



OPENTTCN

OpenTTCN Debugger Tutorial OpenTTCN Tester 2011

VERSION 4.1.6 – April 25, 2012

Contents

1	OVERVIEW OF OPENTTCN DEBUGGER OF OPENTTCN TESTER 2011	3
2	FUNCTIONALITY OF OPENTTCN DEBUGGER	4
3	DEBUGGING THE EXAMPLE TEST SUITE	5
3.1	EXECUTING TEST CAMPAIGNS IN THE DEBUGGER	5
3.2	DEBUGGING WITH THE DEBUG PERSPECTIVE	7
3.3	DEBUGGER CONTROLS	9
3.4	DEBUGGER SUMMARY	11
4	DEBUGGING COMMON PROBLEM SCENARIOS	12
4.1	DEBUGGING TEST CASE DEADLOCKS	12
4.2	INTRODUCING THE DEADLOCK	12
4.3	DEBUGGING THE DEADLOCK	13
5	CONTACTING OPENTTCN	14

1 Overview of OpenTTCN Debugger of OpenTTCN Tester 2011

OpenTTCN Tester 2011 is a TTCN-3 test development and execution package built by integrating popular OpenTTCN Tester product and widely used Eclipse framework providing an easy to use TTCN-3 editing, compilation, and execution environment.

Two major new features of OpenTTCN Tester 2011 are TTCN-3 debugger and GFT (Graphical Presentation Format) log viewer.

The latest technical preview version of OpenTTCN Tester combines new OpenTTCN Debugger component providing the capability to debug TTCN-3 code at runtime. This tutorial explains the basic features of the debugger and provides some examples of debugging basic errors found in test suites.

The basic structure of the guide is as follows: At first brief description of OpenTTCN Debugger functionality is given. Next we demonstrate use of the debugger with a simple example test suite. Then we demonstrate finding specific kinds of bugs from a test suite, such as deadlocks.

2 Functionality of OpenTTCN Debugger

OpenTTCN TTCN-3 debugger allows running TTCN-3 code in debug mode without special debug compilation. Therefore all TTCN-3 code can be then run in debug mode during TTCN-3 test case development and test case validation, allowing use of debugger features with ease whenever needed. User can proceed debugging without extra delay. Debugging is done in separate Eclipse Debug perspective.

The debugger allows setting break- and tracepoints by double clicking next to the source code line of interest. Breakpoints allow suspending TTCN-3 test case execution in order to allow examination of parameter and variable values at any given line. Tracepoints allow inspection of parameter and variable values without suspending the test case execution. Execution can be suspended when breakpoint is hit, or when a user breaks execution manually. When the execution has been suspended with the debugger, the execution can be continued with step in, step over, step out, and continue instructions.

Multiple test components can be debugged in sequence. Test component to be debugged can be selected from the list of available test components. This can be done after the test execution has been suspended. Under the selected test component, the user can select of specific stack frame for display of parameters and variable values of that context.

3 Debugging the Example Test Suite

We will be using the Fibonacci example included with the OpenTTCN Tester distribution. To access the example, you must first import the TTCN-3 code into the workspace. When first starting OpenTTCN Tester, the welcome screen contains the option to *"Create example workspace"*. Select this option to create the example projects including the Fibonacci project. If you have an existing workspace, either create a new workspace for experiments, or select *Help -> OpenTTCN Tester Welcome Screen* to access the welcome screen again, and create the Fibonacci example.

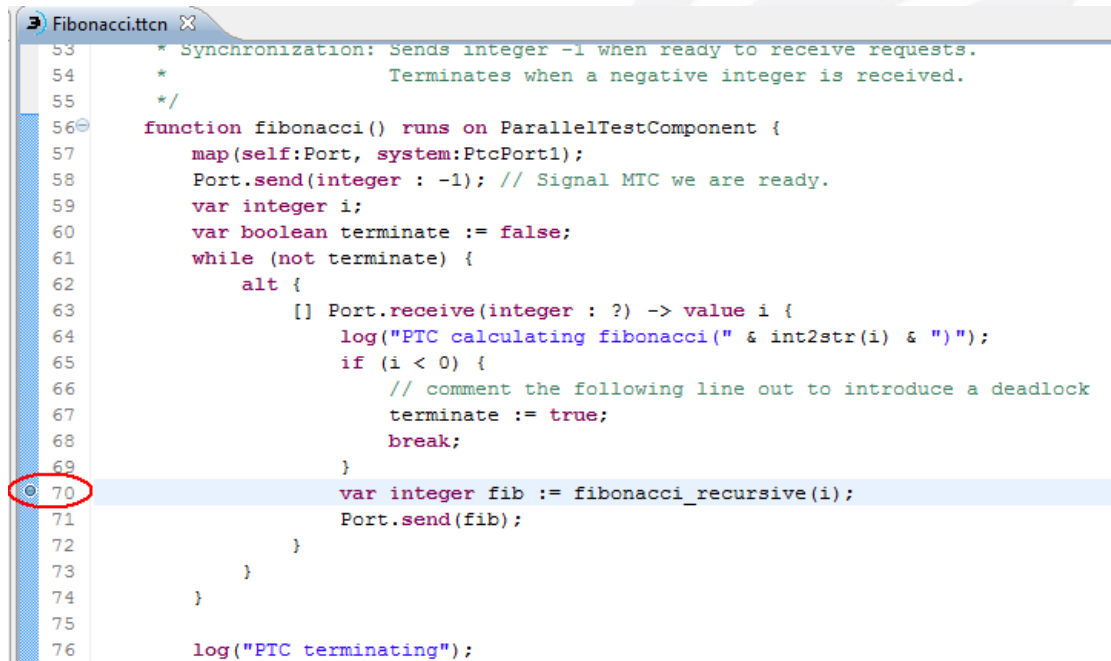
The Fibonacci example is a simple one test suite consisting of one test case. It uses a Parallel Test Component (PTC) as a Fibonacci calculating server that can be requested to calculate the i'th number in the Fibonacci sequence (0,1,1,2,3,5,8,13 and so forth, read more at http://en.wikipedia.org/wiki/Fibonacci_number). After the PTC is started and synchronized, the MTC then sends it a request to calculate the **10th** number in the Fibonacci series, and the PTC complies, responding with **55**.

You can run the test case by selecting its launch configuration from the list of launch configurations. The test campaign should execute and you should receive one pass verdict. Next let's look at how to run the test case in the debugger.

3.1 Executing test campaigns in the debugger

The debugger uses the same launch configurations as normal test campaign execution. First we must however define a breakpoint at the location we wish to examine in the debugger. Open the Fibonacci.tcn file (under Fibonacci project) in the editor by double-clicking it. Let us place a breakpoint at the point where the recursive fibonacci_recursive() function is invoked by the PTC. Navigate to line 70 and double-click the left margin to set a breakpoint. Alternatively you can right-click the margin and select *"Toggle Breakpoint"*.

You can see to breakpoint set in Image 1. The margin location you must double-click to set the breakpoint is circled in red (Note that the line numbers are not visible by default, you can turn them on by right-clicking in the text editing area, select "Preferences..." and in the dialog tick "Show line numbers").



```
53  * Synchronization: Sends integer -1 when ready to receive requests.
54  *
55  */
56  function fibonacci() runs on ParallelTestComponent {
57    map(self:Port, system:PtcPort1);
58    Port.send(integer : -1); // Signal MTC we are ready.
59    var integer i;
60    var boolean terminate := false;
61    while (not terminate) {
62      alt {
63        [] Port.receive(integer : ?) -> value i {
64          log("PTC calculating fibonacci(" & int2str(i) & ")");
65          if (i < 0) {
66            // comment the following line out to introduce a deadlock
67            terminate := true;
68            break;
69          }
70          var integer fib := fibonacci_recursive(i);
71          Port.send(fib);
72        }
73      }
74    }
75
76    log("PTC terminating");
```

Image 1. Set breakpoint on the highlighted line by double-clicking on the margin.

Now that we have our breakpoint set, let's switch to the debug perspective: Click on the icon near the top right corner of your workbench showing a table with a plus sign. From the list of perspectives, select "Debug". You can see this illustrated in Image 2.

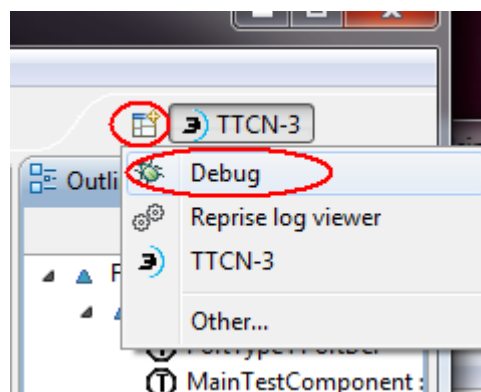


Image 2. Open the debug perspective by clicking the plus icon and selecting Debug from the list.

Now we are ready to launch. Since the Fibonacci example comes with a ready-made launch configuration, simply select the same launch configuration you used to execute the test campaign normally from the list of debug configurations to start debugging. The proper menu is shown in Image 3.

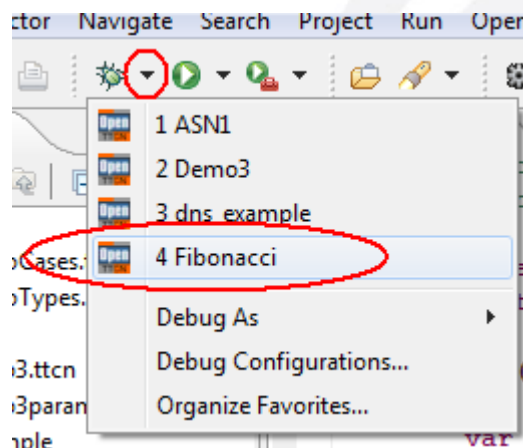


Image 3. Launch the Fibonacci launch configuration as a debug configuration from the debug dropdown menu.

The test campaign should start and rapidly break at the breakpoint we set. In the next chapter, we'll go over how the debug perspective looks after breaking on a breakpoint and the operations you can perform there.

3.2 Debugging with the Debug Perspective

When a breakpoint is hit, the debug perspective automatically shows the line where the break happened. In this case, it's the line where we set our only breakpoint. Image 4 shows what the debug perspective looks like after hitting the breakpoint.

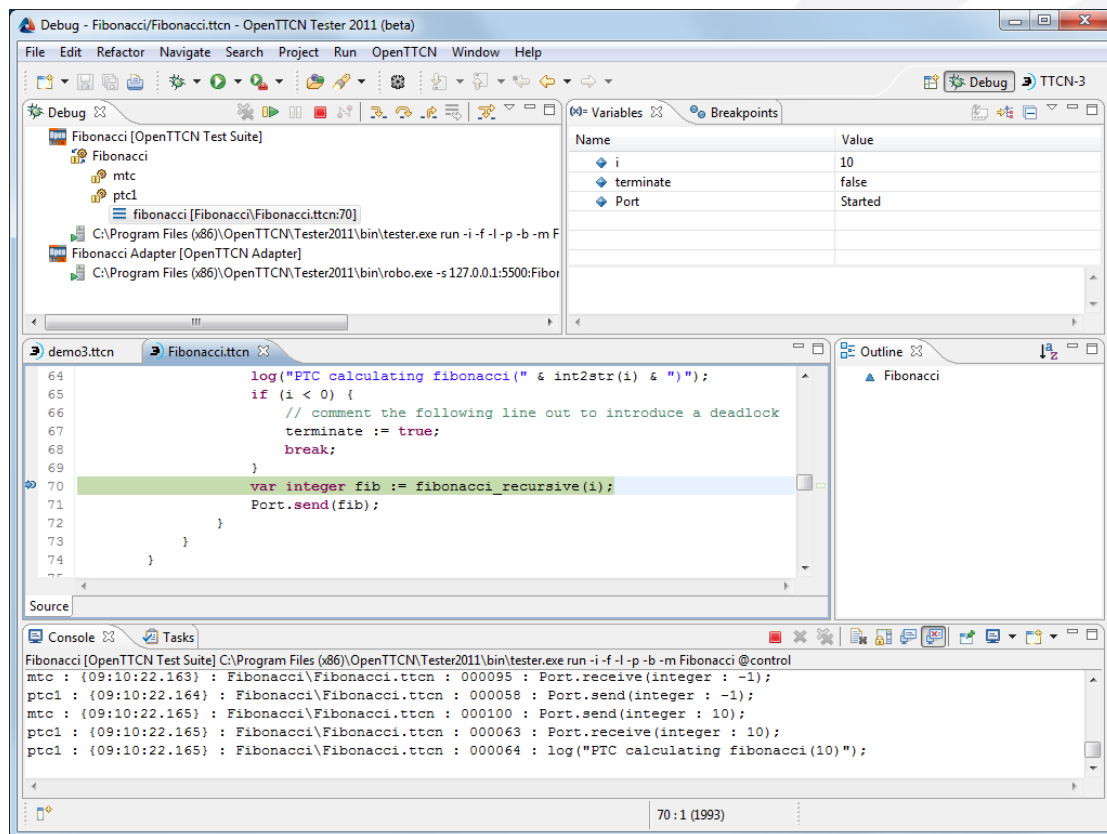


Image 4. Debug perspective after hitting breakpoint.

Let's run over the views visible on the debug workbench: On the top left is the *Debug* view showing the currently running items. The Fibonacci test campaign is visible, with two running test component, mtc and ptc1. From the toolbar in the Debug view you can control the execution by breaking, continuing, terminating and stepping over code. We'll get back to this later. Clicking on a stack frame will select that frame as the active stack frame. The *Variables* and code views will be updated to show the active stack frame.

On the top right of the perspective is the *Variables* and *Breakpoints* views. The *Breakpoints* view is hidden in Image 4 behind *Variables*. The *Breakpoints* view shows a list of active breakpoints, and you can also easily disable and remove breakpoints there. The *Variables* view shows the variables in the active stack frame. In this case its showing that the Fibonacci value we will be calculating (*i*) is 10.

In the middle is the code and outline views, that show the code for the current stack frame. Modifications made here will not be reflected in the running code until the test campaign is restarted. At the bottom is the test log for the test campaign.

3.3 Debugger Controls

You can control the execution of the test campaign with the debugger controls shown in Image 5, or by pressing the corresponding hotkeys. The following list refers to the red numbers shown in the image 5:

- 1) Resume (F8): Resumes execution of the test campaign regardless of how it was suspended.
- 2) Suspend: Suspends a running test campaign at the next opportune time. Mostly used for debugging deadlock situations.
- 3) Terminate (Ctrl-F2): Terminates the test campaign.
- 4) Step In (F5): Steps the active stack frame one line at a time, entering into any functions.
- 5) Step Over (F6): Steps the active stack frame one line at a time, skipping over any function calls (moves directly to next line in current stack frame).
- 6) Step Out (F7): Steps out of current stack frame. Execution continues until current stack frame returns to the previous level.

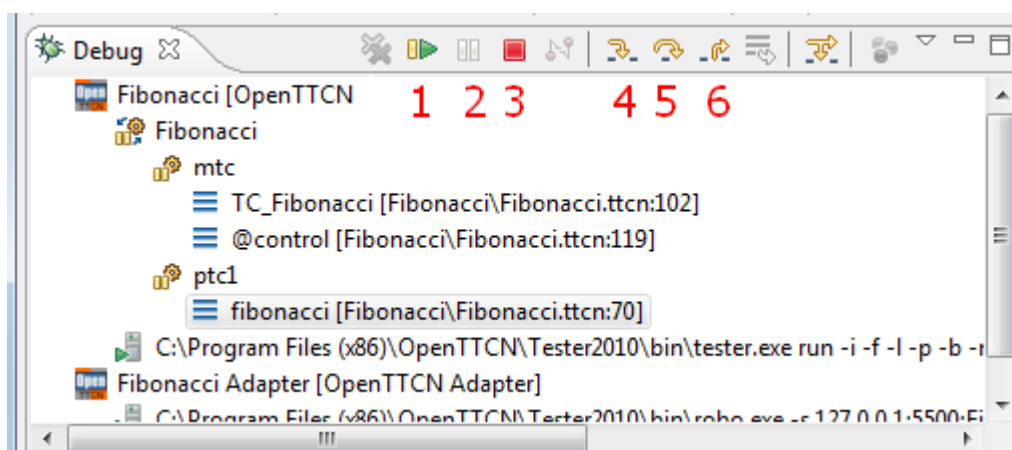


Image 5. Control buttons in the *Debug* view.

Let's try these controls with our debugging session. We were suspended at the breakpoint where the system is about to call *fibonacci_recursive(10)* for the first time. Try stepping through the code to see how the Fibonacci calculation proceeds. Make sure you have *ptc1* as the active stack frame and press F5 once to enter the *fibonacci_recursive* function. Now press F6 four times to skip over the input check, the end condition check and the two recursive call to *fibonacci_recursive()*. You should end up on the last line of the function with the return statement, as shown in Image 6.

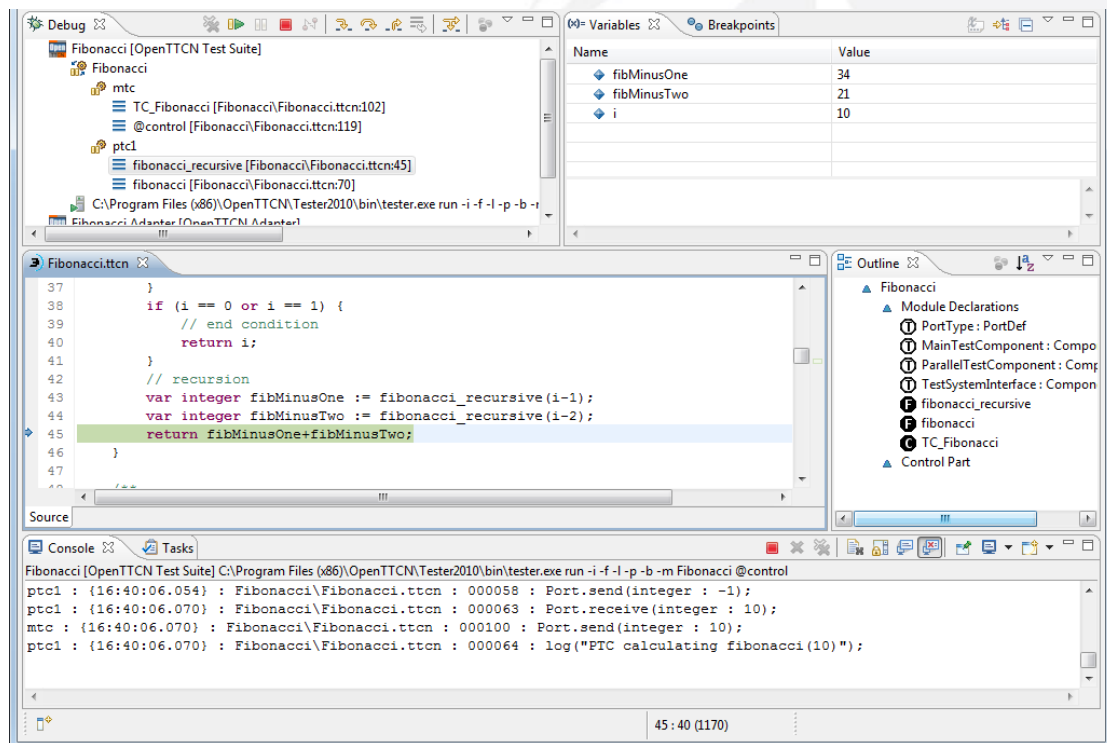


Image 6. View on screen after stepping through the *fibonacci_recursive()* function.

Here in the *Variables* view you can see the **fibMinusOne** variable contains the sum 34 from one branch of the recursion, while **fibMinusTwo** shows the sum 21 from the other branch of recursion. 34+21 is 55, which is indeed the 10th number in the Fibonacci series we are calculating (variable **i**). You can now press F8 to let the test campaign continue execution till the end.

3.4 Debugger Summary

This simple example showed you how to navigate code in the debugger. As a test of your skills, you could now try removing the break points set previously (e.g. through the *Breakpoints* view) and set a new breakpoint at the end condition of the recursion. This is line 40 as visible in Image 6 on the previous page, with the **return i;** statement. If you now launch the debug session again, the execution will stop with many levels of recursive calls of *fibonacci_recursive()* visible. You can click on the stack frames to see the path the recursion has taken and see in the variables and their values in each level of the stack.

If you press F8, the execution will continue until the next recursion hits the end condition. Here you can see why this recursive algorithm is inefficient, as it will recurse to the end condition very many times while calculating the same intermediate results over and over, before arriving at a final result. You can also choose to use Step Out (F7) and Step Over (F6) to work your way out from the recursive call one level at a time and see how the end result is put together through the *Variables* view.

Debugging recursive code may be complex at times. Rest assured that it is much easier to debug normal functional code.

4 Debugging Common Problem Scenarios

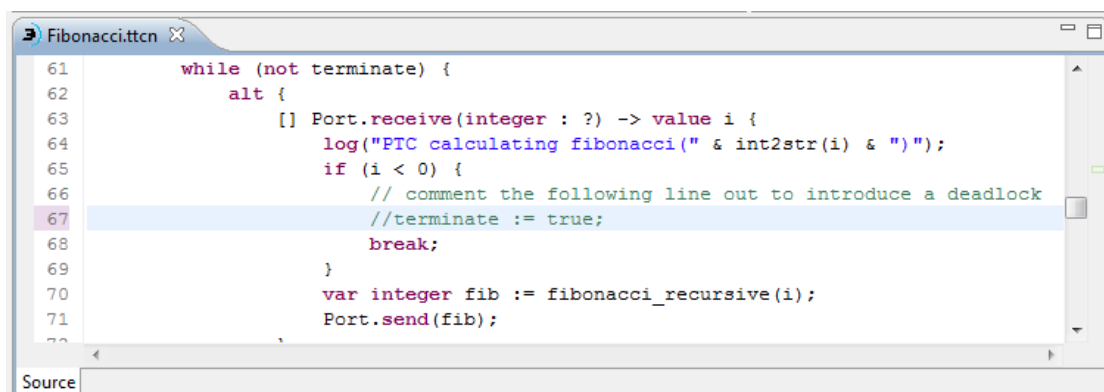
Here we go through debugging of common problem scenarios. A common problem in debugging test system with any level of parallelism, whether from Parallel Test Components (PTCs) or a non-synchronous System Under Test (SUT).

4.1 Debugging Test Case Deadlocks

Deadlocks can occur when dealing with Parallel Test Components (PTCs). Our Fibonacci example conveniently uses a PTC for calculations, so let's introduce a deadlock into the PTC and see how the debugger helps in solving it.

4.2 Introducing the Deadlock

Let us comment out the terminating condition for the PTC, causing it to ignore the termination message from the MTC. Find the line containing the code **terminate := true;** in the PTC behavior function **fibonacci()**. This should be located on line 67 as shown in Image 7. Place two slash characters before the line to comment it out ("**//**") and save the document to compile the changes.



```
61     while (not terminate) {
62         alt {
63             [] Port.receive(integer : ?) -> value i {
64                 log("PTC calculating fibonacci(" & int2str(i) & ")");
65                 if (i < 0) {
66                     // comment the following line out to introduce a deadlock
67                     //terminate := true;
68                     break;
69                 }
70                 var integer fib := fibonacci_recursive(i);
71                 Port.send(fib);
72             }
```

Image 7. Line commented out to introduce a deadlock into the test.

Make sure your test session isn't still running and that you don't have any breakpoints set (delete all breakpoints if there are any).

4.3 Debugging the Deadlock

After the setup in the previous chapter, you are ready to start debugging the deadlock issue. Launch the debug session as previously. This time the test campaign will not break, since we did not set any break points. It does not reach its natural end either. It is deadlocked.

To start solving the problem, select the launch in the *Debug* view and press the Suspend button (looks like a pause button, with two yellow vertical bars). The test execution will suspend, showing you that there are two test components running. Let's look at what they are doing to find where the deadlock is. Start by clicking on the topmost stack frame in **mtc**. The code shows that it is waiting for a PTC to be done. From the variables view we can see it has already calculated the Fibonacci number, and from the comment on the previous line we know it has already sent a message to the PTC to terminate.

Based on the previous, let's see what the PTC is doing. Select the topmost stack frame of **ptc1** in the *Debug* view. It is waiting to receive a message. Looking at the variables, you can see that the previous received message was **-2** which is the termination message, but the variable **terminate** is still *false*.

This trail leads us right to the line we commented out, which was supposed to set the **terminate** flag to true when the terminate message is received. Now that we've found the problem, remove the comment sign in front of the terminate line to remove the bug and press save to compile the changes. To terminate the still deadlocked test campaign, you must press the red stop button in the *Debug* view. Finally, you can launch the test campaign again in Debug mode to confirm that it indeed runs through smoothly once again.

5 Contacting OpenTTCN

Please give us your feedback about OpenTTCN Debugger and OpenTTCN Tester 2011 in general by sending your questions using support@openttcn.fi e-mail address.

Latest user guides and training course materials can be found from:

<http://www.openttcn.com/support/user-guides>

Tutorials and articles about OpenTTCN use and programming can be found from:

<http://wiki.openttcn.com>

You can contact OpenTTCN sales by sales@openttcn.fi e-mail.

OpenTTCN wishes you a good time using OpenTTCN Tester 2011!