USER'S GUIDE
BACI C−− Compiler and Concurrent PCODE Interpreter

Bill Bynum/Tracy Camp
College of William and Mary/Colorado School of Mines

July 14, 2003

Contents

1 Introduction

The purpose of this document is to provide a brief description of the C−− BACI Compiler and Concurrent PCODE Interpreter programs and a description of how to use them. The C−− compiler first compiles the user's program into an intermediate object code called PCODE, which the interpreter then executes. The C−− compiler supports binary and counting semaphores and Hoare monitors. The interpreter simulates concurrent process execution.

Programs of the BACI System

program	function	described in
`bacc`	BACI C−− to PCODE Compiler	this guide (cmimi.ps)
`bapas`	BACI Pascal to PCODE Compiler	guidepas.ps
`bainterp`	command-line PCODE Interpreter	cmimi.ps, guidepas.ps disasm.ps
`bagui`	Graphical user interface to the PCODE Interpreter (UNIX systems only)	guiguide.ps
`badis`	PCODE de-compiler	disasm.ps
`baar`	PCODE archiver	sepcomp.ps
`bald`	PCODE linker	sepcomp.ps

The Pascal version of the compiler and the interpreter were originally procedures in a program written by M. Ben-Ari, based on the original Pascal compiler by Niklaus Wirth. The program source was included as an appendix in Ben-Ari's book, "Principles of Concurrent Programming". The original version of the BACI compiler and interpreter was created from that source code. Eventually, the Pascal compiler and interpreter were split into two separate programs. and the C−− compiler was developed to compile source programs written in a restricted subset of C++ into PCODE executable by the interpreter.

The syntax for the C−− compiler is explained below. This guide is applicable only to the C−− compiler and not to the BACI Concurrent Pascal compiler. Users interested in the Pascal compiler should consult its user guide (see the file **guidepas.ps**).

2 C−− Compiler Syntax

1. As in C++, comments can be delimited with '`/*`' and '`*/`' or '`//`'.

2. There are no files other than standard input and output: `cout`, `cin`, and `endl` behave in C−− BACI as in standard C++. The main program must have one of the following forms:

```
int main ()
void main ()
  main ()
```

3. The only simple C/C++ types available in C−− BACI are `int`, and `char`. There are also other types related to the control of concurrency; these will be discussed below.

   All variables must be declared at the beginning of the code block in which they appear. In particular, the index variable of a `for` loop cannot be declared in the loop header, but instead must be declared at the beginning of the block containing the `for` loop.

4. A `string` type is supported. To declare a string, the length of the string must be specified. The following declaration defines a string of length 20:

```
string[20] string_name;
```

The length specifier should be the number of characters that the string should have and should not include space for the termination byte. The compiler takes care of reserving space for the termination byte. The length specifier must be either a literal constant or a program constant.

The `string` keyword is used in the declaration of a function for declaring a parameter of type `string`:

```
void proc(string formal_parm)
```

This declaration asserts that `formal_parm` is of type `string[n]`, for some positive value of `n`. Parameters of type `string` are passed by reference. No check for string overrun is performed.

5. Arrays of any valid type are supported. Array declaration follows the usual C syntax:

```
element_type arrayname[index1][index2][index3]...[indexN];
```

6. The keyword `typedef` is supported in C−− BACI. For example, to make the variable name `length` synonym with `int`

```
typedef int length;
```

7. Constants (`const`) of simple types are supported:

```
const int m = 5;
```

8. In the declaration of variables of `int` and `char` types, initializers are supported. The value of the initializer must be a literal or program constant:

```
const int m = 5;
int  j = m;
int  k = 3;
char c = 'a';
```

9. Procedures and functions are supported. Standard scope rules apply. Recursion is supported. Parameter declarations are pass-by-value or pass-by reference:

```
int afunc( int   a,     /* pass-by-value  */
           int&  b  )   /* pass-by-reference */
```

Each program must have a `main()` function of type `int` or `void`, and this function must be the last function in the source file. Execution begins with a call to `main()`.

10. The executable statements are `if-else`, `switch`/`case`, `for`, `while`, `do-while`, `break`, and `continue`. The syntax of these statements are as in standard C/C++. Bracketing of code is standard, i.e., `{ ... }`.

11. Standard C/C++ file inclusion is supported:

```
#include < ... >
#include " ... "
```

Both styles of `include` statement have the same semantics, because there is no "system" include directory.

12. The `extern` keyword for defining external variables is supported. An external variable can be of any valid C−− type. Initializers cannot be used with external variables. The `extern` keyword can only occur at the global ("outer") level. Typical examples:

```
      extern int i;
      extern char a[20];
      extern string[30] b;
//  Initializers are not allowed ----> extern int i = 30;
//  (initialization, if present, must occur where i is defined)
      extern int func( int k );
      extern monitor monSemaphore { // see Section 3.  Only externally
         void monP();               // visible details of the monitor
         void monV();               // need be given here
      }
```

The `-c` option must be used with `bacc` to compile source files that contain external references. See the *BACI System Separate Compilation Guide* for more information about the use of external variables.

3 Concurrency Constructs

3.1 `cobegin` Block

A C−− process is a `void` function. In the BACI system, the term "concurrent process" is synonymous with the term "concurrent thread." A list of processes to be run concurrently is enclosed in a `cobegin` block. Such blocks cannot be nested and must appear in the main program.

```
cobegin {
     proc1(...); proc2(...); ... ; procN(...);
}
```

The PCODE statements belonging to the listed procedures are interleaved by the interpreter in an arbitrary, 'random' order, so that multiple executions of the same program containing a `cobegin` block will appear to be non-deterministic. The main program is suspended until all of the processes in the `cobegin` block terminate, at which time execution of the main program resumes at the statement following the ending of the block.

3.2 Semaphores

The interpreter has a predeclared `semaphore` type. That is, a semaphore in C−− is a non-negative-valued `int` variable (see definition below) that can be accessed only in restricted ways. The binary semaphore, one that only assumes the values `0` and `1`, is supported by the `binarysem` subtype of the `semaphore` type. During compilation and execution, the compiler and interpreter enforce the restrictions that a `binarysem` variable can only have the values `0` or `1` and that `semaphore` type can only be non-negative.

3.2.1 Initializing a Semaphore

Assignment to a `semaphore` or `binarysem` variable is allowed only when the variable is defined. For example, either of the following declarations is valid

```
semaphore  s = 17;
binarysem  b = 0;
```

The built-in procedure,

```
initialsem( semaphore, integer_expression );
```

is the only method available for initializing a semaphore of either type at runtime. In the call,

```
integer_expression
```

can be any expression that evaluates to an integer and is valid for the semaphore type (non- negative for a `semaphore` type, `0` or `1` for a `binarysem` type). For example, the following two `initialsem` calls show an alternative way to initialize the two semaphores declared above:

```
initialsem( s, 17);
initialsem( b, 0);
```

3.2.2 `p` (or `wait`) and `v` (or `signal`) Functions

The `p` function (or synonymously, `wait`) and the `v` function (or synonymously, `signal`) are used by concurrently executing processes to synchronize their actions. These functions provide the user with the only way to change a semaphore's value.

The prototypes of the two functions are as follows:

```
void p( semaphore& s );
```

or equivalently,

```
void wait( semaphore& s );
```

and

```
void v( semaphore& s );
```

or equivalently,

```
void signal( semaphore& s );
```

The `semaphore` argument of each function is shown as a reference parameter, because the function modifies the value of the `semaphore`.

The semantics of the `p` and `v` function calls are as follows:

```
p( sem );
```

If `sem` > 0, then decrement `sem` by 1 and return, allowing `p`'s caller to continue.
If `sem` = 0, then put `p`'s caller to sleep. These actions are **atomic**, in that they are non-interruptible and execute from start to finish.

```
v( sem );
```

If `sem` = 0 and one or more processes are sleeping on `sem`, then awake one of these processes. If no processes are waiting on `sem`, then increment `sem` by one. In any event, `v`'s caller is allowed to continue. These actions are **atomic**, in that they are non-interruptible and execute from start to finish.

Some implementations of `v` require that processes waiting on a semaphore be awakened in FIFO order (queuing semaphores), but BACI conforms to Dijkstra's original proposal by randomly choosing which process to re-awaken when a signal arrives.

3.2.3 Examples of Semaphore Usage

To help to explain semaphore usage, we offer the following brief example:

```
BACI System: C-- to PCODE Compiler, 09:24   2 May 2002
Source file: semexample.cm  Sun Apr 28 20:40:12 2002
 line  pc
   1    0  // example of C-- semaphore usage
   2    0
   3    0  semaphore count;     // a "general" semaphore
   4    0  binarysem output;    // a binary (0 or 1) semaphore for unscrambling output
   5    0
   6    0  void increment()
   7    0  {
   8    0     p(output);        // obtain exclusive access to standard output
   9    2     cout << "before v(count) value of count is " << count << endl;
  10    6     v(output);
  11    8     v(count);         // increment the semaphore
  12   10  }  // increment
  13   11
  14   11  void decrement()
  15   11  {
  16   11     p(output);        // obtain exclusive access to standard output
  17   13     cout << "before p(count) value of count is " << count << endl;
  18   17     v(output);
  19   19     p(count);         // decrement the semaphore (or stop -- see manual text)
  20   21  }  // decrement
  21   22
  22   22  main()
  23   23  {
  24   23     initialsem(count,0);
  25   26     initialsem(output,1);
  26   29     cobegin {
  27   30        decrement(); increment();
  28   36     }
  29   37  }  // main
```

The program uses two semaphores. One semaphore, `count`, is of `semaphore` type, which indicates to the BACI system that the semaphore will be allowed to have any non-negative value. The two concurrent procedures, `increment` and `decrement`, "signal" each other through the `count` semaphore. The other semaphore, `output`, is of `binarysem` type, which indicates to the BACI system that the semaphore should always have the value zero or one; any other value causes a run-time exception. This semaphore is used to keep the output from the two concurrently executing procedures, `increment` and `decrement` from intermingling.

We produced the above compiler listing with the command

```
prompt% bacc semexample
Pcode and tables are stored in semexample.pco
Compilation listing is stored in semexample.lst
```

The `semexample.pco` file can then be executed with the BACI PCODE interpreter:

```
prompt% bainterp semexample
Source file: semexample.cm  Sun Apr 28 20:40:12 2002
Executing PCODE ...
before v(count) value of count is 0
before p(count) value of count is 1
```

This execution of the PCODE file is one of the three possible outputs that the program can produce. The other two possible program outputs are

```
prompt% bainterp semexample
Source file: semexample.cm  Sun Apr 28 20:40:12 2002
Executing PCODE ...
before p(count) value of count is 0
before v(count) value of count is 0

prompt% bainterp semexample
Source file: semexample.cm  Sun Apr 28 20:40:12 2002
Executing PCODE ...
before v(count) value of count is 0
before p(count) value of count is 0
```

An interested reader might find it instructive to supply explanations for ways in which these three program outputs are generated and to show that these three outputs are the only outputs possible.

3.3 Monitors

The monitor concept, as proposed by Hoare, is supported with some restrictions. A `monitor` is a C−− block, like a block defined by a procedure or function, with some additional properties. All functions in the monitor block are visible (that is, are callable entry procedures) from the outside of the block, but the monitor variables are not accessible outside of the block and can only be accessed by the monitor functions. In C−−, a monitor can be declared only at the outermost, global level. Monitors can not be nested. A monitor can have an optional `init{}` block as its last block for initializing the values of the monitor's variables. This code is run when the main program is started.

Only one procedure or function of the monitor block can execute at any one time. This feature makes it possible to use monitors to implement mutual exclusion. Use of monitors to control concurrency is advantageous, because all of the code controlling concurrency is located in the monitor and not distributed widely across callers, as is the case when semaphores are used.

Three constructs are used by the procedures and functions of a monitor to control concurrency: `condition` variables, `waitc` (wait on a condition), and `signalc` (signal a condition).

3.3.1 Condition Variables

A `condition` variable can only be defined in a monitor, and thus, can only be accessed by the monitor's processes. A `condition` variable never actually 'has' a value; it is somewhere to wait or something to signal. A monitor process can wait for a condition to hold or signal that a given condition now holds through the `waitc` and `signalc` calls.

3.3.2 `waitc` and `signalc` Functions

`waitc` and `signalc` calls have the following syntax and semantics:

```
void waitc( condition cond, int prio );
```

The monitor process (and hence, also the outside process calling the monitor process) is blocked and assigned the priority `prio` for being re-awakened (see `signalc` below). Note that this blocking action allows some other monitor process to execute, if one is ready.

```
void waitc( condition cond );
```

This call has the same semantics as the `waitc` call above, but the wait is assigned a default priority of 10.

```
void signalc( condition cond );
```

Wake some process waiting on `cond` with the smallest (highest) priority; otherwise, do nothing. Note that this is quite unlike the semaphore `v` or `signal`, because `signalc` is a no-op if no one is waiting,

whereas `v( sem )` increments `sem` if no one is waiting, thus "remembering" the action when future `p( sem )`'s occur.

The priority scheme can be used to implement a FIFO discipline in re-awakening waiters. If each monitor process increments a monitor variable associated with the current priority assigned to a condition, then successive `signalc`'s to the condition will awaken the sleeping processes in a FIFO order.

The C−− compiler provides an `int` function `empty( cond )` that returns `1` if there are no processes waiting in the queue of the condition `cond` and `0` otherwise.

3.3.3 Immediate Resumption Requirement

This is the requirement that a process waiting on a condition that has just been signaled should have priority in re-entering the monitor over new calls to monitor processes (those wanting to enter "from the top"). The requirement rests on the assumption that the condition that has just been signaled has more "urgent" business to perform than a new entry into the monitor. The Immediate Resumption Requirement is implemented in BACI by suspending the signaller of a condition and picking (at random) one of the waiters on the condition with the appropriate priority to run. Because of this, monitor procedures that `signalc` a condition typically do so as their last instruction.

When the process re-awakened by the `signalc` leaves the monitor, a process executing in the monitor that has been suspended after issuing a `signalc` call is allowed to resume execution in the monitor in preference to processes attempting to enter the monitor "from the top."

3.3.4 An Example of a Monitor

The following example of an implementation of a general semaphore with a monitor illustrates the monitor syntax:

```
monitor monSemaphore {

   int semvalue;
   condition notbusy;

   void monP()
   {
      if (semvalue == 0)
         waitc(notbusy);
      else
         semvalue--;
   }

   void monV()
   {
      if (empty(notbusy))
         semvalue++;
      else
         signalc(notbusy);
   }

   init{ semvalue = 1; }

}  // end of monSemaphore monitor
```

3.4 Other Concurrency Constructs

BACI C−− provides several low-level concurrency constructs that can be used to create new concurrency control primitives: These functions can be used to create a "fair" (FIFO) queued semaphore. The code to accomplish this is beyond the scope of this user's guide.

3.4.1 `atomic` Keyword

If a function is defined as `atomic`, then the function is *non-preemptible*. The interpreter will not interrupt an `atomic` function with a context switch. This provides the user with a method for defining new primitives. The following program illustrates how a `test_and_set` primitive can be defined and used to enforce mutual exclusion:

```
atomic int test_and_set(int& target) {
   int u;
   u = target;
   target = 1;
   return u;
}

int lock = 0;

void proc(int id) {
   int i = 0;
   while(i < 10) {
      while (test_and_set(lock)) /* wait */ ;
      cout << id;
      lock = 0;
      i++;
   }
}

main() {
   cobegin { proc(1); proc(2); proc(3); }
}
```

3.4.2 `void suspend( void );`

The `suspend` functions puts the calling thread to sleep.

3.4.3 `void revive( int process_number );`

The `revive` function revives the process with the given number.

3.4.4 `int which_proc( void );`

The `which_proc` function returns the process number of the current thread.

3.4.5 `int random( int range );`

The `random` function returns a "randomly chosen" integer between `0` and `range - 1`, inclusive. It uses a different random number generator stream than the one used by the interpreter; that is, `random()` calls do not affect interpreter execution.

4 Built-in String Handling Functions

4.1 `void stringCopy(string dest, string src);`

The `stringCopy` function copies the `src` string into the `dest` string. No check for string overrun is performed. For example,

```
string[20] x;
   ...
stringCopy(x,"Hello, world!");
stringCopy(x,"");
```

will initialize the string x to a well-known value. The second stringCopy resets the string x to a zero-length string.

4.2 void stringConcat(string dest, string src);

The stringConcat function concatenates the src string to the end of the dest. No check for string overrun is performed.

4.3 int stringCompare(string x, string y);

The stringCompare function has the same semantics as the strcmp function from the C string library: a positive number is returned if string x is lexicographically after the string y, zero is returned if the strings are equal, and a negative number is returned if string x is lexicographically before the string y.

4.4 int stringLength(string x);

The stringLength function returns the length of the string x, not including the termination byte.

4.5 int sscanf(string x, rawstring fmt,...);

Like the "real" sscanf, the sscanf function scans the string x according to the format string fmt, storing the values scanned into the variables supplied in the parameter list, and returns the number of items scanned. Only the %d, %x, and %s format specifiers of the real sscanf are supported. An additional format specifier %q (quoted string), unique to BACI, is supported. For this specifier, all characters delimited by a pair of double quotes (") will be scanned into the corresponding string variable. When the %q specifier is encountered in the format string, if the next non-whitespace character of the string being scanned is not a double quote, then the %q scan fails, and scanning of the string terminates.

The variables appearing after the format string are reference variables (that is, the ampersand (&) is not required.

In the following example, the value of i returned by the sscanf call will be 4, the value stored in the variable j will be 202, the value stored in the string stored in string x will be alongstring, the value stored in the variable k will be 0x3c03, and the string stored in the string y will be a long string.

```
string[50] x,y;
int i,j,k;
stringCopy(x,"202 alongstring 3c03 \"a long string\"");
i = sscanf(x,"%d %s %x %q",j,x,k,y);
```

4.6 void sprintf(string x, rawstring fmt,...);

Like the "real" sprintf function in the C library, the sprintf function creates a string stored in the variable x, using the format string fmt and the variables following the format string.

The %d, %o, %x, %X, %c, and %s format specifiers are supported, in the full generality of the real sprintf. In addition, the %q format specifier will insert a doubly-quoted string into the output string. The %q format specifier is equivalent to the \"%s\" specifier.

For example, in the following code fragment

```
string[80] x;
string[15] y,z;

stringCopy(y,"alongstring");
stringCopy(z,"a long string");
sprintf(x,".%12d. .%-20s. .%q. .%08X.",202,y,z,0x3c03);
```

the string `x` becomes

```
.         202. .alongstring         . ."a long string". .00003C03.
```

5 Using the BACI C−− Compiler and PCODE Interpreter

There are two steps to executing a program with the BACI system.

1. Compile a "`.cm`" file to obtain a "`.pco`" file.
   Usage: `bacc [optional_flags]  source_filename`
   Optional flags:

   ```
   -h  show this help
   -c  make a .pob object file for subsequent linking
   ```

   The name of the source file is required. If missing, you will be prompted for it. The file suffix "`.cm`" will be appended if you don't supply it.

2. Interpret a "`.pco`" file to execute the program
   Usage: `baininterp [optional_flags]  pcode_filename`
   Optional flags:

   ```
   -d  enter the debugger, single step, set breakpoints
   -e  show the activation record (AR) on entry to each process
   -x  show the AR on exit from each process
   -t  announce process termination
   -h  show this help
   -p  show PCODE instructions as they are executed
   ```

   The name of the PCODE file is required. If missing, you will be prompted for it. The file suffix "`.pco`" will be appended to the filename you give.

It is not necessary to recompile the source file each time that you execute the `.pco` file with `bainterp`.

There is a shell script, `baccint`, that will call the compiler and then call the interpreter for you. It passes the options that you give it (see above) along to the interpreter.

6 Sample program and output

The following listing was produced by the C −− BACI compiler. The number to the right of the line number is the PCODE offset of the instruction that begins that line. The BACI compiler creates this listing from the file "`incremen.cm`". The listing is placed in the file "`incremen.lst`". An "`incremen.pco`" file is also created; this file is used by the interpreter.

```
BACI System: C-- to PCODE Compiler, 09:24   2 May 2002
Source file: incremen.cm  Wed Oct 22 21:18:02 1997
 line  pc
   1    0  const int m = 5;
   2    0  int n;
   3    0
   4    0  void incr(char id)
   5    0  {
   6    0     int i;
   7    0
   8    0     for(i = 1; i <= m; i = i + 1)
   9   14     {
  10   14         n = n + 1;
  11   19         cout << id << "  n =" <<  n << "  i =";
  12   25         cout << i << " " << id << endl;
  13   31     }
  14   32  }
  15   33
  16   33  main()
  17   34  {
  18   34     n = 0;
  19   37     cobegin
  20   38     {
  21   38        incr( 'A'); incr( 'B' ); incr('C');
  22   50     }
  23   51     cout << "The sum is " <<  n << endl;
  24   55  }
```

The following listing was produced by the BACI interpreter. The interpreter executes the program that was compiled into the file "`incremen.pco`".

```
Source file: incremen.cm  Wed Oct 22 21:18:02 1997
Executing PCODE ...
C  n =1  i =A  n =1 C2  i =
1 A
C  n =4  i =2 C
B  n =A  n =5  i =24 A
  i =1 B
AC  n =  n =6  i =3 C6  i =3
A
C  n =7  i =4 C
B  n =9  i =2 BA  n =8
  i =4 A
C  n =8  i =5 A  n =9C
  i =5 A
B  n =10  i =3 B
B  n =11  i =4 B
B  n =12  i =5 B
The sum is 12
```