ConEE Tutorial

ConEE is a simulator designed to assist in writing and debugging simple multithreaded programs, for the purpose of examining common concepts and issues involved in concurrent programming.

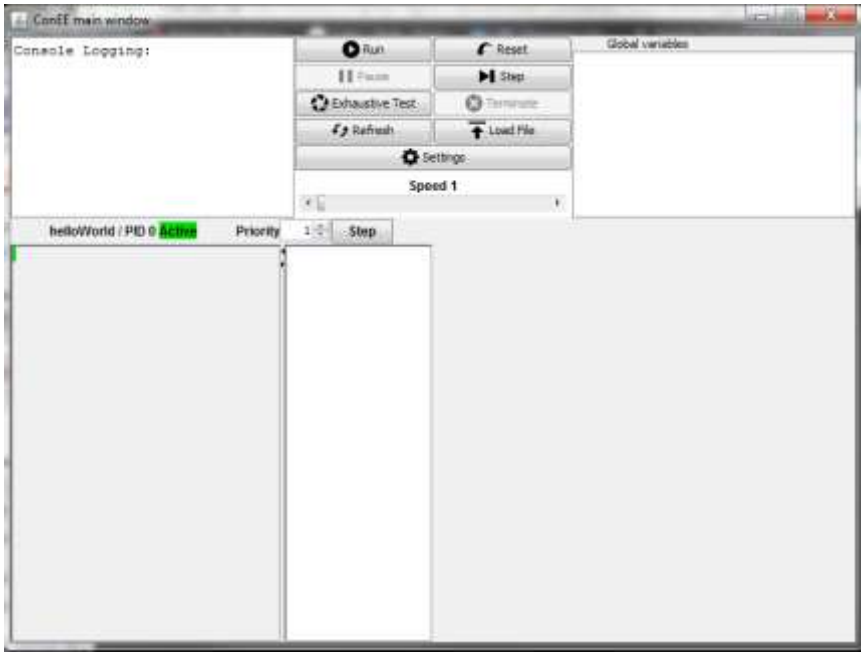
Part One: Hello World
All that is required to write programs for ConEE is a text editor. So find one and write:

```
Thread helloWorld{

}
```

This is a thread block header. Block headers are followed by a { and a } on another line. They are ConEE's method of organising code and variables.

You now have everything necessary to open the program in ConEE. First, save your file as a .txt. Next, find your ConEE.jar and double click it. Click the "Load File" button, and navigate to your file. Open your file. You should see this:



Now let's make it actually do something.
Inside the thread block, add a Code block. Be sure to capitalize the C in Code. And inside the Code block, add a print statement:

```
Thread helloWorld{
      Code{
            put_line("Hello World");
      }
}
```

Now save your file, hit the "refresh" button, and then press either one of the "step" buttons.

If you click the “reset” button, the simulator resets and you can try using the other step button.

Now let’s add another print statement, and then refresh and use the step buttons again to follow the thread’s execution.

```
Thread helloWorld{
      Code{
            put_line(“Hello World”);
            put_line(“Goodbye World”);
      }
}
```

Now let’s start trying out this concurrency idea. First, let’s copy our thread a few times by modifying our hello world code as follows:

```
Thread(3) helloWorld{
      Code{
            put_line(“Hello World”);
            put_line(“Goodbye World”);
      }
}
```

You can create as many threads as you like by putting any number inside the brackets, but once you start getting more than 10 they start getting a bit hard to see.
Save your file, refresh, and use the run button to watch the different ways the 3 threads can execute. Reset and then use the run, step and reset buttons to execute the code in a different order.

Part 2: Enter the Variable!

As much fun as it is to say hello to the world (endlessly), let's try something more interesting. Add a new block type called GlobalVariables. This block will contain all the variables that you wish to be accessed by all threads. Let's start by adding an integer.

```
Thread(3) helloWorld{
      Code{
            put_line("Hello World");
            put_line("Goodbye World");
      }
}
GlobalVariables{
      Integer int(0);
}
```

Note the placement of the global variables is outside of the Thread contents. Inside the brackets of the variable, you place the initialization value. Again, it can be any number. How about we add a Boolean and a String while we are at it.

```
Thread(3) helloWorld{
      Code{
            put_line("Hello World");
            put_line("Goodbye World");
      }
}
GlobalVariables{
      Integer int(0);
      Boolean boo(false);
      String str("I am the string");
}
```

Let's put those variables to work. Let's make each thread increment the Integer, and when the integer equals 2, we'll print the string. To increment we need to add the following assignment statement.

```
Thread(3) helloWorld{
      Code{
            put_line("Hello World");
            g->int:=g->int+1;
            put_line("Goodbye World");
      }
}
GlobalVariables{
      Integer int(0);
      Boolean boo(false);
      String str("I am the string");
}
```

Note the interesting assignment statement syntax. The `g->` tells the program to look for the variable in the global variables, rather than in the threads local variables.

It has a counterpart `d->` which tells the program to look for the variable in the local variables. Remember to add the : in front of the =, otherwise the program tries to compare the two.

Save your file, refresh ConEE, and see how it runs.

Now we need an if statement.

```
Thread(3) helloWorld{
      Code{
            put_line("Hello World");
            g->int:=g->int+1;
            if(g->int=2)then
                  put_line(g->str);
            end if;
            put_line("Goodbye World");
      }
}
GlobalVariables{
      Integer int(0);
      Boolean boo(false);
      String str("I am the string");
}
```

All if statements must end with an `end if;` regardless of whether they have an `elseif` or an `else`. More information on if statements can be found in the ConEE Language guide.

So now we have a program that prints the string once. Or does it? Save, refresh and run your program. **Can you find a way to make the program run so that it never prints the string at all?**

The problem comes in where one thread increments the Integer, and then another comes in and increments it as well, before the first thread has checked if it's equal to 2. Welcome to the world of concurrent programming! Run the program one way and it does one thing, run it another and it does something else.

Part 3: Enter the Semaphore!

If you don't know what a Semaphore is, go read the Little Book of Semaphores up to and including Section 3.3. Seriously, go read it. Don't be a slacker.

If you understood what the book said, you should be able to detect that the problem with the code we've written so far is a **mutual exclusion problem**. Another thread changes a variable before the first thread has finished with it. So what we need to fix this problem is a *Mutex* (if you don't know what that is, go back and *actually* read it this time!). Let's add one to our GlobalVariables.

```
Thread(3) helloWorld{
      Code{
            put_line("Hello World");
            g->int:=g->int+1;
            if(g->int=2)then
                  put_line(g->str);
            end if;
            put_line("Goodbye World");
      }
}
GlobalVariables{
      Integer int(0);
      Boolean boo(false);
      String str("I am the string");

      Semaphore sem(1);

}
```

Since you've actually read the book this time, you know why it's initialised to 1. The Semaphore commands that ConEE uses are `wait(g->sem);` and `signal(g->sem);` Where do you think we should put the wait and signal in the program? How about:

```
Thread(3) helloWorld{
      Code{
            put_line("Hello World");
            wait(g->sem);
            g->int:=g->int+1;
            if(g->int=2)then
                  put_line(g->str);
            end if;
            signal(g->sem);
            put_line("Goodbye World");
      }
}
GlobalVariables{
      Integer int(0);
      Boolean boo(false);
      String str("I am the string");
      Semaphore sem(1);
}
```

Save, refresh, and let's go see if it works.

Part 4: Testing

Well, seems to be working. The string prints exactly one. But how can we be sure that it always prints exactly once? Well, one way is to try out every single possible combination of the threads. Doesn't that sound like fun? Thankfully, ConEE has a system to do that.

There are two things we want to be sure of when testing this code. First, that the Mutex is working, and no more than one thread is accessing the integer at any given time. Second, when the integer is equal to 2, the string is printed.

This sort of testing is extremely difficult, because it's hard for the computer to know exactly when to check which condition. This is where programmers must be smart. ConEE provides a separate set of variables to do this sort of testing. These variables are declared in an AcceptanceTests block.

For this program, we will want two of the three kinds of AcceptanceTest variables, a CriticalSection, and a Serialization. The CriticalSection should keep track of how many of each kind of thread has entered the critical section, and the Serialization keeps track of how many times a certain line has executed. So let's create our acceptance test variables.

```
Thread(3) helloWorld{
      Code{
            …
      }
}
GlobalVariables{
      …
}
AcceptanceTests{
      CriticalSection critSec;
      Serialization seri;
}
```

Acceptance test variables do not require any initialization. Now there are two checks we need to put in. The first is a check to see if a helloWorld thread is in the critical section, check that it is the only helloWorld thread in that section.

```
Thread(3) helloWorld{
      Code{
            put_line("Hello World");
            wait(g->sem);
            g->int:=g->int+1; \check(1, helloWorld, critSec)\
            if(g->int=2)then
                  put_line(g->str);
            end if;
            signal(g->sem);
            put_line("Goodbye World");
      }
}
```

An acceptance test line will run when the last line before it does. So when the line `g->int:=g->int+1;` runs, the computer does a check that there is 1 or less helloWorld threads in critSec. But how does it know what is in the critSec? We have to tell it by adding the following:

```
Thread(3) helloWorld{
      Code{
            put_line("Hello World");
            wait(g->sem);
            g->int:=g->int+1; \critSec entered\
            \check(1, helloWorld, critSec)\
            if(g->int=2)then
                  put_line(g->str);
            end if;
            signal(g->sem); \critSec exited\
            put_line("Goodbye World");
      }
}
```

Now the program knows when a thread has entered the critical section and when it has exited the critical section. But why is `\critSec entered\` behind `g->int:=g->int+1;` instead of `wait(g->sem);` ?

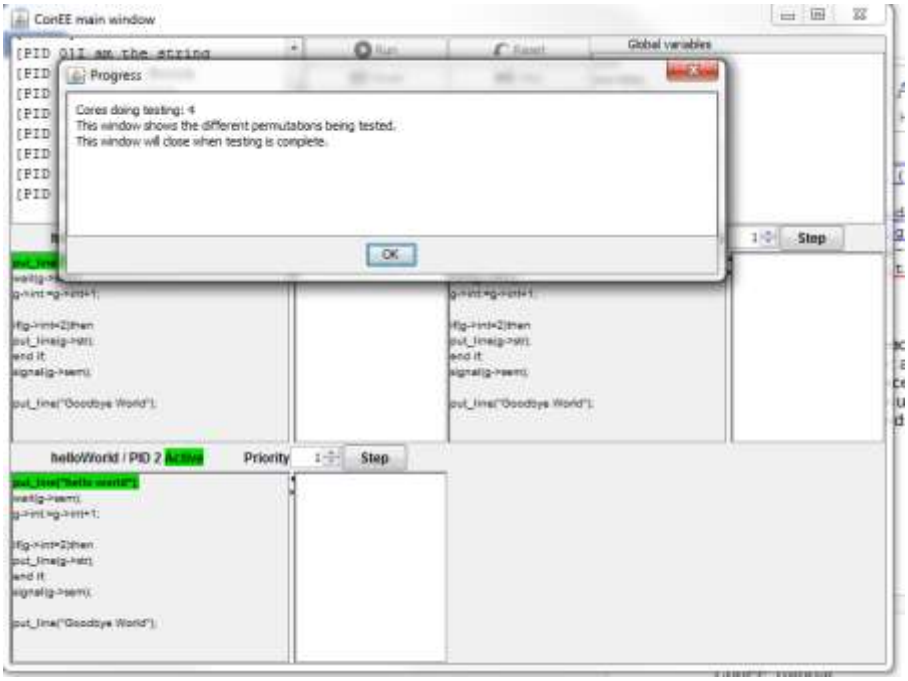
Think about this, what happens if the thread executes `wait(g->sem);` and then is waiting? Is it in the critical section? We can only be sure that a thread is in the critical section when it executes the first line in the critical section.
Note: you can have multiple acceptance test checks on the same line.
Now let's do the serialization.

```
Thread(3) helloWorld{
      Code{
            put_line("Hello World");
            wait(g->sem);
            g->int:=g->int+1; \critSec entered\
            \check(1, helloWorld, critSec)\
            if(g->int=2)then
                  put_line(g->str); \seri executed\
            end if;
            signal(g->sem); \critSec exited\
            \g->int<2 OR check(1, seri)\
            put_line("Goodbye World");
      }
}
```

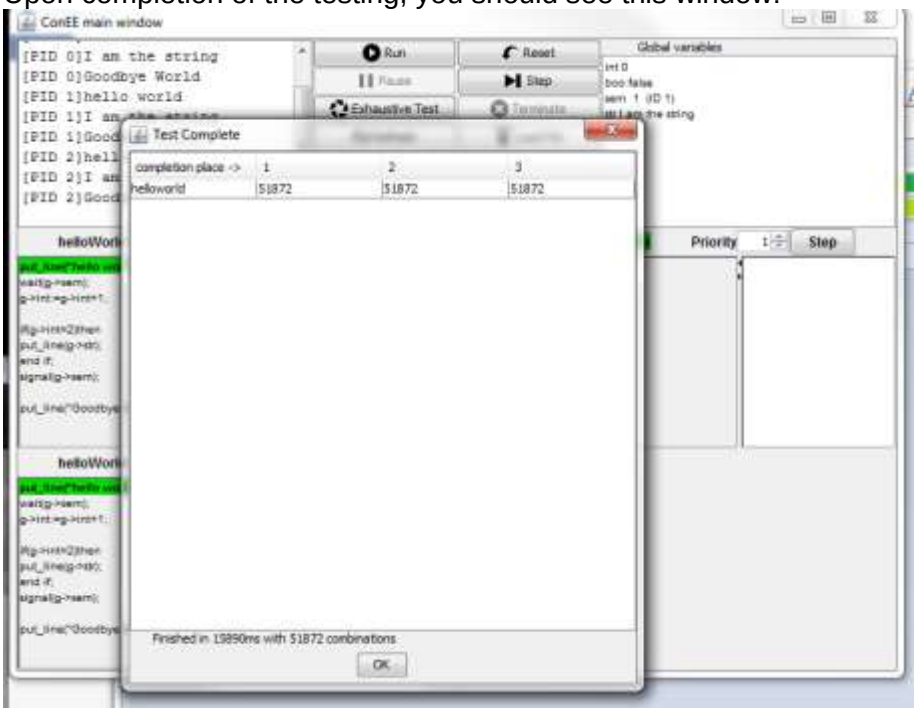
Wait, what? Yes. You read that correctly. `g->int<2 OR check(1, seri)`
Acceptance test lines act a bit like if statements. Anything you can put into an if statement you can put into an acceptance test check. In this case, integers must be less than two, but if it's not then seri must have executed at least once. The simulator evaluates the statement and if the result is false, then it sends out an alert that one of your acceptance tests has failed.

These tests you can be pretty sure won't fail, but if you want to see what it looks like when they do, just take out the `\seri executed\` and run it. More information on Acceptance Tests and Boolean logic is available in the ConEE Language guide.

Now we are ready to test it. Press the Exhaustive Test button. Now wait. This'll take a couple seconds. Usually about 30 or so, during which time you will see:



Upon completion of the testing, you should see this window:



The completion popup tells you that after running every single possible combination of the lines of code, it found no errors. The completion place table tell you in which order the threads completed. This isn't very helpful right now, as this program only has one kind of thread, but it can become useful later, when you start running into issue of starvation.

In this case, this window also tells you that it ran the program 51872 times in order to try out all the possible combinations. The number of combinations increases exponentially with the number of threads and the number of lines. If you want to use the Exhaustive Test to full effect it is important to be frugal with your code lines.

ConEE Tutorial

ConEE Advanced Tutorial

This tutorial will be focused on a couple problems from the Little Book of Semaphores, and will cover some extra syntaxes needed to solve them.

Part 1: Rendezvous

> *Section 3.5: Barrier Problem*
> *For convenience, I will copy the text here as it's short.*
>
> *Consider again the Rendezvous problem from Section 3.2. A limitation of the solution we presented is that it does not work with more than two threads.*
>
> *Puzzle: Generalize the rendezvous solution. Every thread should run the following code:*
> *1 rendezvous*
> *2 critical point*
> *The synchronization requirement is that no thread executes critical point until after all threads have executed rendezvous. You can assume that there are n threads and that this value is stored in a variable, n, that is accessible from all threads. When the first n − 1 threads arrive they should block until the nth thread arrives, at which point all the threads may proceed.*
>
> *-The Little Book of Semaphores, Allen B. Downey, version 2.1.5, pages 21-22*

To solve this problem, you will need assignment statements, if statements, and semaphore commands, all of which were covered in the previous tutorial.

For indicating the critical point you can either put in a comment (indicated by a -- at the start of the line) or add a print statement. The advantage to using a comment is that you have fewer lines to test later. However, as at this point, size will not be a huge issue, it's probably better to use a print statement, as that gives you a more natural place to run acceptance tests.

Now, for running acceptance tests to test a barrier you will need to use the last kind of acceptance test variable, the Rendezvous.

```
AcceptanceTests{
Rendezvous ren;
}
Thread(3) yourThreadName{
      Code{
      --your barrier code
      put_line("arrived at rendezvous"); \ren arrived\
      --more barrier code
      put_line("The critical point"); \check(3, yourThreadName, ren)\
      --rest of your barrier code
      }
}
```

The Rendezvous ‘check’ syntax is identical to the CriticalSection ‘check’ syntax, but they have different functions. While the Critical Section checks that there are no more than x threads in the critical section, the rendezvous checks that at least x threads have executed arrived.

Code with Rendezvous in it are particularly tricky to test, due to their unsymmetrical nature. The last thread that arrives at the barrier must execute a different chunk of code then all the preceding threads. You may find that a single Rendezvous is insufficient for testing the code, and you may need to add a Serialization or an extra variable to test it properly.

Part 2: Multiple thread types

Up to now, the problems covered in this tutorial have had only one kind of thread. For this next problem, you will need more than one kind of thread.

> *Section 3.7: Queue*
>
> *Semaphores can also be used to represent a queue. In this case, the initial value is 0, and usually the code is written so that it is not possible to signal unless there is a thread waiting, so the value of the semaphore is never positive. For example, imagine that threads represent ballroom dancers and that two kinds of dancers, leaders and followers, wait in two queues before entering the dance floor. When a leader arrives, it checks to see if there is a follower waiting. If so, they can both proceed. Otherwise it waits.*
>
> *Similarly, when a follower arrives, it checks for a leader and either proceeds or waits, accordingly. Puzzle: write code for leaders and followers that enforces these constraints.*
>
> *-The Little Book of Semaphores, Allen B. Downey, version 2.1.5, page 45*

For this problem we need two kinds of threads, leader and follower. To create more than one thread type, simply add multiple thread blocks.

```
Thread leader{
      Code{
            --leader code
      }
}

Thread follower{
      Code{
            --follower code
      }
}
```

You can have any number of thread blocks, provided they all have different names.
Write the code for leaders and followers and test it.

ConEE Tutorial

Part 3: Arrays

We are now going to skip several problems[1] and move on to Section 4.4 Dining Philosophers. Go read the Little Book of Semaphores section 4.4, and come back when you reach the line that says “Puzzle: what’s wrong?”

Done?

As you can see from the solution given, Dining Philosophers classically uses arrays to solve the problem. While ConEE does provide a fully functional array syntax, it (tragically) does not provide any function definition syntax as of yet.

Here we present the ConEE version of listings 4.29-4.31 in the text:

```
--BasicPhilosophers
GlobalVariables{
      --listing 4.30
      Array(Semaphore) fork(5){1, 1, 1, 1, 1};
}
Thread(5) Philosopher{
Code{
      --listing 4.29 and 4.31
      --get forks
      wait(g->fork(ID));
      wait(g->fork((ID+1) Rem 5));

      --put forks
      signal(g->fork(ID));
      signal(g->fork((ID+1) Rem 5));
}
}
```

We have presented several brand new bits of code. ID can be used anywhere, it’s a reserved word that is simply an integer with the value of the threads ID number.

All variables can be put into arrays using the same syntax. The only differences that occur are the initialization variables. Variables that do not get initialized do not have the {} on the end, only the count number.

Ex. `Array(CriticalSection) critArray(5);`

Ex2. `Array(String) stringArray(3){"str1", "str2", "str3"};`

The Rem function is the same as % in Java. As it has higher precedence than +, ID+1 must be put into brackets to be sure the function executes correctly.

[1] A note on the skipped problems: Not all problems from the Little Book of Semaphores are doable in ConEE, and not all of them are worth doing. ConEE by default uses strong semaphores, eliminating several problems that assume the use of weak semaphores. It can however been made to simulate a worst case scenario with weak semaphores, by change the semaphores to Waking Highest Priority First, under settings, and giving all threads a higher priority than the thread you wish to starve.

Now, try loading BasicPhilosophers into ConEE and see if you can figure out what's wrong with it. Tip: if you're having trouble, change all the 5s in the program to 3s and run Exhaustive Test. (if you don't change the numbers it will take far too long to finish testing).

Figured it out? Good. Solve the deadlock (tips in Little Book of Semaphores).

In order to be sure your code is working, you may want to write some acceptance tests to be sure your fork mutexs are working, and then run it through the Exhaustive Test. For the acceptance tests, you will likely need an array of critical sections to go with your forks.

The Exhaustive test itself could take minutes, hours, or maybe even days, depending on how well you've written your code. As that is the case, we'll wrap up this tutorial.

You've now been given a fairly comprehensive overlook of ConEEs functionality, if you have any other questions about the available commands and operations, refer to the ConEE Language guide.

ConEE Tutorial

Extra Assignment
ConEE Advanced Tutorial #2

Non Atomic Assignment

Being a simulator, ConEE displays some behaviour that does not necessarily occur in the field. All of ConEE's operations are atomic, but this not the case with many parallel programming languages. It is possible to simulate non atomic assignment in ConEE, using local variables.

For example, here is a problem that uses concurrency to sum up 10 numbers and write the sum to result :

```
      GlobalVariables{
            Array(Integer) numbersToSum(10){1, 2, 3, 4, 5, 6, 7, 8, 9,
10};
            Integer result(0);
            Integer counter(0);
      }
      Thread(4) Adders{
            Variables{
                  Integer localMemory(0);
                  Integer localCounter(0);
                  Integer localSum(0);
            }
            Code{
            sumLoop: loop
                  --read opperation
                  d->localCounter:=g->counter;
                  --arithmetic is only done with local variables
                  d->localMemory:=d->localCounter+1;
                  --write opperation
                  g->counter:=d->localMemory;

                  exit sumLoop when(d->localCounter>=10);

                  d->localMemory:=g->numbersToSum(d->localCounter);
                  d->localSum:=d->localSum+d->localMemory;

            end loop sumLoop;

            d->localMemory:=g->result;
            d->localMemory:=d->localMemory+d->localSum;
            g->result:=d->localMemory;

            --clear local variables.
            d->localMemory:=0;
            d->localCounter:=0;
            d->localSum:=0;

            }
      }
```

Parallelizing this problem may seem pointless as the task is so small, but if, say, a bank wanted you to sum up the 2,000,000 transactions that had occurred over the last month, knowing how to do it in parallel could prove useful.

Puzzle: Add Semaphores so that program sums correctly (You can modify the code as much as you like, but be sure that it is still concurrent).

There are two ways to test your solution. The first is to use acceptance tests. The second is to use ConEE's data race tester. Under Setting there is a checkbox that says: Test for data races and race conditions. Checking this box will have ConEE check the program's variables at the end of the program compare them against its first run to see if the result is different. If any variable is different, then ConEE throws an error.

This problem is too long for ConEE to test completely, but leaving the test running for five to ten minutes can assure you that you haven't got any glaring errors.