BACI Debugger User's Guide

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.

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 1 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.

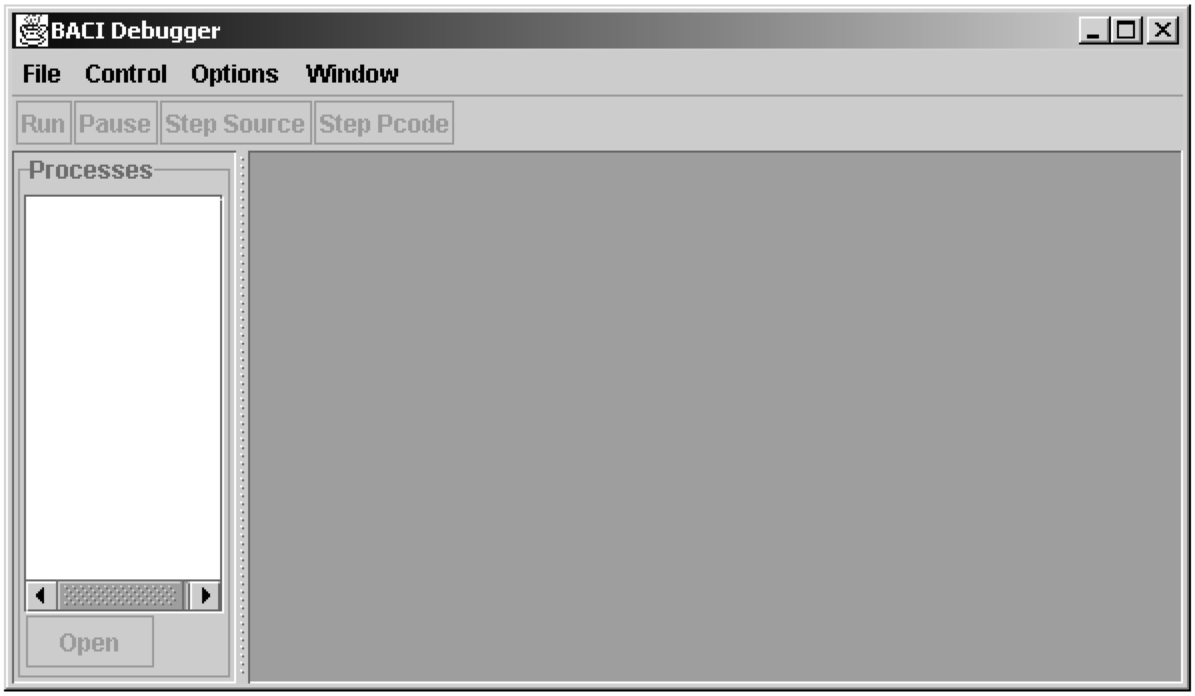
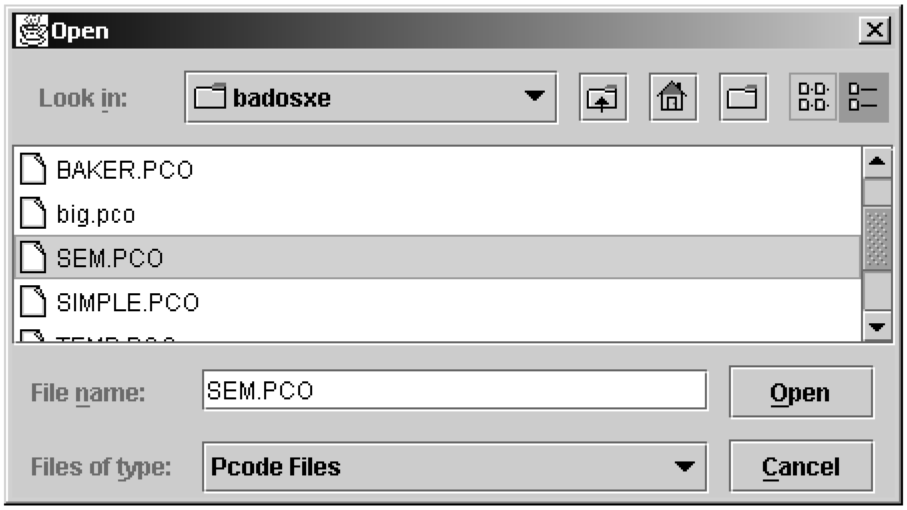


Figure 1: Main window, before P-code file is selected

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 2 shows an example of using the file chooser dialog to select a P-code file.



Look in: badosxe

BAKER.PCO
big.pco
SEM.PCO
SIMPLE.PCO

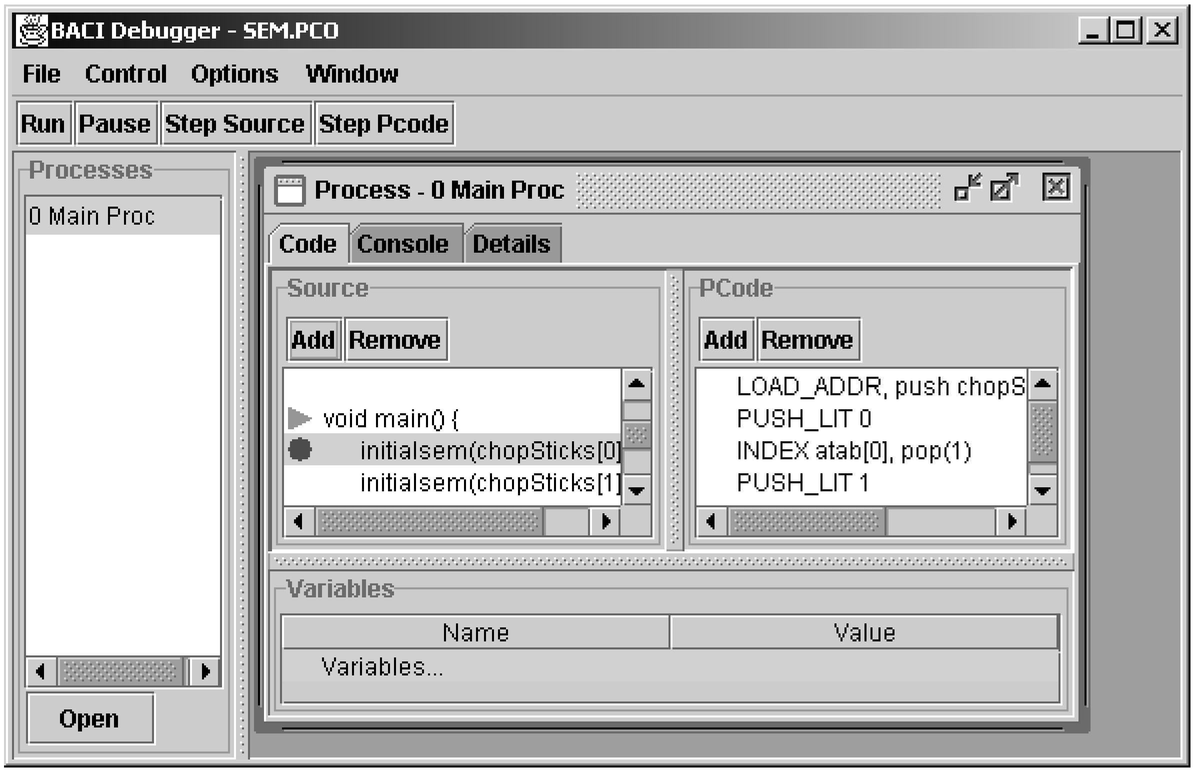
File name: SEM.PCO

Files of type: Pcode Files

Figure 2: 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 3 shows the process list with an entry for the main process and a *Process* sub-window opened for the process.



0 Main Proc

```
void main() {
    initialsem(chopSticks[0]
    initialsem(chopSticks[1]
```

```
LOAD_ADDR, push chopS
PUSH_LIT 0
INDEX atab[0], pop(1)
PUSH_LIT 1
```

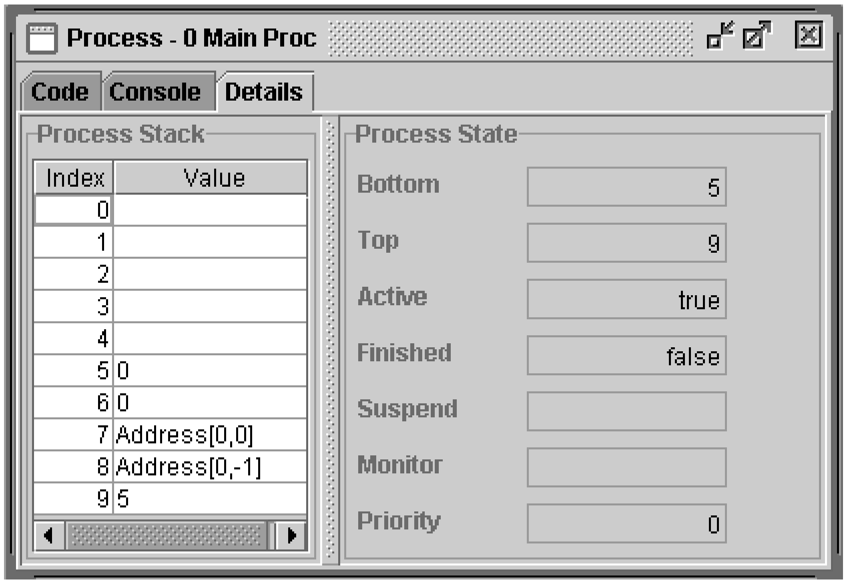
Variables...

Figure 3: Process sub-window for the main process

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 4 shows a Details tab.



Index	Value
0	
1	
2	
3	
4	
5	0
6	0
7	Address[0,0]
8	Address[0,-1]
9	5

Bottom	5
Top	9
Active	true
Finished	false
Suspend	
Monitor	
Priority	0

Figure 4: Details Tab

Program Control

The execution control buttons are displayed on a toolbar at the top of the main window. Figure 5 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.

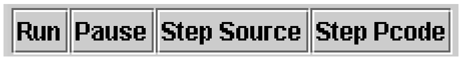
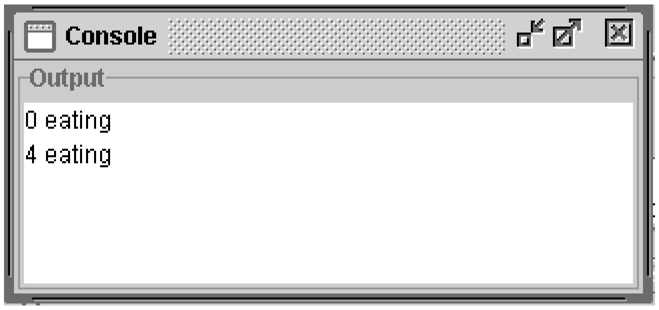


Figure 5: Execution Control Buttons

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 6 shows the console sub-window.

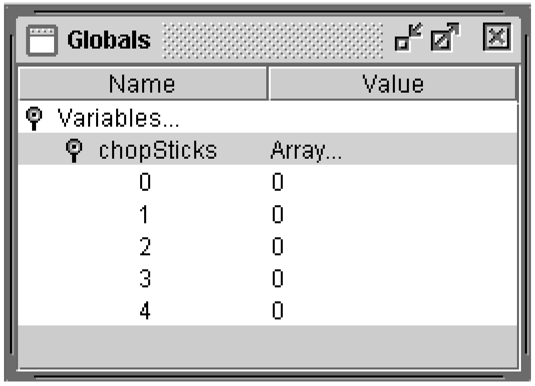


```
0 eating
4 eating
```

Figure 6: Console Sub-Window

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 7 shows the *Globals* sub-window.

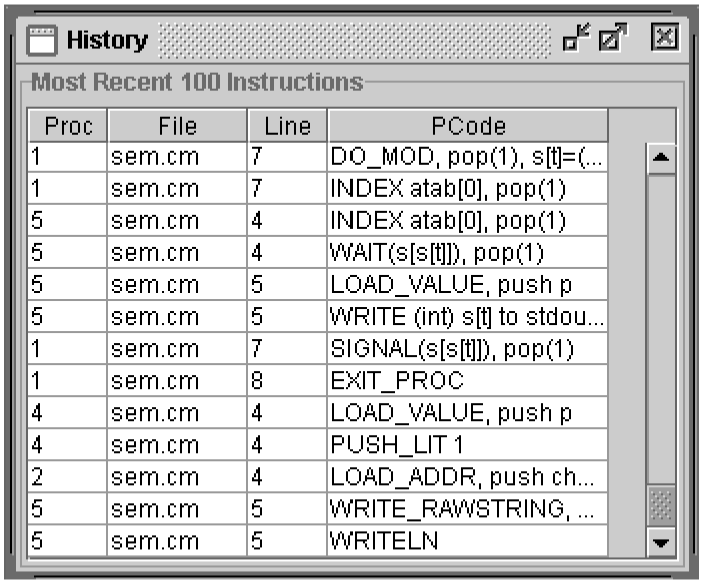


Name	Value
Variables...	
chopSticks	Array...
0	0
1	0
2	0
3	0
4	0

Figure 7: Globals Sub-Window

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 8 shows the history sub-window.



Proc	File	Line	PCode
1	sem.cm	7	DO_MOD, pop(1), s[t]=(...
1	sem.cm	7	INDEX atab[0], pop(1)
5	sem.cm	4	INDEX atab[0], pop(1)
5	sem.cm	4	WAIT(s[s[t]]), pop(1)
5	sem.cm	5	LOAD_VALUE, push p
5	sem.cm	5	WRITE (int) s[t] to stdou...
1	sem.cm	7	SIGNAL(s[s[t]]), pop(1)
1	sem.cm	8	EXIT_PROC
4	sem.cm	4	LOAD_VALUE, push p
4	sem.cm	4	PUSH_LIT 1
2	sem.cm	4	LOAD_ADDR, push ch...
5	sem.cm	5	WRITE_RAWSTRING, ...
5	sem.cm	5	WRITELN

Figure 8: History Sub-Window

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 9 shows the options menu.

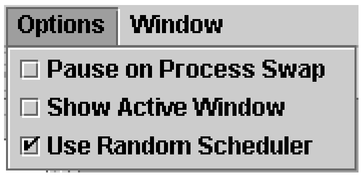


Figure 9: Options Menu

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 10 shows the input dialog.

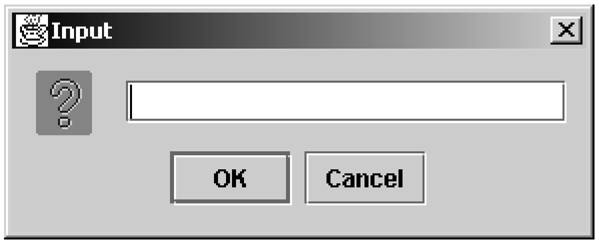


Figure 10: Input Dialog