Sensor voter

We consider a set of distributed components communicating via a bus network:

```
mtype = {sensor_msg, voter_msg}

chan q = [4] of {mtype, bool, short};
```

The channel interconnects all components, allowing for messages to contain an `mtype` which supports message types `sensor_msg` and `voter_msg`, a `bool` for specifying whether a self-checked error is reported, and a `short` containing the value to be sensed and voted.

Our system contains the following distributed components: three *Sensors*, one *Voter* and one *Controller*. The three Sensors read independent values and feed them to the *Controller* via the *Voter*.

Sensors

Sensors are fundamentally very simple: Each sensor is a process initialized with a certain `input` value, which represents some measured quantity. Each sensor also has an `id` identifying it. Consider the complete code:

```
proctype Sensor(byte input; byte id) {
    short sensor_value;
    bool self_check_ok = false;
    if
    :: true ->
    self_check_ok = true;
    sensor_value = input
    :: true ->
    self_check_ok = false;
    sensor_value = -1
    fi;
    ghost_values[id] = sensor_value;
    ghost_self_checks[id] = self_check_ok;
    if
    :: true -> skip
    :: true -> end_failed: (false)
    fi;
    q ! sensor_msg, self_check_ok, sensor_value
}
```

Essentially, the sensor reads the `input` value into the local variable `sensor_value` and writes it into the bus channel `q`. It may, non-deterministically, detect and report a self-checked error. It may also block forever at the `end_failed` label.

Voter

The voter will read from bus channel `q` the values provided by the sensors. It loops, reading from the channel, and updates the `read[]` array accordingly. Messages that have the self-check bit set to `false` indicate that the sensor is faulty – those messages are discarded.

```
active proctype Voter() {
    short read[n];
    short voter_result
    bool sensor_self_check;
    byte count_values = 0;
    do
    :: q ? sensor_msg, sensor_self_check, read[count_values];
    if
    :: sensor_self_check -> count_values++
    :: else -> skip
    fi
    :: count_values == n -> break
    od;
    voter_result = read[0];
    byte i = 0;
    short max, min;
    do
    :: i < count_values ->
    if
    :: read[i] < min -> min = read[i]
    :: read[i] > max -> max = read[i]
    :: else -> skip
    fi;
    i++
    :: else -> break
    od;
    i = 0;
    do
    :: i < count_values ->
    if
    :: min < read[i] && read[i] < max ->
        voter_result = read[i]
    :: else -> skip
    fi;
    i++
    :: else -> break
    od;
    q ! voter_msg, true, voter_result
}
```

Once the voter receives all messages, it determines the maximum and minimum values received and then selects one of the values. Preferrably, the median value should be selected, although sometimes not all three values are available. In fact, all sensors may fail simultaneously, and the voter should output `-1` to indicate that.

Once the voted value is computer by this process, the voter sends a message to the bus channel `q`. Specifically, the controller is the destination of that message.

Controller

The controller process is the one that should receive the voted value, by reading from the bus network.

```
active proctype Controller() {
    short voter_output = -1;
    q ? voter_msg, true, voter_output;
recv:
    ghost_voter_output = voter_output
}
```

The controller code is indeed very simply. It waits for a `voter_msg` to be available on the network bus and then reads it. The `voter_output` is then written into a global variable named `ghost_voter_output` which is introduced in the model exclusively for specifying and verifying properties. It should be highlighted that an alternative would be to use remote references to local variables but that is incompatible with Spin's partial order reduction.

Running

To run Spin, we start by placing the source code of the complete voter example in a file named `voter.pml`. Then, a simulation may be executed by running:

```
$ spin voter.pml
```

Exhaustive verification can be performed by executing:

```
$ spin -a voter.pml
$ gcc pan.c -O2 -o pan
$ ./pan -a
```

Spin immediately finds an error, which is reported via `errors: 1` and we are able to follow the trail to the problem by running:

```
$ spin -t -v voter.pml
```

There's a problem: neither the voter nor the controller finish in a valid end state, specifically when all sensors report a self-checked error. So we need to fix some things, which leads us into the problems section below.

Problems

To solve the exercises, you may install Spin or use a *web interface for Spin*.

1. Notice that Spin checks some properties by default for us (even if we don't specify anything at all). Examples include array index bonds, arithmetic overflows and termination. Regarding termination, all processes are required to terminate or block at an `end label`. End labels mark valid points for a process to block. Identify the problem reported by Spin when we run `spin -t -v voter.pml` and fix it.

2. Once we fix the `timeout` problem of the previous exercise, running the exhaustive verification encounters the state-space explosion problem so frequent when doing explicit-state model checking. It will take about 12GB of memory and a few minutes to verify the model. However, in this exercise we can enable Partition Symmetry Reduction, which is a technique proposed to address the combinatorial problem of initial values:

   Barbosa, R., Fonseca, A. & Araujo, F. Reductions and abstractions for formal verification of distributed round-based algorithms. Software Quality Journal 29, 705–731 (2021). https://doi.org/10.1007/s11219-020-09539-6

3. One of the correctness properties that we need to formulate is that the controller eventually receives a message from the voter (regardless of the sensors). This is a classical *liveness property* that we should specify using Linear Temporal Logic (LTL). Specify and verify this property exhaustively.

4. We must also specify a *safety property* regarding the voter output: always when the voter output received by the controller is different from `-1` then this implies that at least one of the sensors' self check is OK (true). Specify and verify this property using Spin.

5. The property specified in the previous example does not hold. If we follow the error trail, Spin shows us that the voter output received by the controller may be zero. This occurs when all sensors report a self-checked error. Therefore, the voter has a bug because it does not handle the situation in which no values are received. It should return `-1` but it is returning `0` instead. Fix this problem and run the verification again.

6. Specify, using LTL, and verify the following property: always when the voter outputs a value different from `-1` then necessarily the output value has to be one of the sensor values.

7. Specify and verify an important liveness property: always when at least one sensor is OK eventually the controller will receive a sensor value.

Author

Raul Barbosa (University of Coimbra)