How TUnit Works

From TinyOS Wiki

There are two elements to the TUnit unit testing framework: a Java engine that drives the test process, and an embedded library that runs the test on the node. This section will give a brief overview of how these two components of TUnit work, and other tutorials (http://docs.tinyos.net/index.php/TUnit) will dive in further with more details.

Contents

TUnit Java Application

On the computer end, a Java application is in charge of deciding which tests to run. The Java side of TUnit is the engine which compiles the tests, installs the tests to microcontroller hardware, runs the tests, cleans up the tests, and gathers the results. If a test causes the embedded hardware to lock-up, the Java TUnit application will safely timeout and continue on with further testing.

TUnit is run from the tunit.jar file, which is checked out with the tinyos-2.x-contrib repository (http://tinyos.cvs.sourceforge.net/tinyos/tinyos-2.x-contrib/) along with the rest of the TUnit framework (http://tinyos.cvs.sourceforge.net/tinyos/tinyos-2.x-contrib/tunit/) . The tunit.jar (http://tinyos.cvs.sourceforge.net/tinyos/tinyos-2.x-contrib/tunit/support/sdk/tunit/) file depends on several other .jar (http://tinyos.cvs.sourceforge.net/tinyos/tinyos-2.x-contrib/tunit/support/sdk/tunit/depends/) files existing in your Java classpath. The setup of the tunit.jar file, classpath, and other dependencies is discussed further in Setting up TUnit.

Configuring Nodes to Test

TUnit relies on nodes being plugged in to the computer, which allows tests to execute on actual hardware. Running a test on real hardware can sometimes be very important: performance variations and other issues may arise on a per-platform basis.

It is difficult to automatically detect what type of hardware is plugged into the computer, especially when variations of even specific hardware types exist. To help TUnit out, we have to tell it in an XML (http://en.wikipedia.org/wiki/Xml) file what hardware we have plugged into the computer, how to talk to it, and which combinations of hardware we want to test. In TUnit, each combination of hardware being tested forms a Test Run.

The file that tells the TUnit Java application which nodes are connected is the tunit.xml file.

After locating, reading, and parsing the tunit.xml file, TUnit verifies all nodes you'd like to test are actually connected to your machine. If any nodes are missing, TUnit notifies you of the problem by creating an error. Everything in TUnit is a test, and this is no exception.

Locating Test Suites

TUnit begins by sequentially focusing on each Test Run defined in the tunit.xml file. With the first Test Run in mind, TUnit traverses through all directories and sub-directories attempting to find Test Suite (http://en.wikipedia.org/wiki/Test_suite) that apply to that Test Run. A Test Suite will only be executed if it is found to apply to the hardware combination defined in the current Test Run. Test that do not apply to the current Test Run are filtered out and ignored.

For example, if you have a Test Suite (http://en.wikipedia.org/wiki/Test_suite) that applies only to a single TMote, you wouldn't want those tests to run on hardware platforms other than a single TMote. And you also wouldn't want it running on two TMotes, since one is enough. The suite.properties file is responsible for defining the rules of each Test Suite. Using the suite.properties file, it is possible to filter out combinations of hardware that do not apply. As TUnit traverses deeper and deeper into sub-directories attempting to find tests to run, the rules that are defined in each directory's suite.properties get added together. In this manner, rules can be defined very easily to apply to an entire branch of sub-directories containing unit tests.

A Test Suite (http://en.wikipedia.org/wiki/Test_suite) is attempted to be compiled, installed, and executed if it meets two criteria:

- A directory is found to contain a compilable application, complete with a valid Makefile
- The aggregate rules setup in all previous and current suite.properties files do not prohibit the test from being run on the combination of hardware currently being tested.

Executing Tests

A Test Suite (http://en.wikipedia.org/wiki/Test_suite) that applies to the current Test Run is compiled. The compilation itself is a test, and included in the results at the end. After compiling, TUnit installs the application to each physical node, defined in the current Test Run it is focused on. Each node is given a unique incremental address, starting with 0. After the application is installed to all nodes in the Test Run, the Java side of TUnit gives the command to begin the test. This command is given over the serial port to a single node: the Driving Node, node 0.

Driving Node vs. Supporting Nodes

Tests may be run on a single node or multiple nodes.

Unit testing code can mostly be done using only a single node. After all, one of the main concepts of certain types of unit testing is to isolate a unit of code from all other factors and influences. Other tests, such as radio tests and especially wireless performance tests, require multiple nodes.

The Driving Node is the primary node in the test. It always has an address of 0, and is the first node defined at the top of a single Test Run in the tunit.xml file. The Driving Node gets the command to begin testing from the computer, and will run through each test in the Test Suite.

A Supporting Node is any other node that plays a part in the test. Supporting Nodes, upon having their Test Suite application installed, may begin performing some function immediately (i.e. transmitting or receiving) or they may be alerted wirelessly by the Driving Node to begin performing some function. These nodes are typically used to characterize performance or proper integration of some code with the radio stack. Unit tests consisting of multiple nodes is an advanced topic that is covered in the Multi-Node Unit Testing section.

Test Flow

The flow of TUnit tests follows that of other XUnit (http://en.wikipedia.org/wiki/XUnit#External_links) frameworks:

- **SetUpOneTime()** - Optionally runs exactly one time at the beginning of the entire Test Suite, like a Boot.booted(). All nodes run SetUpOneTime(), and Supporting Nodes run it before the Driving Node.
    - **SetUp()** - Optionally runs at the beginning of each individual test. This is only run by the Driving Node.
        - **TestCase.run()** - Your test that you define. This is only run by the Driving Node.
    - **TearDown()** - Optionally runs at the end of each individual test. This is only run by the Driving Node.
- **TearDownOneTime()** - Optionally runs exactly one time at the end of the entire Test Suite. This runs on every node that took part in the test.

The test flow is covered in further detail in the TUnit Test Flow section.

Test Cases

Each Test Case (http://en.wikipedia.org/wiki/Test_case) , executed by the Driving Node, must make at least one assertion (http://en.wikipedia.org/wiki/Assertion_%28computing%29) . TUnit assertions are discussed in the TUnit Assertions section.

Like many things in TinyOS, Test Cases are split-phase. When a Test Case is instructed to run(), it could be some time later that the test actually finishes. It is imperative to alert the TUnit framework when your test is finished by calling TestCase.done() at the end of the test. At that point, the next Test Case (if any) is executed. When all tests in the Test Suite have completed, the Java TUnit application instructs all nodes to run their optional TearDownOneTime() function.

Results

At the end of execution, the TUnit Java application prints out a summary of the results and all errors / failures it encountered to the command line. This is useful for an individual developer running individual unit tests in an ad-hoc manner.

During execution, TUnit also outputs all results from every Test Suite into XML files. With Ant, these XML files are converted into HTML files (http://www.lavalampmotemasters.com/reports/html/index.html) . TUnit has extra functionality to clean up those HTML files, adding in graphs and statistics for extra information beyond a boolean success/fail.

TUnit Embedded Library

On the embedded side, a TUnit library is added into the compilation of each Test Suite (http://en.wikipedia.org/wiki/Test_suite) by the TUnit Java application when compiling. This library is located in tinyos-2.x-contrib/tunit/tos/lib/tunit (http://tinyos.cvs.sourceforge.net/tinyos/tinyos-2.x-contrib/tunit/tos/lib/tunit/) .

The embedded library is responsible making it easy for you, the developer, to quickly and easily write unit tests without having to think much about details. This library will:

- Communicate with the computer over the regular TinyOS serial stack
- Queue up assertions and other messages generated by your test until given the chance to pass those messages to the computer. These messages are flushed between each individual Test Case, so its message queue is always empty at the start of each test.
- The library tells your test what to do: SetUpOneTime(), SetUp(), TestCase.run(), TearDown(), TearDownOneTime(). All your unit test must do is obey the orders its given and respond back when you're done().

The TUnit embedded library, including the serial stack, takes up around 7 kB of ROM (depending on the microcontroller). The RAM is more configurable - although the default message queue is around 5, the suite.properties file can increase or decrease the number of assertions possible to queue up for each Test Case through its @assertions parameter.

TUnit Shortcomings

Although TUnit is meant to be easy to use and follows industry standard testing philosophies, there are a few elements of the framework that could cause problems.

The first pitfall, especially to newcomers, is TUnit's requirement that your code tell the framework when it's done(). Being a split-phase operating system, there isn't much TUnit can do to infer when your test is done executing (except checking the task scheduler, which is a fallacy - your test may still be running). By not telling the framework when something is done(), the testing process cannot continue. This will cause your test to time out. It typically takes writing a few tests before it becomes natural to call done() when you're done.

The second shortcoming is TUnit's dependency on the serial stack. If the TinyOS serial stack is not working, then TUnit doesn't work - but at least you would have a good general idea of where the problem is. This can make unit testing brand new microcontrollers impossible until the serial stack is working. It also makes it difficult to test the serial stack itself. It has been proven, however, that TUnit can test the low-level UartByte and UartStream interfaces as they are integrated with the computer.

See Also

- TUnit
- Setting up TUnit

Next

- TUnit Philosophy

Retrieved from "http://tinyos.stanford.edu/tinyos-wiki/index.php?title=How_TUnit_Works&oldid=2999"

Category: TUnit

- This page was last modified on 26 February 2010, at 03:26.
- This page has been accessed 13,825 times.