A Novel Run-Time Monitoring Architecture

CISTER - Research Center in Real-Time & Embedded Computing Systems

Geoffrey Nelissen, David Pereira and Luís Miguel Pinho
{grrpn, dmrpe, lmp}@isep.ipp.pt

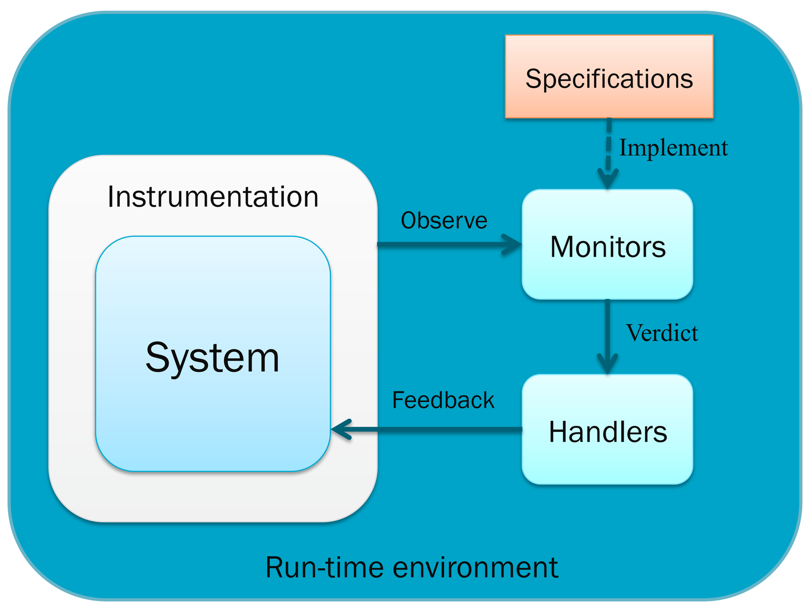
Run-Time Verification

Rationale:

- Computing architectures become more and more complex and sometimes unpredictable.
- Verification techniques showed their limit, essentially when timing properties, available only at run-time, are involved.
- Testing improves the confidence but does not prove the correctness of the system in all possible situations.

Run-time verification as a solution:

- Add monitors in the system that check **at run-time** if specifications are respected.
- In case of detected anomalies, a counter-measure can be activated →play the role of a **safety-net**.



Safety-Critical Systems Requirements

- **Composable and independent development** of the monitors and monitored applications:

  Different components of a same system are usually developed by different teams or companies. Therefore, their integration should not impact their individual properties. Similarly, the monitors should not impact the properties of the monitored application.
- **Space and time partitioning:**

  Partitioning ensures that a fault in the monitored application cannot propagate to the monitors, which are supposed to detect it.
- **Simplicity:**

  To accelerate the development and ease the certification.
- **Efficiency and responsiveness:**

  To detect and react to an anomaly as soon as it happens.

Run-Time Monitoring State-of-the-Art

There exists two different implementations of run-time monitoring in the state-of-the-art:

1. **Code Injection:**
   - The monitoring code is directly called by the application code

Task 1

```
while(true) {
   ...
   buffer1.write(x);
   Call_Mon1( writeBuf1 );
   ...
   WaitNextPeriod();
}
```

Task 2

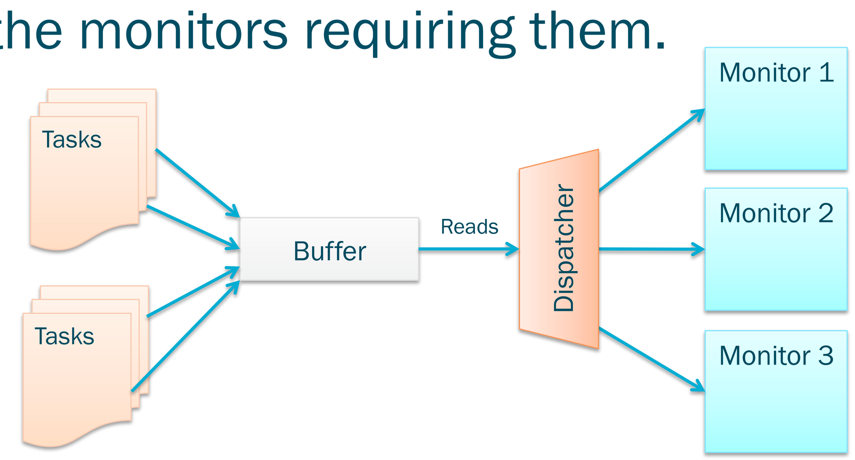
```
while(true) {
   ...
   buffer2.write(y);
   Call_Mon1( writeBuf2);
   ...
   WaitNextPeriod();
}
```

Task 3

```
while(true) {
   ...
   Call_Mon1( readBuf1);
   buffer1.read(x);
   Call_Mon1( readBuf2);
   buffer2.read(y);
   ...
   WaitNextPeriod();
}
```

   - Limitations:
     - Impact the execution time of the monitored tasks → no composability;
     - The monitor becomes a shared resource. Several tasks accessing a same monitor can be blocked by each other → no time partitioning;
     - A failure in the task may propagate to the monitor → no space partitioning.

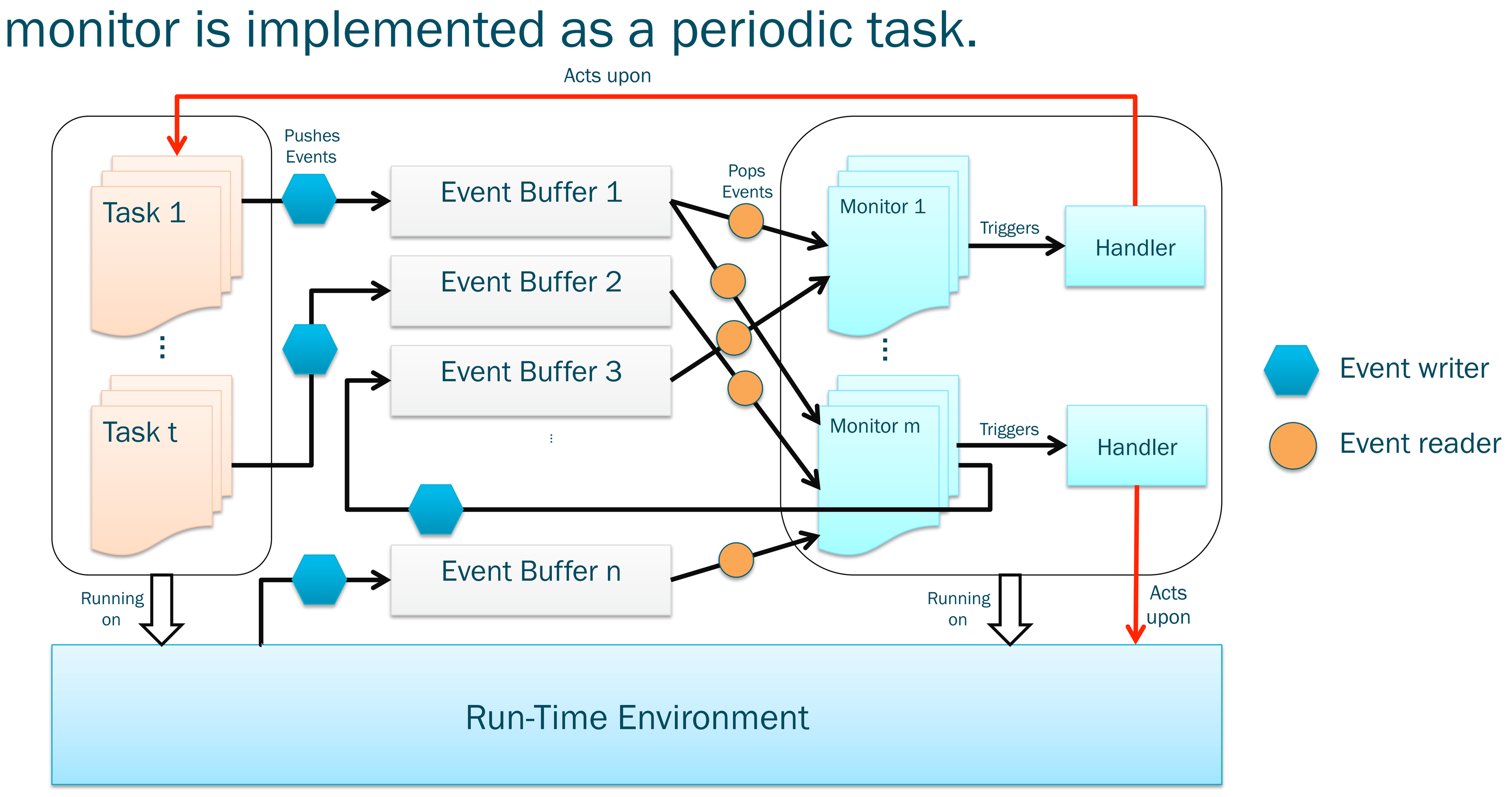
2. **Communication through buffers:**
   - All the tasks write events in a shared buffer.
   - A dispatcher reads the events saved in the buffer and sends them to the monitors requiring them.



   - Advantage: The buffer isolates in space the monitors from the monitored tasks.
   - Limitations:
     - The buffer is a shared resource. Tasks writing in the buffer may be blocked by each other → no time partitioning;
     - The dispatcher is a bottleneck, which limits the parallelism.

A Novel Architecture

- Uses one buffer per event type.
- Only one task can write in a given buffer. The write access is granted by an "event writer".
- Multiple monitors can read events from the same buffer using "event readers".
- A monitor is implemented as a periodic task.



- Advantages:
  - Buffers isolate in space the monitors from the monitored tasks.
  - No synchronization mechanism is required → no extra blocking times.
  - Composable and independent development of the monitors.
  - The monitor responsiveness can be configured by modifying its period.
- Limitation: The monitors must re-order events stored in different buffers.

Final Remarks

- The run-time monitoring architecture has been **implemented in Ada**
- A new **specification language** for safety critical embedded systems is being designed.
- An automatic correct-by-construction **monitor generation** tool is under development.

CONCERTO project ARTEMIS/0003/2012 - JU grant nr. 333053, and EMC2 project ARTEMIS/0001/2013 - JU grant nr. 621429. Co-financed by: