but these are the only values any correct schedule involving A and B should produce. Suppose the following interleaving is used:

<table>
<thead>
<tr>
<th>Time</th>
<th>Process</th>
<th>Instruction</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>Register1 = $x$</td>
<td>(copies 10 into Register1)</td>
</tr>
<tr>
<td>2</td>
<td>A</td>
<td>Register1 = Register1 + 1</td>
<td>(Register1 now 11)</td>
</tr>
<tr>
<td>3</td>
<td>B</td>
<td>Register2 = $x$</td>
<td>(copies 10 into Register2)</td>
</tr>
<tr>
<td>4</td>
<td>B</td>
<td>Register2 = Register2 $\times$ 2</td>
<td>(Register2 now 20)</td>
</tr>
<tr>
<td>5</td>
<td>B</td>
<td>$x = \text{Register2}$</td>
<td>(copies 20 into $x$)</td>
</tr>
<tr>
<td>6</td>
<td>A</td>
<td>$x = \text{Register1}$</td>
<td>(copies 11 into $x$)</td>
</tr>
</tbody>
</table>

After running processes A and B, using the above interleaving, the value of $x$ will be 11, which is an inconsistent value. Therefore, access to the shared variable, $x$, needs to be protected. The
two most common process synchronization techniques used to protect shared variables and order processes are semaphores and monitors.

2.3 Semaphores

A semaphore is an integer variable with an associated queue that after initialization can only be accessed through special, atomic semaphore operations: wait and signal. A semaphore can be used to control access to a shared resource or to force a certain ordering of processes. The wait operation checks the value of the semaphore. If the value is greater than zero, the value is decremented by one and the calling process is allowed to continue. If the value is less than or equal to zero, the calling process is blocked and placed on the semaphore’s queue, waiting for the semaphore to be signaled by another process. The signal operation checks the value of the semaphore. If the value is zero and there is at least one process waiting on the semaphore, one of the waiting processes is activated. Otherwise, the semaphore value is incremented by one. In both cases, the calling process is allowed to continue. The initial value of the semaphore determines the number of processes that are allowed in the critical section at one time [15].

Figure 2 shows how to use a semaphore for mutual exclusion. The variable mutex is a semaphore. Typically, it will be initialized with a value of one, which means that only one process will be allowed in the critical section at a time. All critical sections must be placed between corresponding wait and signal calls. Figure 3 shows how to use a semaphore to enforce a certain process order. The variable go is a semaphore initialized to zero. The example forces process B to run before process A since A will wait for B to signal the semaphore go.

```
wait(mutex)
critical section
signal(mutex)
```

Figure 2: Using a semaphore to enforce mutual exclusion
2.4 Monitors

A monitor is a special block of code that contains user-defined functions and variables. The variables defined in a monitor can only be accessed by the functions defined within the monitor. The functions can be called from outside of the monitor. The monitor automatically ensures that only one process can be actively running one of its functions at any one time.

Figure 4 shows the declaration of a monitor called `simpleMonitor`. The monitor contains one variable and one function. The variable, `myVariable`, is only visible from within the monitor and can only be accessed from functions within the monitor. The function, `doSomething`, can be called from outside of the monitor. Since only one process at a time may be active within a monitor, while `doSomething` is being executed no other code in the monitor can be executed.

```c
monitor simpleMonitor {
   int myVariable;
   void doSomething() {
   
   }
}
```

**Figure 4:** Simple Monitor Declaration
In addition to the basic definition of a monitor, special condition variables can be used. A condition variable does not store a value, but is instead a construct that is used to queue waiting processes. A process can perform a wait or a signal on a condition variable. When a process performs a wait on a condition variable, the process is suspended until another process performs a signal on that condition variable. A call to signal on a condition variable will revive one suspended process. If no processes are waiting on the condition variable, the call has no effect. Condition variables give programmers greater flexibility for implementing their own concurrency schemes.

Since only one process can be active within a monitor at one time, processes attempting to enter a monitor must wait if another process is already active in the monitor. A separate queue is kept for each monitor of waiting processes. There is also a separate queue for each condition variable of process waiting on the condition variable [15].

3 Teaching Concurrent Programming and Process Synchronization

The topics of concurrent programming and process synchronization are typically taught in an operating systems class. When these concepts are taught, they will be little more than abstract, theoretical concepts to students unless they have a chance to try these concepts for themselves. There are several ways that students can get hands-on experience: a programming language that supports concurrency constructs, operating system calls in an operating system that supports concurrency constructs, or a programming language designed for teaching concurrent programming. The first option requires the use of a language such as Concurrent Pascal [3][11], Ada [4][13][17], Modula [16], SR [1][10], or Java [12]. Depending on the background of the students, one or more new programming languages would need to be learned before concurrent
programming could begin. The second option requires students to learn the details of a specific operating system. Both of these options have a difficult learning curve, which could distract students from the goal of learning about concurrent programming. The third option is to use a programming language designed for teaching concurrent programming. A language designed for teaching current programming should support the common concurrency constructs and should be similar to a programming language that most students already know. The Ben-Ari Concurrent Interpreter is such a language designed for teaching concurrent programming [5].

4 Ben-Ari Concurrent Interpreter

The Ben-Ari Concurrent Interpreter (BACI) is a system designed to allow students to write and experiment with concurrent programming problems. The system supports two programming languages: one is based on Pascal and the other is based on C++. The Pascal variant is referred to as Concurrent Pascal and the C++ variant is referred to as C++. Both of the BACI languages support a subset of their respective languages and are enhanced with additional features for concurrent programming. Since most students have experience with either Pascal or C++, the majority of students can easily use BACI. BACI supports several synchronization techniques, including general semaphores, binary semaphores, and monitors. In addition to these process synchronization constructs, BACI provides additional features that allow students to implement their own synchronization constructs. For example, semaphores in BACI have a random wake up order. Students can implement fair semaphores where the processes awaken in the order in which they were queued on the semaphore.

The original description and implementation of this system was provided by Ben-Ari in his book Principles of Concurrent Programming [2]. In this original implementation,
Concurrent Pascal was the only language supported and semaphores were the only process synchronization construct supported. The system was later enhanced by Bynum and Camp to its current form which supports both Concurrent Pascal and C--. Support was also added for monitors and binary semaphores [5].

The two BACI languages support only a subset of their respective languages [7][8]. No files are allowed except for standard input and output. A limited set of standard data types is allowed. Concurrent Pascal allows only integer, char, and boolean. C-- allows int and char. A special string type is added to both to support character strings. A set of functions similar to the standard C string functions is defined in both languages to perform common string operations. These simplifications slightly limit what can be done with a BACI program, but it is generally not a problem for the types of programs written when learning about concurrent programming.

BACI defines concurrent programming constructs in both languages. A cobegin/coend block is a special block that is defined in the main program. Each of the function calls within this block starts a new concurrent process. When the end of this block is reached, the main process is paused and the concurrent processes are executed by a random preemptive scheduler. When all of the concurrent processes complete, the main process continues execution. Figure 5 shows the syntax for a cobegin/coend block in both Concurrent Pascal and C--. In both examples, a process is created for each of the three functions, foo1, foo2, and foo3.

<table>
<thead>
<tr>
<th>Concurrent Pascal</th>
<th>C--</th>
</tr>
</thead>
<tbody>
<tr>
<td>COBEGIN</td>
<td>cobegin</td>
</tr>
<tr>
<td>foo1();</td>
<td>foo1();</td>
</tr>
<tr>
<td>foo2();</td>
<td>foo2();</td>
</tr>
<tr>
<td>foo3();</td>
<td>foo3();</td>
</tr>
<tr>
<td>COEND;</td>
<td>}</td>
</tr>
</tbody>
</table>

Figure 5: COBEGIN block example
BACI also defines semaphore and binarysem data types for declaring semaphore variables. The functions initialsem, wait, and signal are defined to operate on semaphores. The initialsem function is used to set the initial value of the semaphore. The wait and signal functions perform the standard wait and signal operations on the semaphore. The data types monitor and condition are defined to support the declaration of monitors and condition variables. Monitor variables can only be declared as global variables and condition variables can only be declared within a monitor. The functions waitc and signalc are defined for waiting and signaling on a condition variable.

Additionally, the following constructs are also supported. A function declared with the atomic keyword cannot be preempted, meaning that the function will be executed in its entirety before a process swap can occur. The suspend function puts the process calling the function to sleep. The revive function wakes up the specified process. The function which_proc returns the process number of the current process. These additional low level constructs allow users to create their own concurrency constructs.

To use the BACI system, a user runs either the Concurrent Pascal or C-- compiler on the file containing the program’s source. The compiler takes this source code file and produces a file containing pseudo machine code (P-code). A P-code instruction is of the form \( f x y \), where \( f \) is the opcode for the instruction and \( x \) and \( y \) are modifiers that are used give the interpreter additional information for the instruction. There is a different opcode for each type of instruction. Appendix A shows a simple C-- program with the corresponding P-code file. As with real machine code, multiple P-code instructions may be needed for each source statement. The user then executes the BACI interpreter on this P-code file. The interpreter executes the P-code instructions. During the execution of the P-code the interpreter simulates the execution of
multiple processes using a random preemptive scheduler. This means that multiple executions of the same program with concurrent processes can produce different results.

5 Requirements for Debuggers

A debugger is a tool that allows a programmer to interact with a program while it is being executed. In addition to the low-level code being executed, the source code that created the program is also displayed. The user is allowed to control the execution of the program. The debugger allows such controls as pausing and resuming the program execution, setting breakpoints, and stepping through the code one statement at a time. The values of variables are displayed while the program is executed. A debugger is an invaluable tool for seeing what is happening while a program is running, thus making it easier to diagnose problems in a program. A debugger is especially useful for a system such as BACI that is designed for use by students.

6 BACI Debugger

The BACI interpreter includes a basic command line debugger. This command line debugger performs the basic functions of a debugger. Like most command line debuggers it is cumbersome to use and not very user friendly. Generally GUI based debuggers are easier to use. The purpose of this project was to implement a GUI based debugger for the BACI system.

The new BACI Debugger allows users to run their BACI programs in an interactive environment. The debugger displays the P-code instructions as they are executed as well as the source code that produced the P-code. Programs written in both Concurrent Pascal and C-- are supported. The debugger allows users to control the execution of the program. Users can set the interpreter to execute the program without pausing or can step through the code. Stepping is
supported at both the source code statement level and the P-code instruction level. Breakpoints are supported to allow the user to specify a specific point for the interpreter at which to pause execution. Breakpoints can be set on a specific line of source code or on a specific P-code instruction. Users can view the values of variables as the program is executed.

7 Functionality

The BACI Debugger replicates the functionality of the interpreter and adds the functionality of a GUI based debugger. The main screen presented to the user is a multiple-document interface window that contains sub-windows. The advantage of this type of screen design is that the user can open, close, move, and resize the sub-windows to display the data he is interested in seeing. Multiple types of sub-windows are available: Process, Globals, Console, and History.

The Process sub-window contains data pertaining to a specific process. One of these sub-windows can be displayed for the main process and each of the concurrent processes. The data in this sub-window is divided into three tabs. The first tab, the Code tab, contains the source code and P-code currently being executed and the value of local variables. The source code file is displayed and the source statement currently being executed is marked. The P-code generated by the current source line is displayed with the current instruction marked. The values of variables in the current function are displayed. The second tab, the Console tab, displays all output written to standard output from this process. The third tab, the Details tab, contains low-level details about the process. This tab displays the process stack, which is used to store the value of variables and for temporary storage used by the interpreter. This tab also displays other detailed information about the current state of the process, such as bounds for the current stack.
frame, whether or not the process is active, and information about a semaphore, monitor, or condition variable on which the process is waiting.

The *Globals* sub-window displays the values of global variables. This includes normal variables, semaphores, and monitors. For monitors, all of the variables declared within the monitor can be displayed. The *Console* sub-window displays data written to standard output. Data written by all processes is displayed here. The *History* sub-window displays a log of the P-code instructions that have been executed. The most recent 100 instructions are displayed. The process index, source file name, source file line number, and the P-code instruction are displayed. This is useful for seeing how the preemptive scheduler has scheduled the concurrent processes.

Figure 6 shows a screen shot of the BACI Debugger. The following features are marked in the figure:

1. **The Process List**: This lists all of the process currently running, including the main process and all of the concurrent processes. Selecting a process from this list will open a *Process* sub-window.

2. **A Process sub-window for one of the concurrent processes**: The *Code* tab which displays source code, P-code, and local variables is displayed.

3. **The area of the Process sub-window displaying source code**: The current line is indicated by the green arrow.

4. **The area of the Process sub-window displaying P-code**: The instructions displayed are the ones corresponding to the current source line. The current instruction is indicated by the green arrow.

5. **The Console sub-window**: This displays the output from all of the processes.
6. The *Globals* sub-window: This displays the values of all global variables.

7. The Control buttons: These allow the user to step through code as well as continue and pause execution.

8. A *Process* sub-window: This displays data for another one of the concurrent processes.

Figure 6: BACI Debugger screen shot
8 BACI Debugger User’s Guide

8.1 Introduction

The Ben-Ari Concurrent Interpreter (BACI) is a system designed to give students hands-on experience with concurrent programming and process synchronization. BACI debugger is a GUI based debugger for the BACI system. It allows users to run their programs in an interactive environment. The debugger displays the P-code instructions as they are executed with the source code that produced the P-code. Programs written in both Concurrent Pascal and C-- are supported. The debugger allows users to control the execution of the program. Users can set the interpreter to execute the program without pausing or can step through the code. Stepping is supported at both the source code statement level and the P-code instruction level. Breakpoints are supported to allow the user to specify a specific point for the interpreter to pause execution. Breakpoints can be set on a specific line of source code or on a specific P-code instruction. Users can view the values of variables as the program is executed. The debugger is started using the following command:

```
java -classpath baci.jar baci.gui.Debugger [pcodefile.pco]
```

where [pcodefile.pco] is an optional P-code file.

The BACI Debugger is written in Java so that it can be run on any platform on which the BACI system runs. A Java 2 (version 1.2 or higher) runtime is required. The debugger requires as input a P-code file and any source code files that the compiler used to create that P-code file. These files must all be in the same directory.
8.2 Screen Layout

The main screen presented to the user is a multiple-document interface that allows multiple sub-windows to be displayed. These sub-windows can be opened, closed, moved, and resized. Several types of sub-windows can be displayed: Process, Globals, Console, and History. These different types of sub-windows are described below. Figure 7 shows the main window before a P-code file has been selected. The two main parts of the screen are the process list on the left and the area on the right where the sub-windows will be displayed. While the interpreter is running, there will be an entry in the process list for each process that is running.

![BACI Debugger](image)

**Figure 7:** Main window, before P-code file is selected
8.3 Using BACI Debugger

To get started, a P-code file must be specified. A P-code file can be chosen by adding the P-code file on the command line when the debugger is started. A P-code file can also be selected using the file chooser dialog by selecting the *Open* option from the *File* menu. Figure 8 shows an example of using the file chooser dialog to select a P-code file.

![File chooser dialog](image)

**Figure 8:** File chooser dialog

Once a P-code file is selected, the debugger will read in the P-code file and the corresponding source files. The information regarding which source files created the P-code file is stored in the P-code file. These source files must be in the same directory as the P-code file. The interpreter is initialized with the data read from the P-code and source files. At this point, the program will be ready to run.

The process list contains entries for the processes that are running. When the interpreter is initialized, the process list will contain an entry for the main process. Selecting an entry in the list and clicking on the open button will open a *Process* sub-window for the selected process. The entry can also be double clicked to open the sub-window. Figure 9 shows the process list with an entry for the main process and a *Process* sub-window opened for the process.
8.4 Process Sub-WINDOW

The Process sub-window displays information specific to one process. One of these sub-windows is available for each process running in the system. These sub-windows are opened by selecting an entry from the process list. The data in this window is divided into three tabs: Code, Console, and Details. The Code tab displays source code, P-code, and variables. The Source area displays the source code file. The line that created the current P-code instruction is indicated with an arrow. The PCode area displays the P-code instructions that are associated with the selected source line. The current P-code instruction is indicated with an arrow. The add and remove buttons in the source code and P-code areas are used to add or remove a breakpoint at the selected line. Breakpoints can be set on a specific source statement or a specific P-code
instruction. When a breakpoint has been set, a red circle will be displayed beside the source statement or P-code instruction.

The **Variables** area displays the names and values of local variables for the block the process is currently executing. Variables that are arrays can be expanded to display the elements of the array. The **Console** tab displays data written to standard output by the process. The **Details** tab displays low level detail about the current state of the process. The process stack is displayed. This is where the interpreter stores the values of variables as well as temporary data used for the execution of instructions. The bottom and top fields define the range of the current stack frame. The active field is true if the process is in a runnable state and false if it is suspended. The finished field is true if the process has finished executing and false otherwise. The suspend field will contain an address of a variable that the process is waiting on. This could be a semaphore, monitor, or condition. The monitor field will contain the address of a monitor if the process is currently executing within a monitor. The priority field will contain the priority of the process. Figure 10 shows a Details tab.

![Figure 10: Details Tab](image-url)
8.5 Program Control

The execution control buttons are displayed on a toolbar at the top of the main window. Figure 11 shows these buttons. The *Run* button will make the debugger run the program without pausing. Execution will stop when one of the following occurs: the end of the program is reached, the *Pause* button is pressed, or a breakpoint is encountered. The *Pause* button causes the debugger to pause the program execution if it is currently running. Pressing the *Step Source* button tells the debugger to execute one line of source code. Execution will be paused after the line is executed. The *Step Pcode* button tells the debugger to execute exactly one P-code instruction. Execution will be paused after the instruction is executed. These options are also available on the *Control* menu. The *Control* menu additionally has a *Restart* option that will restart the debugger at the beginning of the program.

![Figure 11: Execution Control Buttons](image)

8.6 Console Sub-Window

The *Console* sub-window displays data written to standard output. Data written from any of the processes is displayed here. The window is opened by selecting the *Console* option on the *Window* menu. Figure 12 shows the console sub-window.

![Figure 12: Console Sub-Window](image)
8.7 Globals Sub-Window

The *Globals* sub-window displays the values of all global variables. For monitors, the variable can be expanded to display the values of variables that are declared within the monitor. Like the variable display on the *Process* sub-window, array variables can be expanded to display the elements of the array. This sub-window is displayed by selecting the *Globals* option from the *Window* menu. Figure 13 shows the *Globals* sub-window.

![Figure 13: Globals Sub-Window](image)

8.8 History Sub-Window

The *History* sub-window displays a log of P-code instructions that have been executed. The most recent 100 instructions are displayed. The process number, source code file name, source code file line number, and P-code instruction are displayed. This is useful for seeing how the preemptive scheduler has scheduled the concurrent processes. This sub-window is displayed by selecting the *History* option from the *Windows* menu. Figure 14 shows the history sub-window.
8.9 Options

The Options menu contains options that allow the user to determine certain behaviors of the debugger. The Pause on Process Swap option determines if the debugger should pause when the preemptive scheduler swaps processes. If this option is checked, the debugger will pause when a process swap occurs. If unchecked, the debugger will not pause on process swaps. The Show Active Window option determines if the Process sub-window for the active process should always be displayed. If this option is checked, the Process sub-window for the new active process will be displayed at each process swap. On a process swap, the corresponding Process sub-window will be created if it has not been opened yet or brought to the foreground if it is already open. If this option is unchecked, the debugger will not automatically display Process sub-windows for the active process. The Use Random Scheduler option determines if the interpreter will use a random scheduler. If this option is checked, a random scheduler will be used. If unchecked, a deterministic scheduler will be used. This will cause multiple runs of a program to produce the same results. Figure 15 shows the options menu.
8.10 Input

If the program requests input from standard input, the input dialog will be displayed. This allows the user to type in the requested data. Entering no data will be treated as an EOL character. The EOF character is not supported. Figure 16 shows the input dialog.

9 Design Choices

The BACI system is available for multiple operating systems. Several Unix flavors and DOS (including Windows) are supported. Since BACI can be used on these various operating systems, it was necessary to design BACI Debugger to work on all of these systems as well. Java was chosen as the language to use for implementing the BACI Debugger. It can be run on any platform that supports Java 2 (version 1.2 or higher). Java is a good choice for programs that need to be run on multiple platforms, because Java programs have a standard behavior across platforms and can be copied from one system to another without being recompiled.
In addition to performing the functions of a debugger, BACI Debugger needs to perform all the functions of the BACI interpreter. It was necessary to learn the details of the P-code instructions and the inner workings of the interpreter in order to implement the functionality of the interpreter. The BACI compilers and interpreter use many fixed upper bounds on the sizes of the various tables, such as the number of processes, the number of P-code instructions, the size of the stack, and the amount of string space. In order to make the debugger as general as possible, memory is allocated dynamically so that these fixed bounds are not necessary. This allows these constants to be changed in the compilers and interpreter without the debugger needing to be changed, so future modifications to the BACI system should pose no problem with the GUI debugger.

The Java classes of the debugger are divided into three main groups: program, interpreter, and GUI. The classes in the program group are related to the input P-code and source files. The Program class stores all of the data tables from the P-code file as well as the source code. Each type of P-code instruction is represented by a class extending from an abstract class, PcodeInstruction. Having these separate classes allows the check on the opcode to be performed only when the instruction is read from the P-code file. When the instruction needs to be executed or displayed to the screen the opcode does not need to be checked. The interpreter classes are involved with actually running a program. The Interpreter class is responsible for keeping track of processes that are running and the overall state of the system, running P-code instructions, and swapping between processes. The BaciProcess class is used to store information specific to one process. The classes in the GUI group are all related to displaying data on the screen. The Debugger class is the class that is started to run BACI Debugger. It is responsible for instantiating an Interpreter and DebuggerFrame. The main outer
window displayed to the user is an instance of DebuggerFrame. Each type of sub-window is represented by a class extending from the abstract class BaciWindow.

10 Conclusion

Concurrent programming and process synchronization are important topics for computer science students to understand. These topics are difficult to understand without practical hands on experience. BACI is a system designed to give students an opportunity to write and experiment with concurrent programs. Because concurrent programs are nondeterministic, finding problems in concurrent programs can be difficult because subsequent runs of the same program may produce different results. In this project, a GUI debugger was developed for the BACI system. BACI Debugger performs the functions of the BACI interpreter and adds the functionality of a debugger. It is designed to help students learn about concurrent programming and process synchronization by allowing them to see what is happening while their concurrent program is running.

Future work for the BACI Debugger could include adding support for Distributed BACI. Distributed BACI is currently being developed by Bynum, Camp, and Burdette to give students an opportunity to write concurrent programs that run in a distributed environment [9].
References


Appendix A: Sample Program

**simple.cm:**

```c
semaphore mutex;

void sayHello(char name) {
    wait(mutex);
    cout << "Hello from " << name << endl;
    signal(mutex);
}

void main() {
    initialise(mutex,1);
    cobegin {
        sayHello('A');
        sayHello('B');
        sayHello('C');
        sayHello('D');
    }
}
```

**simple.pco:**

BACI System: C-- to PCODE Compiler, 11:27 09 Feb 2001
Source file: simple.cm Sun Nov 18 20:52:56 2001

```
0  32                 PCODE table
lc  f   x   y
 0  0  0  0
 1  6  0  0
 2 28  0  0
 3  1  1  5
 4 29  0  3
 5 63  0  0
 6  0  0  0
 7  7  0  0
 8 32  0  0
 9 80  0 32
10  0  0  0
11 24  0  1
12 39  0  0
13  4  0  0
14 18  0  3
15 24  0 65
16 19  0  5
17 3  0  1
18 18  0  3
19 24  0 66
20 19  0  5
21 3  0  1
22 18  0  3
23 24  0 67
24 19  0  5
25 3  0  1
26 18  0  3
27 24  0 68
```
28 19 0 5
29 3 0 1
30 5 0 0
31 31 0 0
32 81 0 0
1 5 IDENTIFIER table
index identifier link obj type ref normal lev adr mon atomic
1 ++-outer-++ 0 7 0 0 1 0 32 0 0
2 mutex 1 1 5 0 1 0 0 0 0
3 sayHello 2 3 0 1 1 0 0 0 0
4 name 0 1 3 0 1 1 5 0 0
5 main 3 6 0 2 1 0 9 0 0
0 2 BLOCK table
index last lastpar psize vsize
 0 5 5 0 1
 1 4 4 6 6
 2 5 5 5 5
0 -1 ARRAY table
index inxtype eltyp elref low high elsize size
0 12 60 STRING table
Hello from
0 0 Input File array
index parent file name
 0 -1 simple.cm
0 13 PCODE debugging information
lc findex flineno
 0 0 4
 2 0 5
 6 0 6
 8 0 7
 9 0 10
10 0 11
13 0 12
14 0 13
18 0 14
22 0 15
26 0 16
30 0 17
31 0 18
33 0 -18