State Interface Test

From TinyOS Wiki

We'll go over the creation a simple isolation test that verifies the behavior of the State interface, provided in the tinyos-2.x baseline by the StateC component (tinyos-2.x/tos/system/StateC.nc).

Contents

State Interface to Test

Here's a look at the State interface:

```
tinyos-2.x/tos/interfaces/State.nc

interface State {

  /**
   * This will allow a state change so long as the current
   * state is S_IDLE.
   * @return SUCCESS if the state is change, FAIL if it isn't
   */
  async command error_t requestState(uint8_t reqState);

  /**
   * Force the state machine to go into a certain state,
   * regardless of the current state it's in.
   */
  async command void forceState(uint8_t reqState);

  /**
   * Set the current state back to S_IDLE
   */
  async command void toIdle();

  /**
   * @return TRUE if the state machine is in S_IDLE
   */
  async command bool isIdle();

  /**
   * @return TRUE if the state machine is in the given state
   */
  async command bool isState(uint8_t myState);

  /**
```

```
   * Get the current state
   */
  async command uint8_t getState();

}
```

The State component is very simple - it's a state machine manager many modules in your system. The rules are this:

- Each module starts off in the S_IDLE state when the system boots.
- If we *request* a state change, the state of a module can only change if its current state is S_IDLE.
- We may *force* a state change, the state will change no matter what the current state is.
- Calling toIdle() will force the state back to S_IDLE, no matter what the current state is.
- We should be able to quickly find out what a module's state is by calling isIdle(), isState(<state>), and getState().

Test Architecture Considerations

Isolation vs. Integration Testing

A unit test typically isolates the functionality you want to test from everything else. This is different from an integration or functional test because those types of tests allow other parts of the system to exist and interact with the module directly under test. TUnit will allow you to construct any of these types of tests - isolation or integration - if you want.

Say you want to test the receive functionality of any radio stack, for example. A good isolation unit test will remove the radio and other functionality from the picture, leaving only the receive module functionality as a black box that provides some interfaces and uses some interfaces. Your test itself may *provide* the interfaces used by Module Under Test (MUT), allowing your test to control and verify how the MUT is interacting with other modules or hardware.

An integration test, on the other hand, will allow that receive module black box test to include the radio hardware. This allows you to test whether or not the receive module is interacting properly in the presence of external stimuli.

Natural Isolation

The StateC component is naturally isolated. If you look at StateC (http://tinyos.cvs.sourceforge.net/*checkout*/tinyos/tinyos-2.x/tos/system/StateC.nc?revision=1.3&content-type=text%2Fplain) and StateImplC (http://tinyos.cvs.sourceforge.net/*checkout*/tinyos/tinyos-2.x/tos/system/StateImplC.nc?content-type=text%2Fplain) , you'll notice that StateImplP (our Module Under Test) does not depend on any external functionality anywhere. In other words, it does not explicitly *use* an interface in any of its configuration files, and does not pull in other components in its configuration files to wire the module to.

If we had a component that did rely on other interfaces, we would stub out those dependencies to isolate the Module Under Test by *overriding* its configuration file and supplying our own. Either our test or some dummy module would provide the interfaces that need to be stubbed out, and our MUT would be completely isolated from the rest of the system.

Test Setup

You'll find this test in the tinyos-2.x-contrib/tunit/tests/tinyos-2.x/tos/system/TestStateC/ (http://tinyos.cvs.sourceforge.net/tinyos/tinyos-2.x-contrib/tunit/tests/tinyos-2.x/tos/system/TestStateC/) .

The State component and interface is in the TinyOS baseline. Therefore, we put the test under the tinyos-2.x baseline tests (tunit/tests/tinyos-2.x). The State component in the baseline exists in the tinyos-2.x/tos/system directory, so the test directory should follow a parallel mapping.

```
tinyos-2.x-contrib/tunit/tests/
|-- tinyos-2.x
|   |-- tos
|   |   |-- system
|   |   |   |-- TestStateC
```

TUnit Tip
*Prefix all test names and directories with the word **Test** for consistency.*

TestStateC Configuration File

We'll supply a configuration file for you to study below. Notice there are 4 different Test Cases we want to run, each with a somewhat descriptive name. Try building your own configuration file and running it in TUnit.

tinyos-2.x-contrib/tunit/tests/tinyos-2.x/tos/system/TestStateC/TestStateC.nc

```
  0|  configuration TestStateC {
  1|  }
  2|
  3|  implementation {
  4|
  5|    components new TestCaseC() as TestForceC,
  6|        new TestCaseC() as TestToIdleC,
  7|        new TestCaseC() as TestRequestC;
  8|
  9|    components TestStateP,
 10|        new StateC();
 11|
 12|    TestStateP.State -> StateC;
 13|    TestStateP.SetUp -> TestForceC.SetUp;
 14|
 15|    TestStateP.TestForce -> TestForceC;
 16|    TestStateP.TestToIdle -> TestToIdleC;
 17|    TestStateP.TestRequest -> TestRequestC;
 18|
 19|  }
```

Line 13 connects up a TestControl (http://tinyos.cvs.sourceforge.net/*checkout*/tinyos/tinyos-2.x-contrib/tunit/tos/interfaces/TestControl.nc?content-type=text%2Fplain) interface to the test. TestControl interfaces are *SetUpOneTime*, *SetUp*, *TearDown*, and *TearDownOneTime* which follow the TUnit Test Flow.

Every TestCaseC instance provides all TestControl interfaces for you to connect your test to. If your test requires some TestControl interfaces, just pick an instance of TestCaseC and specify exactly which TestControl interface you want. In this case, we arbitrarily picked the TestForceC component (an instance of TestCaseC) to provide *SetUp*.

You'll notice the TestControl interface itself is identical to the TestCase (http://tinyos.cvs.sourceforge.net/*checkout*/tinyos/tinyos-2.x-contrib/tunit/tos/interfaces/TestCase.nc?content-type=text%2Fplain) interface. This is only for ease of use - notice when we wire up actual TestCase's on lines 16 through 19, we don't specify that we want to wire to the TestCase interface. Compare this to line 13, where we have to specify *SetUp*.

TestStateP Module File

Below is the TestStateP module file for our test.

```
 0|  #include "TestCase.h"
 1|
 2|  module TestStateP {
 3|    uses {
 4|      interface State;
 5|
 6|      interface TestControl as SetUp;
 7|
 8|      interface TestCase as TestForce;
 9|      interface TestCase as TestRequest;
10|      interface TestCase as TestToIdle;
11|    }
12|  }
13|
14|  implementation {
15|
16|    /**
17|     * Possible states
18|     */
19|    enum {
20|      S_IDLE,
21|      S_STATE1,
22|      S_STATE2,
23|      S_STATE3,
24|    };
25|
26|    /***************** Prototypes ****************/
27|
28|    /***************** SetUpOneTime Events ***************/
29|    event void SetUp.run() {
30|      call State.toIdle();
31|      call SetUp.done();
32|    }
33|
34|    /***************** Tests ****************/
35|    event void TestForce.run() {
36|      call State.forceState(S_STATE1);
37|      assertTrue("TestForce: S_STATE1 not forced correctly", call State.getState() == S_STATE1);
38|      call State.forceState(S_STATE2);
39|      assertEquals("TestForce: S_STATE2 not forced correctly", call State.getState(), S_STATE2);
40|      assertTrue("TestForce: 1. isState() failed", call State.isState(S_STATE2));
41|
42|      call State.forceState(S_IDLE);
43|      assertFalse("TestForce: State was supposed to be idle", call State.getState() == S_STATE3);
44|      assertFalse("TestForce: 2. isState() failed", call State.isState(S_STATE2));
45|
46|      call TestForce.done();
47|    }
48|
49|    event void TestRequest.run() {
50|      call State.toIdle();
51|      call State.requestState(S_STATE1);
52|      assertTrue("TestReq: S_STATE1 not requested correctly", call State.getState() == S_STATE1);
53|      call State.requestState(S_STATE2);
54|      assertFalse("TestReq: S_STATE2 requested incorrectly", call State.getState() == S_STATE2);
55|      call State.requestState(S_IDLE);
56|      assertTrue("TestReq: S_IDLE not requested correctly", call State.isIdle());
57|
58|      call TestRequest.done();
59|    }
60|
61|
62|    event void TestToIdle.run() {
63|      call State.forceState(S_STATE3);
64|      assertTrue("TestToIdle: S_STATE3 not forced correctly", call State.getState() == S_STATE3);
65|      call State.toIdle();
66|      assertTrue("TestToIdle: toIdle() didn't work", call State.getState() == S_IDLE);
67|      assertTrue("TestToIdle: toIdle()/isIdle() didn't work", call State.isIdle());
68|
69|      call TestToIdle.done();
70|    }
71|
72|  }
```

On line 29, we get a signal to run the SetUp procedure, which simply sets our State back to IDLE before each test is run. As with all other interactions with TUnit, line 31 tells TUnit the SetUp is done().

Each TestCase is run() one after the other. The only place we can make assertions is after a TestCase is run(), and before the callback with done(). Read through the test and understand what it's doing.

This test can form the basis of creating the StateC component's functionality from scratch by defining its expected behavior and requirements, or it can be used to make sure we don't break the StateC component down the road.

Makefile

Each test needs a Makefile to compile.

tinyos-2.x-contrib/tunit/tests/tinyos-2.x/tos/system/TestStateC/Makefile

```
COMPONENT=TestStateC
include $(MAKERULES)

```

suite.properties

A suite.properties file is always optional. In this case it will save us time by filtering out Test Run's that will result in redundant testing - specifically test runs containing more than one node. Being an isolation test, this test cannot interact with the radio or any other node.

tinyos-2.x-contrib/tunit/tests/tinyos-2.x/tos/system/TestStateC/suite.properties

```
/**
 * Valid keywords are:
 *  @author <optional author(s)>  (multiple)
 *  @assertions <max number of assertions expected in a single TestCase>
 *  @testname <optional testname>  (once)
 *  @description <optional, multiline description>  (once)
 *  @extra <any build/install extras> (multiple)
 *  @ignore <single target>  (multiple)
 *  @only <single target> (multiple)
 *  @minnodes <# nodes>  (once)
 *  @maxnodes <# nodes>  (once)
 *  @exactnodes <# of exact nodes>  (once)
 *  @mintargets <# of minimum targets for heterogeneous network testing>  (once)
 *  @timeout <timeout duration of the test in minutes, default is 1 min.>
 *  @skip  (once)
 */

@exactnodes 1

```

TUnit there cannot be any spaces before @ symbols for TUnit to recognize the rules you create.

Execute

After putting all these files into the same directory (tinyos-2.x-contrib/tunit/tests/tinyos-2.x/tos/system/TestStateC (http://tinyos.cvs.sourceforge.net/tinyos/tinyos-2.x-contrib/tunit/tests/tinyos-2.x/tos/system/TestStateC/)) we can run it:

```
$ tunit

...

T-Unit Results
----------------------------------------------
Total runtime: 42.844 [s]
Total tests recorded: 12
Total errors: 0
Total failures: 0
```

Notice if you try to alter the behavior of the State component in any way, the test will begin to fail and tell you what's going wrong which would help you debug the problem.

See Also

- TUnit
- Single-Node Unit Testing
- TUnit Test Flow
- suite.properties

Case Studies

- TestRxFifo (http://tinyos.cvs.sourceforge.net/tinyos/tinyos-2.x-contrib/tunit/tests/tinyos-2.x-contrib/blaze/tos/chips/blazeradio/CC2500/receive/TestRxFifoReceive/)
  - A test isolating the receive functionality of a radio stack from the rest of the system, allowing us to test edge cases observed by receiving various types of data in the software implemented RX FIFO.

- AckBehavior (http://tinyos.cvs.sourceforge.net/tinyos/tinyos-2.x-contrib/tunit/tests/tinyos-2.x-contrib/blaze/tos/chips/blazeradio/shared/acks/AckBehavior/)
  - Isolation test of the acknowledgements layer of the CC1100/CC2500 dual radio stack.

- CsmaBehavior (http://tinyos.cvs.sourceforge.net/tinyos/tinyos-2.x-contrib/tunit/tests/tinyos-2.x-contrib/blaze/tos/chips/blazeradio/CC2500/csma/CsmaBehavior/)
  - Isolation test of the CSMA layer of the CC2500 radio stack.

Next

- Characterization Testing with Statistics

Retrieved from "http://tinyos.stanford.edu/tinyos-wiki/index.php?title=State_Interface_Test&oldid=2825"

- This page was last modified on 23 November 2009, at 04:25.
- This page has been accessed 19,502 times.