ConEE Language Guide

Written by Anna Offenwanger

Reference Sheets:

Tip: if the program is giving odd errors, check that that all your variables have either a global tag (`g->`), or a local tag (`d->`)

Threads:

Declaration:

```
Thread(1) thread1{
  Variables{
    --variables block is optional.
  }
  Code{
    --code lines go here
  }
}
```

Variables:

```
GlobalVariables{
  Boolean nameOfBoolean(true);
  Boolean differentNameForBoolean(false);
  Integer nameOfInteger(6);
  String nameOfString("Strings must be enclosed in quotation
marks");
  Semaphore nameOfSemaphore(5);
  Mutex nameOfMutex;
  Barrier nameOfBarrier(3);
  Array(Integer) arr(3){5, 6, 7};
  Array(Semaphore) semArray(2){0, 1};
  Array(Mutex) mutArray(4);



}
```

Comments: `--double dash means comment`

Assignment: `g->globalVar:=d->localVar;`

Output statement: `put_line("String"&g->globalVar);`

Semaphore Commands:

```
wait(g->semName);
signal(g->semName);
```

If Statements:

```
if(booleanExpression)then
```

```
  --statements
elsif(booleanExpression)then
  --statements
  --elsifs are optional, and you can have as many as you like
else
  --statements
  --else is also optional, but you can only have one ☹
end if;
```

Loop Statements

```
nameOfLoop: loop
--statements
exit nameOfLoop when(booleanExpression);
--statements
end loop nameOfLoop;
--will not pass this point until it exits
```

Boolean Expressions

Calculus comparison	
Greater than:	>
Less than:	>
Greater than or equal to:	>=
Less than or equal to:	<=
Equal to	=
Not Equal to	/=
Boolean comparison	
And	AND
Or	OR
Exclusive or	XOR
Not	NOT

Calculus Expressions

Addition	+
Subtraction	-
Multiplication	*
Division	/
Remainder division	rem
Modulo division	mod

Note: floating point is not supported

Acceptance Tests:

```
AcceptanceTests{
  CriticalSection critSec;
  Rendezvous rend;
  Serialization seri;
}
```

CriticalSection

Ex: `put_line("thread is in the criticalSection") \critSec entered\`

Ex: `put_line("at this point there should be only 1 thread1 in the critical section") \check(1, thread1, critSec)\`

Rendezvous

Ex: `put_line("this is the rendezvous line") \rend entered\`

Ex: `put_line("at this point two thread1s should have arrived at the rendezvous") \check(2, thread1, rend)\`

Serialization

Ex: `put_line("this is the line monitored") \seri executed\`

Ex: `put_line("at this point the line should have executed once") \check(1, seri)\`

Use Of Boolean expressions

Ex: `\check(2, thread1, rend) OR g->counter/=2\`

Ex: `\NOT(check(2, thread1, rend)) OR check(1, thread1, critSec)\`

Arrays

Ex: `Array(CriticalSection) csArray(3);`

Ex: `\check(1, thread1, csArray(0))\`

Ex: `\csArray(0) entered\`

Ex: `\csArray(ID-1 rem 3) exited\`

Detailed Explanations:

Threads:

ConEE is primarily a multithread simulator and tester.
Threads contain a block of code, and optionally a variable block, which declares variables local to a specific instance of that thread type.

Thread declaration syntax: Thread(numberOfThisThreadType) nameOfThread{

The code block and variable blocks follow the Thread declaration.

Complete thread block:

```
Thread(1) thread1{
```

```
Variables{
--variable block is optional.
}
Code{
--code lines go here
}
}
```

Variables:

Variables are declared in a variable declaration block, indicated by the reserved word GlobalVariables (declared outside thread blocks and accessible by any thread), or the reserved word Variable (declared inside thread blocks and accessible by individual threads only). The reserved word is followed by a curly bracket, and then the variables declared on separate lines. On the last line place the closing curly bracket.

Variable declaration syntax: VariableType nameOfVariable(initialValue);

The four basic variables types are
Boolean
Integer
String
Semaphore

All four take an initial parameter.

Complete variable declaration block:

```
GlobalVariables{
  Boolean nameOfBoolean(true);
  Boolean differentNameForBoolean(false);
  Integer nameOfInteger(6);
  String nameOfString("Strings must be enclosed in quotation
marks");
  Semaphore nameOfSemaphore(5);
}
```

Advanced Variables:

There are two advanced variable types, the Mutex and the Barrier. The Mutex and Barrier are both specialized kinds of Semaphore.

The Mutex is a semaphore that is automatically initialized to one, and so doesn't take a parameter.

```
Mutex nameOfMutex;
```

The Barrier is more complex. A Barrier is a Semaphore with a max capacity, that blocks all threads and increments its value. If the value equals its max capacity, and then it releases all waiting threads simultaneously and resets its value to zero.

The Barrier takes one parameter, its max capacity.

```
Barrier nameOfBarrier(3);
```

Note: signal cannot be used on Barriers, only wait.

Arrays:

ConEE also provides an array syntax.

Array declaration syntax: Array(TypeOfArray) nameOfArray(sizeOfArray){intiVal1, initVal2, … , initValN};

```
Variables{
  Array(String) strArray(3){“str1”, “str2”, “str3”};
  Array(Semaphore) semArray(2){0, 1};
  --Mutexs are a bit different as they cannot be
initialized.
  Array(Mutex) mutArray(5);
}
```

Syntax Notes:
-Capitalization in names is disregarded.
-Indentation and white space are disregarded.
-Variables cannot have the same name, unless one is a global variable and the other is a local variable.
-All declaration statements must end with a semicolon (`;`).

Code/Language:

The ConEE language consists of five different command types:

Assignments
Output statement
Semaphore commands
If statements
Loops

Language notes:
-Only one command per line
-Almost all commands must end with a semicolon (`;`).

-The exceptions to this rule are the beginnings of Loops and If Statements.

Comments:

```
--comments are indicated by a double dash
```

Assignments and Variables:

When indicating a variable you must tell the program whether to look for it in the global variables or the local variables.

GlobalVariable indication: `g->variableName`

LocalVariable indication: `d->variableName`

To assign a new value to a variable, use: `:=`

```
g->variableName := g->variableName + d->variableName +5;
```

Assignment notes:

-It is very easy to forget the g->/d->. If a code line is giving an error, check the variables.

-all Assignment commands must be followed by the endExpression, which is a semicolon (;).

Arrays:

To access an array: g->array(index)

```
g->variableName(0):= g->variableName(1)+d->variableName(1);
```

Output Statements:

There is only one output statement: `put_line(“String goes here”);`

Strings can be concatenated with variables using &:

```
d->variableName & “String” & g->variableName
```

Semaphore Commands:

Semaphores are accessed like any other variable, using g->/d->.

The Semaphore specific commands are:

```
wait(g->semName);
signal(g->semName);
P(g->semName);
V(g->semName);
```

wait and P have identical functionality, so do signal and V.

wait decrements the semaphore, and blocks the thread if the semaphore goes negative. signal increments the semaphore and wakes a waiting thread, if any.

Semaphores by default wake threads on a First In First Out basis. ConEE provides an option to switch all semaphores to wake on a Highest Priority First basis in the settings dialog.

Arrays:

Example:

```
wait(g->semName(g->index));
signal(d->semName(4));
```

If Statements

If statement command:

```
if(booleanExpression)then
--statements
elsif(booleanExpression)then
--statements
--elsifs are optional, and you can have as many as you like
else
--statements
--else is also optional, but you can only have one ☹
end if;
```

Notes:

-if statements must always end with an end if;

-for more on Boolean expressions, see Boolean Expressions

-no endExpression (;) after the `then`s and `else`.

Loop Statements

Loop statement:

```
nameOfLoop: loop
--statements
exit nameOfLoop when(booleanExpression);
--statements
end loop nameOfLoop;
```

Notes:

-no endExpression (;) after the loop declaration, but it is necessary after all the other commands.

-the exit statement can go anywhere inside the loop.

-all loops are infinite if they do not contain a break statement.

Operators

Boolean operators

Calculus comparison	
Greater than:	>
Less than:	>
Greater than or equal to:	>=

Less than or equal to:	`<=`
Equal to	`=`
Not Equal to	`/=`
Boolean comparison	
And	`AND`
Or	`OR`
Exclusive Or	`XOR`
Not	`NOT`

A complicated Boolean expression:

```
(g->anInteger>6 AND g->aBoolean OR NOT(d->aLocalBoolean XOR g->anInteger<0))
```

Notes:
-use of brackets is recommended

Calculus Operators

Addition	`+`
Subtraction	`-`
Multiplication	`*`
Division	`/`
Remainder division	`Rem`
Modulo division	`Mod`

Notes:
-only useable with integers, floating point is not supported
-use of brackets is recommended

Operator precedence (highest to lowest):

```
NOT, *, /, Mod, Rem
+, -
=, /=, <, >, <=, >=
AND, OR, XOR
```

Notes:
-Calculus expressions will always we evaluated before Boolean.

Thread ID

Each thread is given an identification number when it is created, this can be accessed via the reserved word ID:

```
g->anInteger:=g->anInteger+ID;
```

Acceptance Testing:

For testing purposes ConEE provides an extra syntax.
Test lines are appended to a line of code surrounded by backslashes, and will execute when the code line does.
If an acceptance test is on a blank line, then it executes when the command above it does.

The acceptance testing provides a set of special variables, which get declared inside an AcceptanceTests block, similar to the GlobalVariables:

```
AcceptanceTests{
  CriticalSection critSec;
  Rendezvous rend;
  Serialization seri;
}
```

The three kinds of Acceptance test variables hold counters as follows

CriticalSection: holds a counter for each thread type, and increments them when a thread of that type enters the critical section it monitors. It uses the command `entered` to indicate that a thread is now in the section it monitors, and `exited` to indicate that the thread has left the section it monitors.
Ex: `put_line(“thread is in the critSection”) \critSec entered\`
Ex: `put_line(“thread has left the critSection”) \critSec exited\`

Rendezvous: holds a counter for each thread type, and increments it when a thread arrives at the rendezvous it monitors. It uses the command `arrived` to indicate that a thread has arrived at the rendezvous.
Ex: `put_line(“this is the rendezvous”) \rend arrived\`

Serialization: holds only one counter that counts the number of times the line it monitors executes. It uses the command executed to indicate that the line it monitors has executed.
Ex: `put_line(“this is the line monitored”) \seri executed\`

The acceptance test provides one command to be used in conjunction with the variables for testing:
check(intgerToCheckAgainst, threadTypeWeWishToCheck, nameOfAcceptanceTestVariable)

check has different functions depending on which type of AccetanceTest variable it is used with.

CriticalSection: checks the integerToCheckAgainst with its counter for the threadTypeWeWishToCheck, if the integer is less than or equal to the count for the thread type

(i.e. there are no more than x of thread type y in the critical section), it returns true, else returns false.
Ex: `put_line(“at this point there should be only 1 thread1 in the critical section”) \check(1, thread1, critSec)\`

Rendezvous: checks the integerToCheckAgainst against its counter for the threadTypeWeWishToCheck, if the integer is greater than or equal to the count for the thread type (i.e. there are at least x of thread type y at the rendezvous), it returns true, else returns false.
Ex: `put_line(“at this point two thread1s should have arrived at the rendezvous”) \check(2, thread1, rend)\`

Serialization: has a slightly different check command format:
check(intgerToCheckAgainst, nameOfSerializetion)
The Serialization checks the intgerToCheckAgainst against its counter, if the number is equal to or great than (i.e. the line executed at least x many times) it returns true, else returns false.
Ex: `put_line(“at this point the line should have executed once”) \check(1, seri)\`

If the check returns true the program continues on, if it returns false it throws an error and pops up a dialog with the test which failed.

checks can be used in conjunction with variables and ConEE’s Boolean Expression syntax to create more thorough tests.
Ex: `\check(2, thread1, rend) OR g->counter/=2\`
Ex2: `\NOT(check(2, thread1, rend)) OR check(1, thread1, critSec)\`
If the Boolean Expression returns true, the test passes.

Acceptance test example in a program:

```
GlobalVariables{
  Semaphore sem(1);
}
AcceptanceTest{
  CriticalSection cs1;
}
Thread(4) thread1{
  Code{
    wait(g->sem);
      put_line("In mutex"); \cs1 entered\ \check(1, thread1, cs1)\
    signal(g->sem); \cs1 exited\
  }
}
```

Note: Multiple tests lines can be placed after a line of code, but each needs to have its own backslash set.

Arrays:

AcceptanceTest variables can be created in arrays in the same way as Mutexs.

Ex: `Array(CriticalSection) csArray(3);`

This creates an array of three criticalSections. They are accessed in the same manner as other variables.

Ex: `\check(1, thread1, csArray(0))\`

Ex2: `\csArray(0) entered\`

Ex3: `\csArray(ID-1 rem 3) exited\`

Program Examples:

Example 1:

```
--Leaders and Followers
GlobalVariables{
     Semaphore lQueue(0);
     Semaphore fQueue(0);
     Mutex     lMut;
     Mutex     fMut;

}
AcceptanceTests{
     Rendezvous ren1;
     CriticalSection cs1;
}

Thread(2) follower{
     Code{
          wait(g->fMut);
          signal(g->lQueue); \ren1 arrived\ \cs1 entered\
          wait(g->fQueue);
          --dance
          signal(g->fMut);
          \check(1, leader, ren1) AND check(1, follower, cs1)\
          \cs1 exited\
          --these acceptance tests execute when the
          --previous line does.
     }
}

Thread(2) leader{
     Code{
          wait(g->lMut);
          signal(g->fQueue); \ren1 arrived\ \cs1 entered\
          wait(g->lQueue);
          --dance
          signal(g->lMut);
          \check(1, follower, ren1) AND check(1, leader, cs1)\
          \cs1 exited\
     }
}
```

Example 2:

```
--dining Philosophers
GlobalVariables{
     Array(Mutex) fork(5);
}

AcceptanceTests{
Array(CriticalSection) crit(5);
}

Thread(5) Philosopher{
     Code{
          --think
          if(ID=0)then
               wait(g->fork((ID+1) rem 5));
                    wait(g->fork(ID));
                    \crit((ID+1) rem 5) entered\
                    \check(1, Philosopher, crit((ID+1) rem 5))\
                    signal(g->fork(ID));
                    \crit(ID) entered\
                    \check(1, Philosopher, crit(ID))\
                    \crit(ID) exited\
               signal(g->fork((ID+1) rem 5));
               \crit((ID+1) rem 5) exited\
          elsif(ID/=0)then
               wait(g->fork(ID));
                    wait(g->fork((ID+1) rem 5));
                    \crit(ID) entered\
                    \check(1, Philosopher, crit(ID))\
                    signal(g->fork((ID+1) rem 5));
                    \crit((ID+1) rem 5) entered\
                    \check(1, Philosopher, crit((ID+1) rem 5))\
                    \crit((ID+1) rem 5) exited\
               signal(g->fork(ID)); \crit(ID) exited\
          end if;

          --eat
          }
}
```