Conformance Test Experiments for Distributed Real-Time Systems

(This is a long version of the paper to appear in ISSTA’02)

Rachel Cardell-Oliver
Dept Computer Science & Software Engineering
The University of Western Australia
35 Stirling Hwy, Crawley WA 6009
+61 8 9380 2231
rachel@cs.uwa.edu.au

ABSTRACT
This paper introduces a new technique for testing that a distributed real-time system satisfies a formal timed automata specification. It outlines how to write test specifications in the language of Uppaal timed automata, how to translate those specifications into program code for executing the tests, and describes the results of test experiments on a distributed real-time system with limited hardware and software resources.

Categories and Subject Descriptors
D.2.4 [Software Engineering]: Software/Program Verification Formal methods, Validation D.2.5 [Software Engineering]: Testing and Debugging Testing tools (e.g., data generators, coverage testing)

General Terms
Performance, Design, Reliability, Experimentation, Verification

Keywords
distributed real-time systems, conformance testing, design for testability, timed automata, Uppaal, Lego RCX

1. INTRODUCTION
Our goal is to design and execute tests to detect faults in distributed real-time software implemented with limited hardware and software resources. The desired behaviour of a system under test (SUT) is described using formal timed automata specifications. These test specifications are then translated to program code for executing tests against an implemented system. The tests determine whether the behaviour of the system under test (SUT) matches the intended behaviour.

Testing distributed real-time systems presents a number of practical difficulties which are not addressed by previous theoretical work on test generation for real-time and distributed systems [2,4,6,9,11,13]. For example, in order to test distributed real-time systems, a tester must control and observe real-time events from different sources. Problems of clock synchronisation arise in trying to control and interpret events in this environment. Another problem, particularly for systems with limited resources, is the probe effect in which the timing behaviour of the SUT is disturbed by the activity of recording events for test purposes. Additionally, the test program or the SUT may be too slow or inaccurate for the control or observation of certain types of events and this can cause misleading test results, either false positives or false negatives [12]. This paper describes a test technique together with test experiments which address these issues.

Section 2 outlines the test technique used in this paper, and Section 3 describes a typical distributed real-time system which is used as running example for the paper. Section 4 explains how tests are specified in the formal language of Uppaal timed automata. Section 5 describes our implementation environment and Section 6 how to translate test specifications into test programs. Three test experiments are described in Section 7.

2. TEST TECHNIQUE
The test technique presented in this paper is based on finite state machine conformance test methods [4,5,9]. The desired behaviour of a system under test (SUT) is given by a formal Uppaal timed automata specification [8]. The tester generates test cases from the formal specification and executes them to determine whether the behaviour of the SUT matches this desired behaviour. Our main test purpose is to find defects in the SUT, and so test cases are chosen to exercise, as far as possible, parts of the system which could exhibit defects. Our test technique can also be used for reliability testing, in which the tester observes “normal” activity with the purpose of increasing confidence that the system reliably meets its specification, but in this paper our focus is on defect testing.

A defect is defined as any SUT deviation in timing or value from the expected behaviour described by an Uppaal timed automata (UTA) specification. In our test method, the value and timing of all inputs relevant to a test purpose is defined explicitly and completely by its UTA specification. That is, test specifications are input closed. All outputs which constitute correct response to the inputs are also
explicitly defined by the UTA test specification. An incorrect output, known as a fault or a defect, is any output which does not match the value and timing explicitly specified in the behaviour. That is, we define defects implicitly rather than explicitly as in [6].

3. EXAMPLE SYSTEM UNDER TEST

Our test method is designed for distributed real-time systems consisting of a set of tasks communicating by shared variables and messages, distributed over two or more processors. The example system used in this paper, adapted from Braberman and Felder [3], is typical of this class of systems. It consists of two processors which monitor inputs from several sources, analyse them and control a valve output which can be either opened or closed.

The first processor collects and manages data from two light sensors. It runs 4 tasks, in the following order of priority:

1. [s1] Monitor input from light sensor 1 every 40cs and store the reading in variable s1.
2. [a] Calculate the average of the currently stored light sensor readings. If the average has just risen above 50% (light sensor on yellow Lego brick) then send message 2 to processor 2. If the light average has just fallen below 35% (light sensor on black Lego brick) then send message 1 to processor 2.
3. [s2] Monitor input from light sensor 2 every 60cs and store the reading in variable s2.
4. [d] Display the average light level every 80cs.

The second processor monitors user input from a touch sensor button and messages sent by processor 1, and uses a micro-motor to open and close a valve. It runs 2 sporadic tasks and one periodic task in the following order of priority:

1. [vm] Whenever a message is received, update the local state (s=2 set when light is high and s=1 set when light is low). If the background light has just become high (s==2) then open the valve for 15cs. The deadline for this task is 40cs, the minimum separation between messages.
2. [cm] Monitor the button queue every 30cs. Clear a button press from the queue if one is waiting: b=b-1. If s==1 and there was an outstanding press then close the valve for 2cs. If s==2 then the background light is high and the valve is not moved.
3. [cc] Whenever the button is pressed, update the queue variable: b=b+1. The deadline for this task is 15cs, the minimum separation between two button presses.

4. SPECIFYING TESTS WITH UPPAAL TIMED AUTOMATA

To specify real-time tests we use the formal language of timed automata, specifically the class, UTA, of timed automata with persistent integer variables as supported by the Uppaal tool [8]. A specification in this model is a network of communicating sequential finite state machines with clock variables, synchronous communication and persistent integer variables.

Specification clocks correspond to the real external time of the system as measured by the program clocks in the tester or system under test. In our environment, clocks measure time in centiseconds (cs), that is at 100 ticks per second. The identifiers x,y,z are used for clock variables.

Integer variables of the specification correspond to persistent variables such as program variables and the values held in message buffers. The identifiers s, b etc. are used for integer variables and m for the content of the message buffer.

Synchronous communication in a UTA specification corresponds to memoryless operations of a SUT such as the act of sending a message or pressing a button. Such events can only be captured by observing them as they happen. Identifiers button? and done! denote input and output events, respectively from the point of view of the system under test. That is, button? is an input for the system under test, but the tester itself provides that value by outputting it. Similarly, the tester waits for event done! which is output by the system under test. In test specifications we show only one side of such communications, but for each one the SUT is assumed to have a matching synchronisation event.

A test is defined by a UTA specification which describes the inputs to be offered to the system under test and the outputs expected. Test behaviours can be expressed at many different levels of abstraction because UTA is a general language. Choosing the right level of abstraction is an important skill of the test designer. Our tests are usually specified as non-terminating processes. They are intended to be run many times with the intention of providing sufficient evidence to overcome problems of inaccuracy in controlling test executions.

4.1 Observing External Events

Example 1. Inputs to the light sensors are specified to change from high (61) to low (35) and then low to high every 100cs. When the light changes to high clock y is reset to 0. At the same time the tester’s message buffer is continuously monitored. All incoming messages are recorded but we are specifically interested in message 20 which signals that the valve has begun to open. A fault is detected if message 20 is not observed whilst clock y<=60.
Example 2. Button presses may occur at most every 15cs with the additional constraint that there are no more than 2 presses in any 60cs interval. The tasks of processor 2 maintain a queue of outstanding requests denoted by a variable b with $b=0$ for the empty queue, $b=1$ with one press queued and $b=2$ when the queue is full. If the queue overflows, a sound is output. The desired behaviour is for b to remain in range 0 to 2, never exceeding the value 2. However, the value of b can not be monitored directly by any tester. In Section 7.2 we show how to monitor the desired condition indirectly by monitoring the sound output. In the given test specification presses are offered at 15cs and then 45cs intervals. In another version of the test, presses are offered every 30cs.

4.2 Observing Internal Events
Examples 1 and 2 are black box tests. They describe properties of the system under test which can be observed without knowledge of the way in which the system has been implemented. We now consider testing that the each of the tasks of the system implementation meet their deadlines. The completion of a system task can not be observed externally and so probes must be added to the code of the system under test to record significant events such as task completion. The tester’s task is to analyse the recorded results and check that each task meets its deadline. This is necessarily done after completion of the test because events are recorded in a datalog which is only readable by uploading from the system under test when it has finished execution. The tests can be run under differing conditions by varying the rate of input events such as button presses and changes in light value.

Example 3. The correct behaviour of a periodic process with period $P_c$s, completion event `done!` and deadline $D \leq P$ can be tested by the following timed automaton.

The correct behaviour of a sporadic process triggered by event `trigger?` and completed with event `done!` before $D_c$s after that event can be tested by the following timed automaton.

4.3 Observing Everything
The language of UTA can be used to specify fully the implemented behaviour of the system including the behaviour of each task and the scheduling strategy employed. However, such specifications turned out to have limited use for test generation because execution times for individual commands in a task and also the system timing used by the implementation scheduler could not be captured sufficiently accurately to make the model’s timing accurate. There are also too many possible behaviours to be tested, even though many of the differences between behaviours are irrelevant because, for example, they are caused by different interleavings of essentially independent tasks.
5. IMPLEMENTATION ENVIRONMENT

In our experiments we have used an implementation environment based on a collection of Lego RCX processors [1]. The RCX is an autonomous programmable microcomputer, using a Hitachi H8/3932 micro-controller. Hitachi H8-series “upwardly-compatible 8-bit and 16-bit micro-controllers (MCUs) are among the industry's most popular embedded system solutions (number 2 in sales, worldwide)” [7]. The RCX brick can be used to control actuators, including a sound generator, lights, and motors, and read input from sensors, including light, pressure, rotation, and temperature sensors. RCX processors communicate by broadcasting messages over an infrared channel. The RCX firmware provides support for monitoring events, reading and setting clocks and simple programming language support with integer variables. NQC [1] is a programming language for the Lego RCX which supports multi-tasking, event monitoring, clocks and integer variables. Up to 4 system clocks may be read and set with 1/100 second accuracy. Parallel tasks are executed by a round robin scheduler with (we estimate) a 0.6cs time slice. Tasks may monitor one or more events, jumping to an event handler whenever the event occurs. This simple mechanism allows events to be lost if they occur during event handling, but it has proved sufficient for our purposes because our event handlers are fast in comparison with minimum event separation. In future work we plan to experiment with operating systems such as the LegOS real-time kernel [10] for the RCX which offers pre-emptive multitasking or the Tiny operating system designed for next generation distributed systems with thousands of processors, each with limited power and processing resources.

6. EXECUTING TESTS

The UTA specification examples in Section 4 each describe a behaviour we would like the SUT to exhibit and describe a situation which we believe is likely to contain implementation faults. In order to execute these tests, we must associate the events of the specification language (channels and assignments) with actual events of the implemented system. Clock operations of the UTA specifications are represented by NQC clocks.

Ideally, we wish to test a distributed real-time system in such a way that the test experiments themselves do not change the behaviour of the system under test, that is, to avoid the probe effect. This is to reduce the risk of failures being masked or caused by the test process itself.

For defect testing we aim to test the system under extreme input conditions and such tests require the active control of system inputs. In the examples in this paper all inputs can be actively controlled externally. However, there are some test cases which require internal control of inputs. For example, replace “read sensor” commands with an assignment statement for the desired input value. We can also use passive inputs from the external environment for testing. For example, a human tester can control the light level or button presses by hand.

Observing outputs is necessarily passive because the tester waits for an output and then records it. Outputs can either be recorded by the system under test, if internal events, or by the tester, if externally visible events such as messages. Recorded outputs are analysed after test execution by uploading and analysing a datalog of recorded results from the SUT RCX(s). Externally visible outputs may be analysed on the fly during test execution by checking that the time and value of an output is as expected as soon as it is observed. Table 1 outlines some of the techniques we have used for controlling and observing possible SUT events for NQC programs on Lego RCX processors. The full table appears in an Appendix.

Table 1. Techniques to Control or Observe SUT Events

<table>
<thead>
<tr>
<th>SELECTED SUT EVENTS</th>
<th>TECHNIQUE FOR CONTROL OR OBSERVATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>light sensor input</td>
<td>A lamp is placed in front of the light sensor with a wire attached to an actuator port. When maximum output is sent to the actuator port the lamp light causes sensor reading 61%. The sensor reads 31% when the lamp is off.</td>
</tr>
<tr>
<td>touch sensor input</td>
<td>The input port used by the SUT for the touch sensor is connected to a tester's actuator port. Sending the actuator OnFloat mimics depressing the sensor, and sending Off mimics releasing the sensor.</td>
</tr>
<tr>
<td>message output</td>
<td>All messages are broadcast over the RCX IR channel. The tester can monitor all messages sent and received by the SUT</td>
</tr>
<tr>
<td>motor output</td>
<td>Each change of state (turning motor on forward or reverse) must be marked in the SUT either by the broadcast of a suitable message which can be observed externally as above, or by recording the change in the system datalog. Motors can not be observed by piggybacking an observation wire to the motor because this drains the power needed to move the motor.</td>
</tr>
<tr>
<td>global variables</td>
<td>Record the value and time of change of global variables of interest in the datalog for analysis after test execution</td>
</tr>
<tr>
<td>task completion</td>
<td>Record in the datalog the name of a task or subroutine and its time of calling for analysis after test execution</td>
</tr>
</tbody>
</table>
7. THREE TEST EXPERIMENTS

7.1 Test Experiment 1.

Purpose: To test that whenever the light level is held high the valve will start opening within 60cs.

UTH Specification: Example 1, Section 4.1

Test Inputs: The SUT's light sensors 1 and 2 are controlled by tester signals sent to lamps placed next to the light sensors (see Table 1). The light level changes from high to low or low to high every 100cs.

Test Outputs: We can not observe the valve motor directly (see Table 1) and so the SUT code is updated so that a message is sent each time the valve motor is activated: message 20 is sent when the motor begins opening the valve and 10 when it begins to close.

Recorded: The tester records each change of light level between high and low light, and all messages sent between processors 1 and 2.

Results: From the test record we extracted the times of change from low to high light and when message 20 was observed. Figure 1 shows the delays between the light changing and valve activation. The variation in response time is the result of differences in the time it takes for both sensor reading tasks to record the changed light level.

![Figure 1. Response Time Test Results](image)

7.2 Test Experiment 2.

Purpose: Check that even when button press inputs are offered at the maximum allowed rate, the queue for button presses is always serviced sufficiently fast.

UTH Specification: Example 2, Section 4.1

Test Inputs: Button presses are controlled by sending signals to the touch sensor input port on processor 2 (see Table 1). Button presses are sent at 15cs then 45cs intervals. In a second test, presses were sent every 30cs.

Test Outputs: We wish to observe the queue variable b continuously to check that the queue never overflows but we can not observe this condition directly. However, the SUT plays a sound should the buffer variable ever exceed 2 and so it is possible to monitor the desired condition indirectly: if no sound is played during the test, then the buffer is being serviced sufficiently fast. We do not need to observe when the queue variable is incremented and decremented.

Results: For both the 15-45 test and the 30-30 test, no sound is played, showing that the buffer is being serviced correctly. When we increased the period of the queue servicing task (cm) from 30cs to 31cs queue overflow did occur and the sound was played for each overflow.

7.3 Test Experiment 3.

Purpose: To test that every task in the SUT meets its deadline.

UTH Specification: Example 3, Section 4.2 with task deadlines and triggers as specified in Section 3.

Test Inputs: Button inputs on processor 2 are offered at the maximum rate at 30cs intervals as in Test Experiment 3. Light inputs on processor 1 are passive inputs from the environment.

Test Outputs: Task calls can not be observed externally, and so we instrument the SUT code on each processor for this test, with each task recording a task identifier and completion time.

Results: Figure 2 shows the completion times of tasks on each processor for a typical 240cs time slice from the full test datalog of 7920cs. The task identifiers of the vertical axis are for processor 1, s1=1, a=2, s2=3, d=4 and for processor 2, vm=5, cm=6 and cc=7. The test results show that all tasks meet their deadlines and sporadic tasks respond correctly to their triggers. Although we have plotted the data
from both processors on the same graph, it should be noted that each processor records data using its own local clock. Thus, in Figure 2 the times between tasks on the same processor are accurate, but the interleaving of processor 1 and 2 tasks as shown may be inaccurate.

**Figure 2. Task Scheduling Results**

*Observations:* The implementation of processor 1 tasks uses a static cyclic executive for which all task deadlines are easily satisfied during each 240cs cycle with a processor utilisation around 12%. Interestingly, in experiments using a different scheduling strategy with independent NQC tasks for each task, a scheduling fault was detected with task s2 missing its deadline at 180cs in each cycle.

8. DISCUSSION

This paper introduces a new technique for testing that a distributed real-time system satisfies a formal timed automata specification. It outlines how to write test specifications in the language of UTA, how to translate those specifications into program code to execute the tests, and describes the results of experiments testing a typical distributed real-time system implemented with limited hardware and software resources. Further details of this technique and our experiments are available at http://www.cs.uwa.edu.au/~rachel/.

Previous methods for generating test cases for distributed and real-time systems [2,4,5,6,9,11,13] have not addressed test execution issues such as the limitations of input control and output observation. For example, although most test generation methods specify a test case as a particular sequence of inputs and outputs, our experiments using limited hardware and software resources showed that it was often not possible to control exactly one expected sequence of timed input and output events because of non-determinism introduced by task scheduling and our imprecise knowledge of the execution time of commands. Instead, we specified input and output behaviours as parallel processes, and used program clocks and off-line analysis to check that all required timing properties were satisfied in the observed behaviours. Also, instead of running each possible test sequence one time, we ran test programs which perform the same test multiple times, with the intention of providing sufficient evidence to overcome problems of inaccuracy in controlling test executions.

Peters and Parnas [12] have studied observation problems for real-time systems for reliability testing using passive requirements-based monitors. In our experiments we consider also test execution problems but with active testers as required for defect testing. Thane [13] introduces test techniques for deriving all possible execution orderings for deterministic distributed real-time systems. In our model non-deterministic systems can also be tested because we do not rely on a single sequential program as test specification, but specify tests using UTA parallel processes.

The practical problems we have identified highlight the importance of taking testing issues into account at the design stage. For example, designers may need to identify which outputs are to be observed externally, and decide how this can be achieved, for example, by allowing a more generous task schedule to avoid probe effect problems.

9. REFERENCES


10. APPENDIX

**Methods for Controlling or Observing NQC events**

<table>
<thead>
<tr>
<th>SUT EVENT</th>
<th>TECHNIQUE FOR CONTROL OR OBSERVATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>light sensor input</td>
<td>a lamp is placed in front of the light sensor with a wire attached to an actuator port. When maximum output is sent to the actuator the lamp causes sensor reading 61% with 31% when the lamp is off</td>
</tr>
<tr>
<td>light sensor input</td>
<td>allow sensor inputs by hand using brick colours to trigger different light levels. Sensor reading for black brick is 35% and for yellow brick 50%</td>
</tr>
<tr>
<td>light sensor input</td>
<td>Replace program calls to read the sensor (e.g. s1=SENSOR_1) with assignments of desired values in the range 0 to 100 (e.g. s1=45)</td>
</tr>
<tr>
<td>touch sensor input</td>
<td>The input port used by the SUT for the touch sensor is connected to a tester's actuator port. Sending the actuator OnFloat represents depressing the sensor, and sending Off represents releasing the sensor.</td>
</tr>
<tr>
<td>message input</td>
<td>connect a wire from the touch sensor of the SUT to an sensor port of the tester and register a copy of each touch sensor event in the system under test</td>
</tr>
<tr>
<td>message output</td>
<td>All messages are broadcast over the RCX IR channel. The tester can send its own messages to the SUT as necessary</td>
</tr>
<tr>
<td>lamp output</td>
<td>wire from lamp to tester sensor reads X when lamp on and Y when lamp off</td>
</tr>
<tr>
<td>motor output</td>
<td>attach a light sensor near the lamp to a light sensor on the tester and record the brightness of the lamp in range 0 to 100</td>
</tr>
<tr>
<td>global variables</td>
<td>record the value and time of change of global variables of interest in the datalog for analysis after test execution</td>
</tr>
</tbody>
</table>

**Overview of Design and Test Method**

- distribute real-time system design
- revise design
- timed automata model
- feedback from test experiments
- generate test cases
- run tests
- NQC implementation
- NQC test program

---


internal observation
task invocation and subroutine calls
internal observation

sound output
passive external observation
observed passively, useful for flagging when an error occurs and can be used for ongoing monitoring of the SUT after delivery

---

Full Test Results (combined) from Test Experiment 3

<table>
<thead>
<tr>
<th>Time (cs)</th>
<th>Processor tasks completed</th>
<th>1 cm</th>
<th>&amp;</th>
<th>2 Light Status (s)</th>
<th>Queue (b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td>s1,a,s2,d</td>
<td></td>
<td></td>
</tr>
<tr>
<td>34</td>
<td></td>
<td></td>
<td>cm</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>43</td>
<td></td>
<td></td>
<td>s1,a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>61</td>
<td></td>
<td></td>
<td>s2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>62</td>
<td></td>
<td></td>
<td>cm</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>75</td>
<td></td>
<td></td>
<td>cc</td>
<td></td>
<td></td>
</tr>
<tr>
<td>84</td>
<td></td>
<td></td>
<td>s1,a,d</td>
<td></td>
<td></td>
</tr>
<tr>
<td>91</td>
<td></td>
<td></td>
<td>cm</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>109</td>
<td></td>
<td></td>
<td>cc</td>
<td></td>
<td></td>
</tr>
<tr>
<td>122</td>
<td></td>
<td></td>
<td>cm</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>124</td>
<td></td>
<td></td>
<td>s1,a,s2</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>142</td>
<td></td>
<td></td>
<td>cc</td>
<td></td>
<td></td>
</tr>
<tr>
<td>155</td>
<td></td>
<td></td>
<td>cm</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>164</td>
<td></td>
<td></td>
<td>s1,a,d</td>
<td></td>
<td></td>
</tr>
<tr>
<td>175</td>
<td></td>
<td></td>
<td>cc</td>
<td></td>
<td></td>
</tr>
<tr>
<td>181</td>
<td></td>
<td></td>
<td>s2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>187</td>
<td></td>
<td></td>
<td>cm</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>199</td>
<td></td>
<td></td>
<td>vm</td>
<td></td>
<td></td>
</tr>
<tr>
<td>203</td>
<td></td>
<td></td>
<td>s1,a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>207</td>
<td></td>
<td></td>
<td>cc</td>
<td></td>
<td></td>
</tr>
<tr>
<td>218</td>
<td></td>
<td></td>
<td>cm</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>240</td>
<td></td>
<td></td>
<td>cc</td>
<td></td>
<td></td>
</tr>
<tr>
<td>248</td>
<td></td>
<td></td>
<td>s1,a,s2,d</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

System Under Test NQC code

```c
// valve manager processor 1
// rco 24/iv/02 - some debug removed

// ...
..\..\..\nqc -Susb -Trcx2 -d processor1.24.4.02nqc

// #define YELLOW 50-2 // light sensor reading for yellow colour - error
#define BLACK 35+2  // light sensor reading for black + error
#define GONEHIGH 2;
#define GONELOW 1;

int  gs1=BLACK, gs2=BLACK,
gs=BLACK, current=YELLOW,     // so initially send gone low
display=BLACK;                  // global shared memory variables

sub Sensor1() { 
gs1=SENSOR_1;            // write input 1 to shared memory
```

sub Sensor2() {
    gs2=SENSOR_3; // write input 2 to shared memory
}

sub Analyser() {

    //old debug
    //if (current==BLACK) PlayTone(440,1);
    //else if (current==YELLOW) PlayTone(550,1);
    //else PlayTone(770,1);

    gs=(gs1+gs2)/2; // calc average sensor vals

    if ((gs>=YELLOW) && (current==BLACK))
        { SendMessage(2); current=yellow; } // send GONEHIGH state change
    else if ((gs<=BLACK) && (current==YELLOW))
        { SendMessage(1); current=BLACK; } // send GONELOW state change

    // remembering last=gs each call didn't work - miss changes
}

sub Displayer() {
    display=gs;
}

task main() {

    SetSensor(SENSOR_1, SENSOR_LIGHT); // input light 1
    SetSensor(SENSOR_3, SENSOR_LIGHT); // input light 2
    SetUserDisplay(display,0); // output display
    // output message also

    while(true) {
        ClearTimer(0);
        Sensor1(); Analyser(); Sensor2(); Displayer();

        until (Timer(0)>=4); // = 40 centiseconds
        Sensor1(); Analyser();

        until (Timer(0)>=6); // = 60 centiseconds
        Sensor2();

        until (Timer(0)>=8); // = 80 centiseconds
        Sensor1(); Analyser(); Display();

        until (Timer(0)>=12); // = 120 centiseconds
        Sensor1(); Analyser(); Sensor2();

        until (Timer(0)>=16); // = 160 centiseconds
        Sensor1(); Analyser(); Display();

        until (Timer(0)>=18); // = 180 centiseconds
        Sensor2();
    }
}
until (Timer(0)>=20); // = 200 centiseconds
Sensor1(); Analyser();

until (Timer(0)>=24); // = 240 centiseconds
}

// valve manager processor 2
// rco 24/iv/02
// updated so motor turned to Rev sends a message 10 and motor turned to FWd sends message 20
// ..\..\nqc -Susb -Trcx2 -d processor2.24.4.02.nqc

int s=1, b=0, // status and queue variables
   m=0, g=1; // incoming message and gate (open=1 or shut=0)

sub ValveManagerResponse() {
   m=Message();
   if (m>0) { s=m; PlayTone(440+50*m,1); } // capture latest gone high or gone low change
   if (s==2) // light has gone high
      { OnFwd(OUT_A);
        SendMessage(20); // let tester know motor is set fwd
        Wait(15);
        Off(OUT_A);
      } // else no change to valve
}

task CommandManager() {
   while (true) {
      if (b>0) { // if button pushes to service then
         b--;
         if (s==1) // if valve most recently went low (default)
            // to be checked: why/if signals are lost
            { OnRev(OUT_A);
              SendMessage(10); // let tester know motor reversing
              OnFor(OUT_A,2); }
            // then reverse a little
      }
      Wait(27); //=27/100s makes average period 30
      // original 29 causes buffer overflow for 15+45 pattern
      // but 23 quite different no overflow but xxxx-xxxx- with gaps
      // between groups of 4
      // 24 similar xxxxxx-xxxxx- and xxxx-xxxxxxx-xxxxxxx- again no overflow
      // 25,26,27 similar longer xx sequences and no overflow
   }
}

sub CommandCaptureResponse() {
   b++;
   if (b>2) PlayTone(300,2); // error: queue overflow
}
task main() {

SetSensor(SENSOR_3, SENSOR_TOUCH); // user button input

SetEvent(1, Message(), EVENT_TYPE_MESSAGE);

SetEvent(3, SENSOR_3, EVENT_TYPE_PRESSED);
SetPower(OUT_A, OUT_HALF);
ClearTimer(0);

start CommandManager; // run periodic task, period=30/100 s

while (true) {
    monitor (EVENT_MASK(1)|EVENT_MASK(3)) {
        Wait(1000);
    }
    catch (EVENT_MASK(1)) { // priority 1, separation at least 40/100 s
        ValveManagerResponse();
    }
    catch (EVENT_MASK(3)) { // priority 3: separation at least 15/100 and
        // no more than 2 pushes in 60cs
        CommandCaptureResponse();
    }
}

Test Programs for the Valve Manager

// FINAL VERSION USED
// TEST PROGRAM 3 for PROC 1 || PROC 2 response to light changes
// VERSION rco 24/iv/02
// RUN ..\..\nqc -Susb -Trcx2 -d test2.24.4.02.nqc
// SAVE TEST(N) ..\..\nqc -Susb -datalog > test2.24.4.02.log.txt
//
// DESCRIPTION
// this program checks response time of output to high lights with
// slower changes than before
// but does not make any assumts about order - in one task change light
// level and in parallel monitor all messages
// again ignore button pushes
// processor1.24.4.02.nqc || processor2.24.4.02.nqc
//
// TEST OUTPUTS connected to SUT INPUTS
// OUT_A a light facing a light sensor attached to proc1 SENSOR3
// OUT_C a light facing a light sensor attached to proc1 SENSOR1
//
// TEST INPUTS connected to SUT OUTPUTS
// use messages to monitor the micro motor on OUT_A
//
// ASSUMPTIONS ABOUT THE ENVIRONMENT
// button inputs are being sent BY HAND
// see button test for max input rate test (2ce per 60)

// test with separation 60 results in av. light readings (48, 61, 36)*

// TO DO - perhaps add stop when datalog full

int i=0; // counts entries in datalog
void initsetlight(const int motID) {
void setlighthigh(const int motID) { // scheme from lugnet.robotics.rcx.ngc
ewsgroup
SetPower(motID, OUT_FULL);
on(motID);
}

void setlightlow(const int motID) {
// place sensor on a base bar (eg.2x8) and light on top of black connector
// the red light on the sensor needs to be partly covered or you can't get
// low enough low light readings
// ALSO need a black brick full hood or low enough readings (<37) can't be got
SetPower(motID, 0);
off (motID);
}

// FAILED Attempt to monitor SUT motor output directly 24/4/02
// set up a wire from the RCX output port to micro motor to measure the val sent
// connect the micromotor to a sensor set up as follows
// rotation and light sensors actually turns the motor - active modes
// sensor mode none and touch is passive
// NONE of these measure motor output so need to update SUT to send, say a message too

void record(int thetime, int theval) {
AddToDatalog(thetime);
AddToDatalog(theval);
}

task dolightinput() {
while (i<=400) {
    wait(100);
    setlighthigh(OUT_C); // monitored each 60cs
    setlighthigh(OUT_A); // monitored each 40cs
    record(FastTimer(0), 2000); i++;
    wait(100);
    setlightlow(OUT_C); // monitored each 60cs
    setlightlow(OUT_A); // monitored each 40cs
    record(FastTimer(0), 1000); i++;
} // end while
} // end task

task monitormessages() {
while (i<=400) {
    monito (EVENT_MASK(1)) {
        wait(1000);
    } // end while
} // end task
task main()
{
    // initialisation of inputs and outputs
    initsetlight(OUT_A);
    setlightlow(OUT_A);

    initsetlight(OUT_C);
    setlightlow(OUT_C); // initially light input is low

    SetEvent(1,Message(),EVENT_TYPE_MESSAGE);

    CreateDatalog(800);

    ClearMessage(); // message monitor ready to receive
    ClearTimer(0);

    start monitormessages;
    start dolightinput;
}
}

Test of Maximum Rate Button Inputs
// test processor1 || processor2 for queue overflow
// by sending button pushes at max rate
// rco 16/iv/02
// ..\nqc -Susb -Trcx2 -d testbuttons.nqc

void initsensout(const int motID) {
    Float(motID);
}

void dopress(const int motID) { // scheme from lugnet.robotics.rcx.nqc newsgroup
    Off(motID);
    Wait(2);
    Float(motID);
}

// sending press inputs

// case 1: closest possible, max pushes in
// separation at least 15/100 seconds (= min separation of sensor inputs)
// no more than 2 per 60/100 seconds
// case 3: inputs with longer separation 100

task main()
{
initsensout(OUT_A);

//case 2
repeat (100) {
dpress(OUT_A);
Wait(30);
}
Wait(300);

//case 1
repeat (50) {

dopress(OUT_A);
Wait(15);
dopress(OUT_A);
Wait(45);
}


Version of Processor 1&2 Instrumented to Record Task Completion Times

// valve manager processor 1
// rco 16/iv/02

// nqc -Susb -Trcx2 -d processor1.16.4.02nqc

#define YELLOW 50-2 // light sensor reading for yellow colour - error
#define BLACK 35+2 // light sensor reading for black + error
#define GONEHIGH 2;
#define GONELOW 1;

int gs1=BLACK, gs2=BLACK,
gs=BLACK, current=BLACK;

// so initially send gone low

display=BLACK; // global shared memory variables

void record(int thetime, int theval) {

AddToDatalog(thetime);
AddToDatalog(theval);
}

sub Sensor1() {

gs1=SENSOR_1; // write input 1 to shared memory
}

sub Sensor2() {

gs2=SENSOR_3; // write input 2 to shared memory
}

sub Analyser() {

// debug to check signalling state change
//if (current==BLACK) PlayTone(440,1);
//else if (current==YELLOW) PlayTone(550,1);
//else PlayTone(770,1);


gs=(gs1+gs2)/2; // calc average sensor vals
if ((gs>=YELLOW) && (current==BLACK))
    { SendMessage(2); current=YELLOW;} // send GONEHIGH state change
else if ((gs<=BLACK) && (current==YELLOW))
    { SendMessage(1); current=BLACK; } // send GONELOW state change
    // remembering last=gs each call didn't work - miss changes

sub Displayer()
{
    display=gs;
}

task main() {
    SetSensor(SENSOR_1, SENSOR_LIGHT); // input light 1
    SetSensor(SENSOR_3, SENSOR_LIGHT); // input light 2
    setUserDisplay(display,0); // output display
    // output message also
    CreateDatalog(900); // captures 50 (9*2*50) scheduler cycles

while(true) {
    ClearTimer(0);
    Sensor1(); Analyser(); Sensor2(); Displayer();
    record(00,FastTimer(0));

    until (Timer(0)>=4); // = 40 centiseconds
    Sensor1(); Analyser();
    record(40,FastTimer(0));

    until (Timer(0)>=6); // = 60 centiseconds
    Sensor2();
    record(60,FastTimer(0));

    until (Timer(0)>=8); // = 80 centiseconds
    Sensor1(); Analyser(); Sensor2();
    record(80,FastTimer(0));

    until (Timer(0)>=12); // = 120 centiseconds
    Sensor1(); Analyser(); Sensor2();
    record(120,FastTimer(0));

    until (Timer(0)>=16); // = 160 centiseconds
    Sensor1(); Analyser(); Sensor2();
    record(160,FastTimer(0));

    until (Timer(0)>=18); // = 180 centiseconds
    Sensor2();
    record(180,FastTimer(0));

    until (Timer(0)>=20); // = 200 centiseconds
    Sensor1(); Analyser();
    record(200,FastTimer(0));

    until (Timer(0)>=24); // = 240 centiseconds
    record(240,FastTimer(0));
}
int s=1, b=0, // status and queue variables
    m=0, g=1; // incoming message and gate (open=1 or shut=0)

void record(int thetime, int theval) {
  AddToDatalog(thetime);
  AddToDatalog(theval);
}

sub ValveManagerResponse() {
  m=Message();
  if (m>0) { s=m; PlayTone(440+50*m,1); } else PlaySound(SOUND_UP); // capture
  latest gone high or gone low change
  if (s==2) // light has gone high
    { OnFwd(OUT_A);
      Wait(15);
      Off(OUT_A);
    } // else no change to valve
}

task CommandManager() {
  while (true) {
    record(200+s*10+b, FastTimer(0));
    if (b>0) { // if button pushes to service then
      b--;
      if (s==1) // if valve most recently went low (default)
        // to be checked: why/if signals are lost
        { OnRev(OUT_A); OnFor(OUT_A,2); }
        // then reverse a little
    }
    Wait(25); //=27/100s makes average period 30
    // original 29 causes buffer overflow for 15+45 pattern
    // but 23 quite different no overflow but xxxxx-xxxx- with gaps
    between groups of 4
    // 24 similar xxxxx-xxxx- and xxxxxxxxxx-xxxx-xxxxx- again no
    overflow
    // 25,26,27 similar longer xx sequences and no overflow
  }
}

sub CommandCaptureResponse() {
  b++;
  if (b>2) PlayTone(300,2); // error: queue overflow
}
task main() {

CreateDatalog(900);

SetSensor(SENSOR_3, SENSOR_TOUCH); // user button input

SetEvent(1, Message(), EVENT_TYPE_MESSAGE);

SetEvent(3, SENSOR_3, EVENT_TYPE_PRESSED);
SetPower(OUT_A, OUT_HALF);
ClearTimer(0);

start CommandManager; // run periodic task, period=30/100 s

while (true) {
if (Timer(0) >= 100) ClearTimer(0); // ie. Fasttimer>=1000

monitor (EVENT_MASK(1) | EVENT_MASK(3)) {
  Wait(1000);
}
catch (EVENT_MASK(1)) { // priority 1, separation at least 40/100 s
  ValveManagerResponse();
  record(111, FastTimer(0));
}
catch (EVENT_MASK(3)) { // priority 3: separation at least 15/100 and
// no more than 2 pushes in 60cs
  CommandCaptureResponse();
  record(333, FastTimer(0));
}
}