

JBACI Concurrency Simulator

User's Guide

Moti Ben-Ari

July 18, 2004

Copyright (c) 2003-4 by Mordechai (Moti) Ben-Ari.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with Invariant Section “History,” no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the file `fdl.txt` included in this archive.

1 History

Over twenty years ago I developed a concurrency simulator for teaching concurrent programming.¹ The simulator was based upon Niklaus Wirth's Pascal-P interpreter and was subsequently published in its entirety in my textbook *Principles of Concurrent Programming* (Prentice-Hall International, 1982). Later, I made many modifications to the simulator, in particular, I developed an Integrated Development Environment (IDE) based upon the user interface features in Turbo Pascal. The software has now become too fragile to maintain and the character interface is unattractive to students.

In the intervening years other educators have developed concurrency simulators of their own. The most comprehensive and widely used is BACI, and I am flattered that it was named after me (Ben-Ari Concurrency Interpreter). BACI (http://www.mines.edu/fs_home/tcamp/baci) was developed at William and Mary College by Bill Bynum and Tracy Camp. While versions of BACI exist for several systems, a graphical user interface (GUI) exists only for Unix systems. Recently, David Strite of Penn State Harrisburg developed a new interpreter for the BACI virtual code that includes a GUI. The interpreter (<http://cs.hbg.psu.edu/~null/baci>) is written in Java and thus portable over all platforms.

JBACI is an integration of the original BACI compilers and Strite's interpreter into an IDE that contains an editor, together with extensions to the GUI to simplify its use by novices. In addition, I have modified the compilers and interpreter to include commands for drawing graphics on a canvas; this enables the student to write concurrent programs that are more fun than character-based programs. JBACI also supports Linda-like synchronization primitives.

¹Cheap concurrent programming, *Software—Practice and Experience* 11(1981), 1261–1264.

2 Installation and execution

The student distribution is in two zip files called `jbaciN.zip` and `jbaciNdocs.zip` where `N` is a version number. The distribution contains the executable jar file and several directories: `compilers` contains the exe files of the Pascal and C compilers; `docs` contains documentation files and `examples` contains sample programs in both Pascal and C. The zip file `jbaciNsrc.zip` contains the directory `baci` with the Java source code, as well as the directory `compilers` which contains files for rebuilding the compiler to accept the graphics commands.

You must have Java (SDK or JRE) Version 1.4 installed in order to run `JBACI`. Open the distribution `jbaciN.zip` into the directory `\jbaci`. To run `JBACI`, execute `java -jar jbaci.jar`. You can also create a shortcut to the jar file and use the icon supplied. There are also batch files: `run.bat` which runs the above command, and `runc.bat` and `runcm.bat` which allow a source file name without the extension `pm` or `cm` to be included.

This document describes the operation of the `JBACI` IDE and the graphics extensions only. For documentation of the dialects of Pascal and C that are accepted, as well as of the synchronization primitives, refer to the original `BACI` documentation which for convenience is included in the directory `doc`.

Most aspects of the GUI are defined in the file `Config.java` and can be easily changed. Certain options can also be changed by editing the configuration file `config.cfg`, which is written out when `JBACI` is closed. Note that the configuration includes in the location of the executable compiler files and the default source directory, so they must be changed if you do not install to the default directory or need to use Unix file syntax.

3 The Integrated Development Environment

The JBACI IDE is similar to other IDEs. When you run the program you are presented with a menu structure, as well as with a toolbar of buttons for common operations. Selection of menu entries and buttons using mouse clicks (or keyboard mnemonics and shortcuts) will be familiar to a user of GUIs. In the following description, button names like Save will be used even when a menu entry like File/Save also exists.

The IDE can be in one of two states: the *edit* state and the *run* state. Edit and Run change from one state to the other. Buttons and menu entries are enabled and disabled accordingly. In both states you can select File/Exit and Help/About .

3.1 Editing and compiling

Figure 1 shows the JBACI window when you are editing a source file. The edit buffer contains the

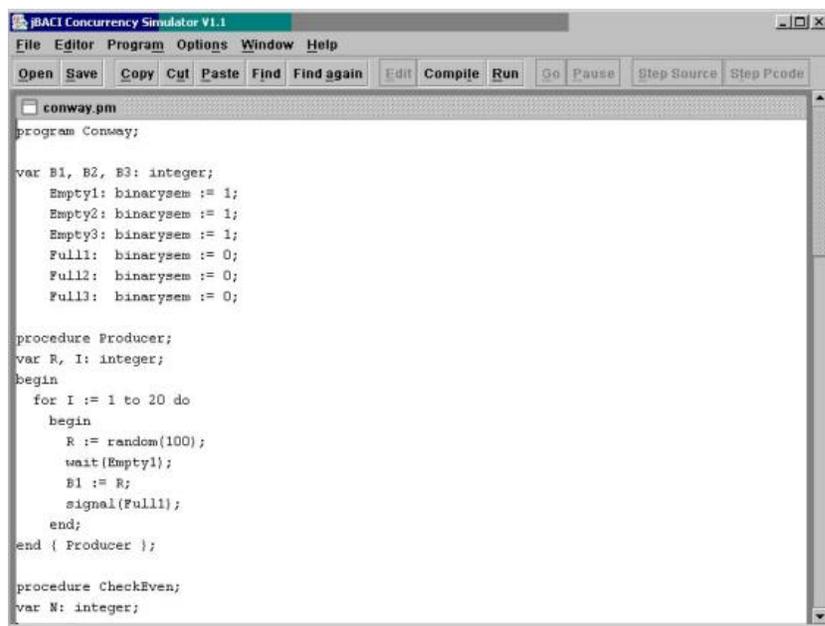


Figure 1: JBACI edit mode

file `conway.pm` and the only buttons enabled are those relevant to editing and compiling, and the Run button to change to the *run* state. The file and editing operations in the *edit* state are familiar: Open, New, Save, Save as, Copy, Cut, Paste, Find, Find again . Files are displayed in a text area and the usual editing keys can be used.

Source files must have the extension `cm` or `pm` for C and Pascal, respectively. This enables the `Compile` operation to select the correct compiler automatically. The output of the BACI compiler is displayed in a popup frame, which can be erased by pressing `Enter`. If there are compilation errors, the cursor will then be positioned at the beginning of the line with the first error.

3.2 Running the program

When a program has been successfully compiled, select `Run` to enter the *run* state and begin program execution (Figure 2). The only buttons enabled are those relevant to running the program,

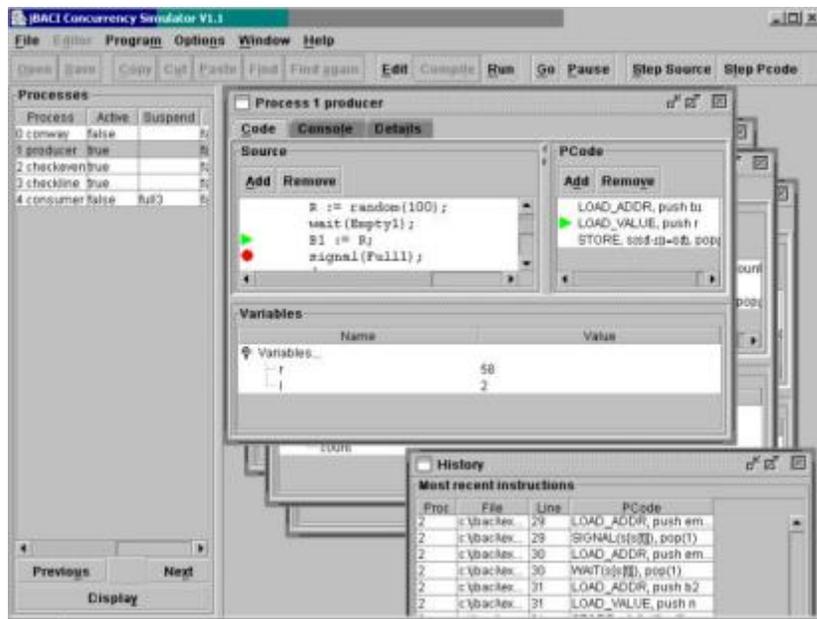


Figure 2: JBACI run mode

and the `Edit` button to return to the *edit* state.

Selecting `Go` will execute the program normally; the execution can be halted by selecting `Pause` and restarted from the beginning by reselecting `Run`. If `Options/Pause on Process Swap` is selected, the execution is automatically halted when a context switch between processes is done.

`Step Source` and `Step PCode` enable you to execute a single step of the program, either a single step of source code or a single step of the PCode. A source code step is defined by a single *line* in the source code file, regardless of how many language statements appear on the line. Writing several statements on a single line enables you to step over them quickly when they have no algorithmic importance (like initialization statements).

Breakpoints can be set by clicking on a source code or PCode line in a process window (see below) and then selecting the Add button above the display of the code. A red dot will appear in front of the line. To remove a breakpoint, click on the line and select Remove .

3.3 Displays

When executing a program the window below the toolbar is divided into two areas. The size of the areas can be adjust by dragging the dividing bar with the mouse. The left area contains the process table and the right area is used to display the system windows and the process windows.

3.3.1 Process table

The process table contains a line for each process, giving the process number and name and the concurrency status, for example, if the process is Active and if it is Finished . The field Suspended is non-empty if the process is suspended on a semaphore, a monitor entry or a monitor condition. The field Monitor will display the monitor name if the process has entered a monitor procedure. See the BACI documentation for the meaning of Priority and Atomic .

The “next instruction” to be executed will be from the process whose entry is highlighted. You can change the highlighted entry by selecting Next or Previous , or by clicking on the process table entry or the process window. This will also display that the process window of the highlighted process (if it is not already displayed).

3.3.2 System windows

There are four system windows:

- The Console window displays the output from the program; this is in addition to the output from each process which is displayed in a pane of the process window.
- The Globals window displays the values of the global variables. In a Pascal program the “global” variables are actually local to the main program, so the Globals window will be empty.
- The History window displays the last 150 source or PCode instructions that have been executed. (The History window is displayed at the bottom right of Figure 2.) Options/History on Source Step selects whether an entry will be added to the History window after each source step or each PCode step; if this is selected, the PCode will not appear in the display.
- The Linda window displays the tuple space.

You can turn the display of these windows on or off by selecting entries in the `Window` menu. The system windows displayed by default is controlled by flags in the configuration.

3.3.3 Process windows

The heart of the display in the *run* state is the set of process windows, one for each process. A process window consists of three tabbed panes: the Code pane, the Console pane and the Details pane. The Console pane show the output from the process, but since the output is also shown on the global Console window, you may not need to display this pane. The Details pane shows the contents of the process stack as well as the concurrency status of the process. However, since the concurrency status of all processes is displayed in the process table, the Details pane is mainly useful if you are tracing the effect of PCode operations on the stack.

The Code pane is itself divided into three areas: the Source area, the PCode area and Variables area. The Source area shows the source code of the *entire program*, not just the code of the process. A green arrow indicates the next source line to be executed. Breakpoints are denoted by red dots, and the green arrow is colored red when a breakpoint is reached. The PCode area displays the PCode for the *current source line*. The same indications are used as for the Source area. The Variables area displays the values of the local variables of the process. The variables are displayed in a tree format, so that values of arrays can be expanded or folded.

3.3.4 Displaying process windows

The default in jBACI is to display a process window when the instruction being executed has been taken from that process, that is, the process window is displayed if it has not been displayed before, and brought to the front of the frame if it has. You can deselect `Options/Show Active Window` , in which case, the window for the active process will not be displayed or brought to the front.

3.4 History file

The history of all source or PCode instructions executed can be written to a file, as selected by `Options/Write History File`. The file has the extension `his` and is re-opened whenever `Run` is selected. (Make sure to select `Run` again if you change the option.) The file is flushed and closed when the end of the program is reached. If you to flush and close a file before termination or for a non-terminating program, you must select `Edit` or `File/Exit`, and then save the file before running the program again.

4 Language extensions

4.1 Graphics

The portable Java API has been used in JBACI to enable graphics routines to be used in concurrent programs. This is useful in order to demonstrate synchronization in a game-like environment. The graphics facilities have been based on the shapes example from BlueJ (<http://www.bluej.org>), which uses a single hidden Canvas upon which graphics figures are drawn (Figure 3).

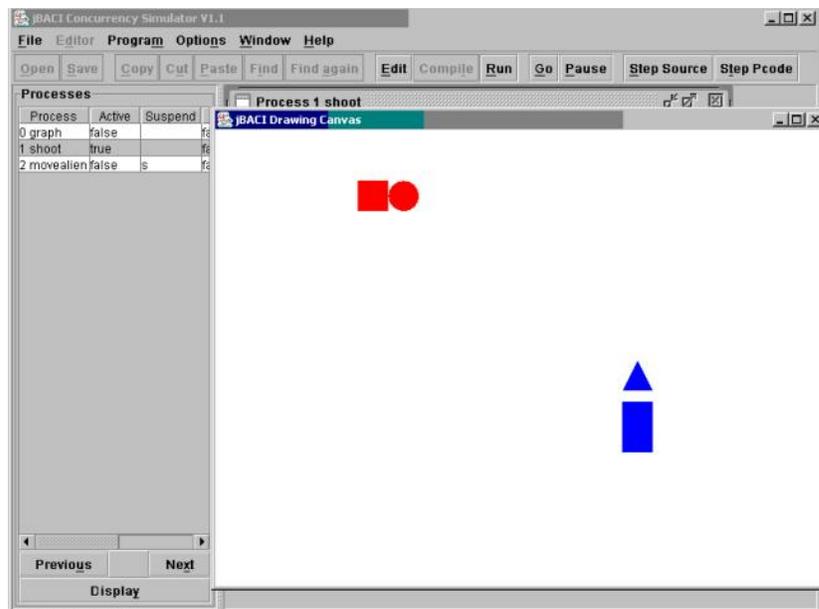


Figure 3: JBACI graphics canvas

Each graphics figure is given a *handle* when it is created so that subsequent graphics commands can refer to that figure. There are five procedures for drawing the figures:

- `create(handle, figure, color, x, y, size1, size2)`: creates a graphics figure with the specified color, location and size.
- `changecolor(handle, color)`: changes the color of the figure.
- `makevisible(handle, flag)`: displays or hides the figure according to the flag (1 or 0).
- `moveto(handle, x, y)`: moves the figure to a new position.
- `moveby(handle, deltax, deltay)`: moves the figure relative to its current position.

All parameters are of type integer. The encodings for the figures (circle, line, rectangle, triangle) and colors (red, black, blue, yellow, green, magenta, white) are given by constants declared in the include files `gdefs.pm` and `gdefs.cm`. These files as well as sample programs (`graph.pm`, `alien.pm`, `graph.cm`) can be found in the directory `examples`.

The positions and sizes are in pixels; the top left corner is (0,0) and the bottom right corner is (600,450), which can be changed in `Config.java`. The meaning of the location and size parameters for each figure is as follows:

Circle The location is the top left corner of the circumscribed square; `size1` is the diameter and `size2` is ignored.

Triangle The location is the top vertex of an isosceles triangle; `size1` is the height and `size2` is the width of the triangle.

Line The location pair and size pair are the endpoints of the line.

Rectangle The location is the top left corner; `size1` is the width and `size2` is the height of the rectangle.

4.2 Non-blocking read

In BACI, input statements are *blocking*, that is, the entire program blocks waiting for the user to input a value. While this is acceptable for entering initial values in the main process, it is not representative of concurrent programs, where an input statement should not block execution at all, or at most, block only the process containing that statement. In the example `alien.pm`, input is used to launch blue and green missiles against a red alien spaceship which moves according to commands within its own process.

In `JBACI`, the standard procedure `read` blocks the entire simulator, while `nbread` is *non-blocking*, that is, an input statement returns immediately after it is called. If the user has entered a value, it is returned; otherwise, a dummy value is returned (-32767 for integer type and '\r' for character type). To wait for input in one process without blocking other processes, you need to write a loop that calls the input statement until a non-flag value is returned. You can use the functions `getNum` and `getChar` in the example programs `ioint.pm(cm)` and `io.pm(cm)`, respectively.

4.3 Monitors

The syntax and semantics of monitors is described in the BACI documentation. As described there, the conditions queues are not FIFO, but priorities can be used to simulate a FIFO queue. According to the immediate resumption requirement, a signaling process is blocked after unblocking a

process waiting on a condition; in JBACI, the interpreter has been modified to assign an artificial priority of -1 to a signaling process so that it will be unblocked before new processes waiting to enter the monitor.

In Pascal, a monitor *must* be declared global to a Pascal main program; the globals window displays the monitor variables:

```
monitor pc;
var ...
procedure Take ...
procedure Append ...
begin ...end;
program ProducerConsumer;
...
```

4.4 Linda

Linda is a concurrent programming paradigm developed by David Gelernter. It is based upon a global *tuple space*: you can read, input and output tuples to the tuple space. If you try to read or input a formal tuple for which no matching actual tuple exists, the process blocks.

The terminology and the operations which are implemented are not standard, but are based upon a course developed at the Weizmann Institute of Science. The implementation is sufficient to demonstrate interesting examples of Linda like load balancing.

The tuple space is defined as a *board*, upon which you can *post*, *remove* or *read* notes. The board is displayed in a separate window as described above.

A *note* consists of a triple: a character and two integer values. The following operations are available:

- `postnote` to post a note on the board.
- `removenote` and `readnote` to remove and read a note from the board. The first parameter must be a character *value*, while the two optional parameters must be integer *variables*. The operations search for an arbitrary note with the character value and return the integer values in the variables. If there is no matching note, the process blocks until a matching note is posted.

The operations can be called with one, two or three parameters; unused integer parameters are given the default value -32767 :

```
postnote('a');    { Equivalent to postnote('a', -32767, -32767) }
postnote('a', 5); { Equivalent to postnote('a', 5, -32767)      }
postnote('a', 5, 10);
```

For historical reasons, there are two syntaxes for matching the values of a tuple to the current values of the parameters:

- If the equal sign = appears after a variable, the value of the note in that position must match the current value of that variable. For example, the following statements will remove any tuple of the form ('a', ..., 5):

```
i2 := 5; removenote('a', i1, i2=);
```

- `removenoteeq` (`readnoteeq`) is like `removenote` (`readnote`), but the two parameters values of the note must match the current values of both variables. For example, the following statements will remove only the tuple ('a', 1, 2):

```
i1 := 1; i2 := 2; removenoteeq('a', i1, i2);
```

Note that `removenoteeq('a', i1, i2)` is equivalent to `removenote('a', i1=, i2=)`.

5 Software structure

JBACI is a single Java package `baci`; the main class `baci.gui.Debugger` is specified in the manifest file so that the software can be executed directly from the jar file. The subpackages are: `event`, `exception`, `graphics`, `gui`, `gui.actionbuttons`, `gui.treetable`, `interpreter`, `program`. Most of my modifications to Strite's code were to the GUI code in `gui`, including the addition of the class `Editor`, based in part on code by Michael Gratton. The addition of the graphics facility led to the new subpackage `graphics`, as well as to the addition of classes in `program` for the new graphics instructions.

To rebuild the IDE and `interpreter`, simply execute `build.bat` from the `\jbaci\baci` directory.

To rebuild the compilers, first copy the files from the subdirectories of `\jbaci\compilers` into the equivalent BACI subdirectories. I do not have a Unix system at my disposal, so the current distribution includes compilers for Windows only. They were created using the MinGW software (<http://www.mingw.org>); batch files for building them can be found in the `compilers` directory.

6 Release notes

- 1.4.5 Alternate syntax for Linda formal parameters.
- 1.4.4 Fix bugs in display of monitor variables and in display of process table when using monitors. Signaling processes are given priority at monitor exit.
- 1.4.3 Fix bug: when writing uninitialized variable, a message is printed instead of throwing an uncaught exception. There are now separate commands `read/cin` for blocking read and `nbread/nbcin` for non-blocking read. User-interface changes: accelerators are defined for all keys in the first three menus; the `Display` button has been removed.
- 1.4.2 Fix bug: postnote in Linda was waking only one blocked process.
- 1.4.1 Editor changed: it is no longer a separate frame; line numbers appear to the left of the text area; if you modify the file, an asterisk is displayed in the title border. Font changed to Java standard Lucida. Font sizes for source, PCode and tables can be changed from the configuration file. The location of the non-blocking input option pane can be set in the configuration file.
- 1.4.0 Linda primitives implemented within the compilers and interpreter.
- 1.3.4 Read in main program is blocking for normal I/O. PCode is not shown by default in process window—you have to click the divider. Windows are tiled, not cascaded; locations and sizes of windows can be set in configuration file. A history file can be written as selected by a configuration file option. History display and file does not display PCode if history of source steps selected.
- 1.3.3 Bugs fixed: exception in display of suspend on condition in monitor, exception if file name contains the substring “line”, mistaken “cross monitor call” in the Pascal compiler and the interpreter, array assignment did not work in Pascal. Modifications: Pascal compiler allows implicit cast between `integer` and `char`; C compiler allows explicit casts `int(c)` and `char(i)`. Additions: Linda tuple space implemented as BACI source modules.
- 1.3.2 Bugs fixed: cannot select window of main process, run without compile caused exception. For Pascal programs, Globals windows (redundantly) shows the variables of main program. History window shows source code lines and an option has been added to select whether to update the window after each source or PCode step. Default file extensions remain `pm` and `cm`, but other extensions are allowed. Non-blocking read. Compatibility with previous simulator: `parbegin/parend` in addition to `cobegin/coend`; `process` in addition to `procedure`; `init(1)` syntax for semaphore initialization in Pascal.

- 1.3.1** Pascal compiler modifications applied to version which enables Boolean initializers and nested procedures.
- 1.3** Configuration file is now a Java properties file; clicking on process table entry or process window selects current process.
- 1.2** Initial public release.