CISTER
Research Center in
Real-Time & Embedded
Computing Systems

Technical Report

A Novel Run-Time Monitoring Architecture for Safe and Efficient Inline Monitoring

Geoffrey Nelissen
David Pereira
Luis Miguel Pinho

CISTER-TR-150308

2015/06/22

A Novel Run-Time Monitoring Architecture for Safe and Efficient Inline Monitoring

Geoffrey Nelissen, David Pereira, Luis Miguel Pinho

CISTER Research Center

Polytechnic Institute of Porto (ISEP-IPP)

Rua Dr. António Bernardino de Almeida, 431

4200-072 Porto

Portugal

Tel.: +351.22.8340509, Fax: +351.22.8321159

E-mail: grrpn@isep.ipp.pt, dmrpe@isep.ipp.pt, lmp@isep.ipp.pt

http://www.cister.isep.ipp.pt

Abstract

Verification and testing are two of the most costly and time consuming steps during the development of safety critical systems. The advent of complex and sometimes partially unpredictable computing architectures such as multicore commercial-of-the-shelf platforms, together with the composable development approach adopted in multiple industrial domains such as avionics and automotive, rendered the exhaustive testing of all situations that could potentially be encountered by the system once deployed on the field nearly impossible. Run-time verification (RV) is a promising solution to help accelerate the development of safety critical applications whilst maintaining the high degree of reliability required by such systems. RV adds monitors in the application, which check at run-time if the system is behaving according to predefined specifications. In case of deviations from the specifications during the runtime, safeguarding measures can be triggered in order to keep the system and its environment in a safe state, as well as potentially attempting to recover from the fault that caused the misbehaviour. Most of the state-of-the-art on RV essentially focused on the monitor generation, concentrating on the expressiveness of the specification language and its translation in correct-by-construction monitors. Few of them addressed the problem of designing an efficient and safe run-time monitoring (RM) architecture. Yet, RM is a key component for RV. The RM layer gathers information from the monitored application and transmits it to the monitors. Therefore, without an efficient and safe RM architecture, the whole RV system becomes useless, as its inputs and hence by extension its outputs cannot be trusted. In this paper, we discuss the design of a novel RM architecture suited to safety critical applications.

A Novel Run-Time Monitoring Architecture for Safe and Efficient Inline Monitoring

Geoffrey Nelissen, David Pereira, and Luís Miguel Pinho

CISTER/INESC TEC, ISEP
Polytechnic Institute of Porto
Porto, Portugal
{grrpn, dmrpe, lmp}@isep.ipp.pt

Abstract. Verification and testing are two of the most costly and time consuming steps during the development of safety critical systems. The advent of complex and sometimes partially unpredictable computing architectures such as multicore commercial-of-the-shelf platforms, together with the composable development approach adopted in multiple industrial domains such as avionics and automotive, rendered the exhaustive testing of all situations that could potentially be encountered by the system once deployed on the field nearly impossible. Run-time verification (RV) is a promising solution to help accelerate the development of safety critical applications whilst maintaining the high degree of reliability required by such systems. RV adds monitors in the application, which check at run-time if the system is behaving according to predefined specifications. In case of deviations from the specifications during the runtime, safeguarding measures can be triggered in order to keep the system and its environment in a safe state, as well as potentially attempting to recover from the fault that caused the misbehaviour. Most of the state-of-the-art on RV essentially focused on the monitor generation, concentrating on the expressiveness of the specification language and its translation in correct-by-construction monitors. Few of them addressed the problem of designing an efficient and safe run-time monitoring (RM) architecture. Yet, RM is a key component for RV. The RM layer gathers information from the monitored application and transmits it to the monitors. Therefore, without an efficient and safe RM architecture, the whole RV system becomes useless, as its inputs and hence by extension its outputs cannot be trusted. In this paper, we discuss the design of a novel RM architecture suited to safety critical applications.

Keywords: Run-Time Monitoring, Run-Time Verification, Safety Critical Systems, Ada

1 Introduction

Run-time verification (RV) [7,16] entails adding pieces of code called *monitors* to a running application. The *monitors* scrutinise the system behaviour and check

if it respects associated specifications. The monitors can detect anomalies during the execution of the application. That information may be logged and back propagated to the system designer. It is therefore an efficient solution to detect bugs and other deficiencies when a complete static verification of the system is not possible. By keeping the monitors running in the system after its deployment, run-time verification can also be used as a tool to increase the safety and reliability of systems during their operation, triggering counter-measures when anomalies are detected and hence acting as a safety net around the monitored application.

With the advent of more and more complex computing platforms (e.g., multicore processors, many-core accelerators, networks on chip, distributed systems interconnected with various communication networks) and the adoption of new computing paradigms to exploit the power of those architectures, verifying whether a system respects its functional (e.g., order of execution and validity of results) and extra-functional (e.g., timing constraints) specifications became a big challenge. Static verification (i.e., the formal proof that the system is correct) has proven limited either because of the state space explosion problem as in the case of approaches based on model checking, or simply due to theoretical limitations related to the expressivity and decidability of approaches based on deductive verification. Furthermore, ensuring the correctness of extra-functional properties before the system deployment is usually subject to a high degree of pessimism, essentially because the data that must be manipulated (e.g., execution time, inter-arrival time or response time) are almost always available only at run-time, and depend on specificities of the underlying hardware (e.g., communication protocols on shared buses, replacement policies in caches, operation ordering in execution pipelines), interactions with the external physical environment and interference caused by concurrent applications.

Testing and simulations are often presented as solutions to improve the confidence in the final system. However, one can never ensure an exhaustive testing of all the possible situations that may be encountered after deployment. Therefore, successfully passing all the tests does not provide any guarantee that the system is bug-free or that it will always respect all its requirements.

Furthermore, today's practices in product development rely extensively on distributed development. Multiple partners and subcontractors develop different functionalities that are later integrated together to form the final product. As a result, most of the components are black-boxes and none of the partners knows exactly how they all have actually been implemented. Therefore, ensuring an exhaustive testing and fully trusted verification of the system before its deployment becomes nearly impossible. Keeping monitors running together with the applications in order to detect potential misbehaviours and trigger safe-guarding measures should therefore be considered as a promising solution to accelerate the product development cycle whilst maintaining the high safety requirements associated to such systems.

Most of the state-of-the-art on run-time verification focuses on the design of formal languages [3,15,18] for the specification of properties that must be verified

at run-time. Those languages are then used to generate monitors in a correct-by-construction manner. However, run-time verification cannot work without appropriate mechanisms to actually monitor the system and extract meaningful information that can then be used by the monitors to assert the respect of the specifications. This basic infrastructure over which any run-time verification framework is built has received much less attention from the research community. Run-time monitoring is however a corner stone of an efficient and safe run-time verification framework as it plays the interface between the monitors and the monitored applications. Previous works have developed run-time monitoring solutions as a part of full run-time verification frameworks. Most of them though, put the accent on the specification language and the monitor generation process, neglecting the implementation and run-time monitoring aspect. In this paper we specifically focus on the run-time monitoring architecture and propose a new, efficient and safe solution to integrate monitors with application code. The design of the presented run-time monitoring framework is perfectly suited to safety critical systems such as avionics, space, railway or automotive applications, as well as any other embedded system.

Contribution. This paper introduces a novel reference architecture for inline run-time monitoring. The presented solution has been designed in order to fulfil the requirements of safety critical systems. The architecture has been kept simple so as to ease its implementation in various run-time environments. A prototype, written in Ada, of this new reference architecture is presented and experimental results show that the overhead caused by the instrumentation of the code is constant w.r.t. the worst-case execution time of the tasks constituting the monitored application.

Paper organisation. In Section 2, the requirements for the design of a reliable and efficient run-time monitoring framework for safety critical systems are presented. In Section 3, those requirements are compared with the solutions existing in the state-of-the-art. The strengths and limitations of each solution identified during this analysis, are used in Section 4 to design a novel reference architecture suited to the run-time monitoring of safety critical systems. Section 5 presents the implementation of the reference architecture in Ada and Section 6 provides results of experiments conducted on this first implementation.

2 Requirements

As later discussed in Section 3, most of the state-of-the-art solutions on run-time monitoring and verification rely on software engineering concepts that are not compatible with the requirements of safety critical systems. Therefore, with the objective of providing an adequate solution for run-time monitoring, in this section, we start by studying the requirements of safety critical systems. Those will serve as a basis to build the foundations of our new run-time monitoring framework presented in Section 4.

2.1 Independent and composable development

Distributing the development of complex systems between different partners and subcontractors became a common practice in the industry. Once delivered, the components are integrated together to form the final product. This methodology is well established in the industry and should not be reconsidered for the development of monitors. Yet, it can only be achieved in an efficient manner if both *composability* and *compositionality* are possible. Compositionality ensures that the overall application behaviour can result from the composition of its constituting subcomponents. On the other hand, composability enforces that the behaviour of individual components remain unchanged when considered in isolation and after their integration within the system.

In fact, the certification standards recommend the development and the verification of safety critical systems to be performed by different actors in order to reduce the chances of undetected anomalies. The same principle should thus be followed for the monitors generation. Indeed, if the same actor is responsible for both the application development and the monitor generation, this would increase the likelihood of common errors being introduced in the monitor specification and in the monitored application. The generated monitor would become useless as the potential system misbehaviour would remain undetected during the execution.

2.2 Time and space partitioning

Time and space partitioning is a way to ease the composable development of software applications, as well as to improve the overall safety of the system. Spatial partitioning ensures that a task in one partition is unable to change private data of another task in another partition. Temporal partitioning, on the other hand, guarantees that the timing characteristics of tasks, such as their worst-case execution times, are not affected by the execution of other tasks in other partitions. Consequently, time and space partitioning enforces two important features:

- the properties of different components pertaining to different partitions and integrated in the same system are not impacted by each other;
- because the tasks of different partitions are isolated, a fault in one partition cannot propagate to another partition, thereby improving the reliability of the whole system.

Therefore, it becomes evident that it should be possible to isolate monitors and monitored applications through time and space partitioning. This would ensure (i) that the newly introduced monitors do not modify the behaviour of the monitored applications and (ii) that a fault happening in the monitored application does not propagate to the monitors, which would prevent its detection and the triggering of the appropriate safe-guarding measures.

In a completely safe design, it should also be possible to isolate monitors from each other. Hence, a monitor misbehaving due, for instance, to corrupted input data or other external causes, could not affect other monitors and their own verification work.

2.3 Simplicity

Although simplicity should always be one of the main concerns in any development process, it is of special importance for safety critical systems. Indeed, a simple design eases the understanding of the overall system, the writing of detailed specifications and hence the potentially expensive and time consuming certification or qualification processes.

Moreover, one of the advantages claimed by run-time verification is the acceleration of the system development by partially removing the need of a complete verification of the system before its deployment. Indeed, correctly implemented monitors that are cleverly introduced in the system will be responsible for keeping safe the corresponding blocks of code that they observe, thereby acting as a safety net in case of the manifestation of an untested execution scenario. It would thus be contradictory to replace the complex and time consuming testing and formal verification phase by a difficult process of code instrumentation and monitor integration.

2.4 Efficiency and responsiveness

Monitors should be able to detect misbehaviours and respond in an acceptable amount of time in order to be useful. It would be of poor interest to launch counter-measures long after some anomalies have caused irremediable consequences to the system. Therefore, an efficient implementation is required to ensure that the information needed for the verification of the system correctness reach the monitors as soon as needed.

3 Study of existing solutions

This section reviews the solutions for run-time monitoring that were already considered in the literature. We relate each of them to the requirements derived in Section 2 in order to extract their strengths and limitations. The results of this study will then be used in Section 4 to design a run-time monitoring architecture suited to safety critical systems.

All the works discussed in this paper, target the run-time monitoring of software systems. In that context, we formally define an *event* and *event instance* as follows:

Definition 1 (Event) *An event is a set of conditions that must be encountered during the system execution.*

Definition 2 (Event instance) *An event instance is the realisation of its associated event. An event instance is therefore characterised by a unique timestamp.*

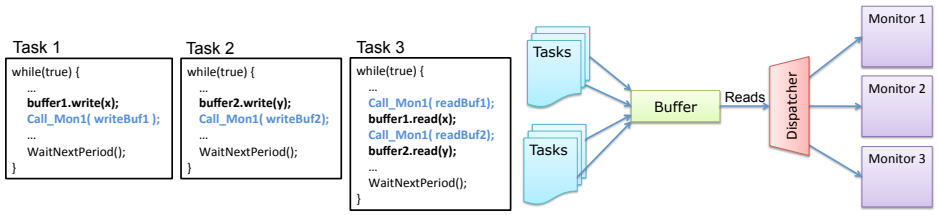

Fig. 1: State-of-the-art run-time monitoring architectures.

3.1 Code injection

The most common implementation of monitors in the state-of-the-art run-time verification frameworks consists in injecting the monitoring code directly in the application code. Fig. 1a illustrates that approach with three tasks. Two tasks write data that are read by the third task. A monitor named `Mon1` is called after each write and before each read to check properties on the data exchange. In such implementation, the monitoring procedure is executed as part of the three tasks.

Eagle [1], Hawk [6], RuleR [2], MOP [4], TemporalRover [8], Rmor [9] and RV [17] are examples of frameworks implementing that solution. Those monitoring frameworks are based on Aspect Oriented Programming [12] such as AspectJ [14], AspectC [5], or similar instrumentation mechanisms. The monitoring procedures are directly called when the monitored application passes by pre-specified positions in the source or compiled code. Whilst simple to implement, this monitoring architecture exhibits multiple drawbacks as discussed below:

- Whereas space partitioning could still be enforced with such monitoring architecture, the existing implementations of those frameworks simply do not consider it. This lack of isolation is the cause of two major issues:
  - Since the monitored application shares the same address space as the monitors, a faulty application generating memory errors due, for example, to stack overflows or wrong pointer manipulations, may corrupt the data manipulated by the monitor and alter its correct behaviour. The monitor would become unreliable, incapable to detect the fault or even worse, outputting a wrong diagnosis, which may lead to the triggering of counter-productive measures, thereby worsening the situation.
  - Monitors have direct reading and writing access to globally defined entities of the monitored application (for example, `Buffer1` and `Buffer2` in the example of Fig. 1a). A faulty monitor, wrongly implemented (voluntarily or not), could therefore modify and corrupt the content of the application variable and hence impact its execution. This generates safety and security threats for the overall system. This would be unacceptable in safety-critical applications.

– Such monitoring architecture does not allow any kind of timing isolation between monitors and monitored applications. Since the monitoring procedures are directly included in the application code, the timing properties such as the application worst-case execution time (WCET), are irremediably modified after the addition the monitoring code. For example, in the example of Fig. 1a, if `Task 1` has a WCET of 15ms before instrumentation and assuming an execution time of 5ms for the procedure `CallMon1(.)` of `Mon1`, the WCET of `Task 1` becomes at least 20ms after integration of the monitor in the system. It goes against the good practice of developing applications and monitors independently and later integrating them in a composable manner in the final system, and leads to two unwanted consequences:
  - Any timing analysis that would have been performed before the monitor integration are not valid anymore after their insertion. They should all be redone, thus generating extra work, which may be expensive and time-consuming.
  - There is no assurance that the behaviour of the application is not impacted by the addition of this extra-code. Therefore, it is impossible to predict if a well-behaving application will still respect all its specifications and thus remain correct if the monitoring code was removed.
– Since the currently adopted approaches introduce monitors as pieces of sequential code in the target application, only one thread at a time can call a given monitoring procedure, access a specific internal data, or change a state variable. A synchronisation protocol must be introduced, thereby limiting the parallelism in the monitored application. This makes this implementation particularly inefficient for multi-threaded applications such as the one presented in Fig. 1a, where events generated by different threads (i.e., tasks) may be required by a same monitor. This is a major drawback of this monitoring architecture in a context of generalisation of the use of multicore processors in embedded and general purpose systems and considering the common adoption of multi-threaded programming languages.

3.2 Independent monitors and buffered communications

The alternative to the monitoring architecture presented above is to implement monitors as independent components that receive events information through buffers (see Fig. 1b). It is the approach followed in Java-MaC [13], Java PathExplorer [10] and Java MultiPathExplorer [19].

One global buffer The most straightforward implementation of this solution would assume that all events are written in a shared buffer. In order to avoid the need for every monitor to read all events stored in the buffer — including those that are of no interest — an event dispatcher (see Fig. 1b) is usually attached to the output of the buffer [10, 13]. The dispatcher reads the events pushed in the buffer by the monitored application and redistributes them to the monitors requiring them.

Unfortunately, the use of a unique global shared buffer causes a bottleneck in the flow of information. Indeed, only one event instance can be written in the buffer at a time. This requires the use of a synchronisation mechanism among different threads that would try to access the buffer concurrently. This may lead to unwanted blocking of the tasks of the monitored application, thereby impacting their timing properties and thus going against any sort of timing isolation as their timing behaviours now become dependent on the number of threads pushing events in the buffer. Moreover, forbidding parallel event writing slows down the transfer of information to the monitors and hence their capacity to detect anomalies in an acceptable amount of time (i.e., responsiveness).

One private buffer per monitor An alternative implementation would be to provide a private buffer for each monitor. As a consequence, there is no need for an event dispatcher anymore. Although diminishing the number of potential concurrent accesses to the same buffer, a synchronisation mechanism is still necessary when events generated by multiple threads are needed by the same monitor. Furthermore, additional issues must now be considered:

- When an event is useful for multiple monitors, the monitored application must be instrumented multiple times in order to push the event in the buffers of each relevant monitor. This means that the instrumentation code is redundant. Moreover, the same event instance might be copied in multiple buffers, thereby increasing the memory footprint of the monitoring architecture.
- The monitors using the events must be known when instrumenting the application. Adding, removing or replacing monitors require re-instrumenting the code, even if they all use the same events and data. This impacts the extensibility of the system as well as the independency between monitors and monitored applications.

4 A novel reference architecture for run-time monitoring

Strong from the analysis performed in the two previous sections, we propose a new reference architecture for run-time monitoring suited to safety critical as well as non-critical applications. This reference architecture could then be implemented in any language, as long as it respects the prescriptions discussed below. The proposed run-time monitoring framework is depicted in Fig. 2. It is composed of five main components: (1) the monitored application, (2) the monitors, (3) *buffers* through which events are transmitted, (4) *buffer writers* for writing events in a specific buffer and (5) *buffer readers* used by the monitors to read events stored in a buffer. This structure may look straightforward, yet multiple details discussed below, allow us to fulfil each and every requirement presented in Section 2.

4.1 Event transmission

Whenever an event instance is generated during the execution of the monitored application, the information must be transmitted to the monitors. However,

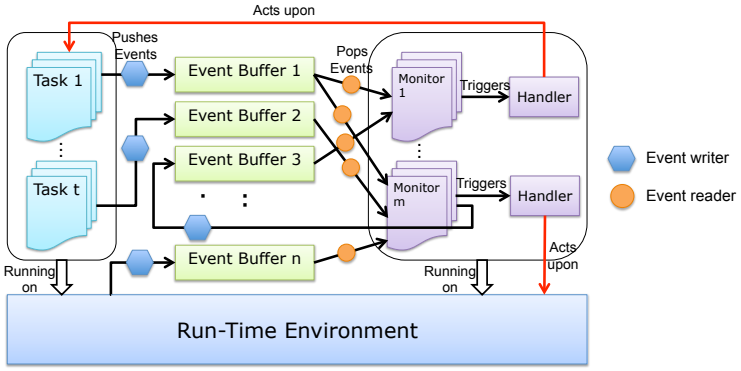

Fig. 2: Proposed reference architecture for run-time monitoring.

unlike most of the run-time monitoring frameworks presented in the previous section, in the proposed reference architecture, event instances are not directly sent to the monitors using them. Instead, the timestamps of the event instances are stored in an intermediate buffer specific to that event. That is, there is one buffer per event rather than one buffer per monitor. As a consequence, the monitored application is unaware of the existence of any potential monitor using the transmitted information thanks to the buffers playing the role of an interface. Moreover, the monitored application must be instrumented only once, thereby reducing the dependency between the application instrumentation and the monitors generation. Finally, it avoids any type of data or monitoring code redundancy.

Efficient event reading As shown in Fig. 2, each event (i.e., the completion of a task, the access to a data, the release of a semaphore, ...) is associated with a different buffer. Each buffer can then be accessed by any monitor that would require information about that specific event. Multiple monitors can access the same buffer using different *buffer readers*. Buffer readers are independent from each other. Indeed, each buffer reader uses its own pointer pointing to the oldest data that has not yet been accessed. Because each buffer reader uses a different pointer to read data in the buffer they are associated with, simultaneous accesses to a same buffer is possible without any blocking time. Furthermore, a data read by one buffer reader is not erased from the buffer and can thus still be accessed by any other buffer reader that did not read it yet. This mechanism allows multiple monitors to use the same events without using any synchronisation protocol nor duplicating the events data for each monitor making use of it. Consequently, there is a gain in terms of transmission time as well as required memory space.

Efficient event writing One of the major design choice in the reference architecture of the proposed run-time monitoring framework is that only one task

can have write accesses to a specific buffer. Those rights are granted by a *buffer writer*. Only one buffer writer can be instantiated per buffer and only one task can use that buffer writer. This added constraint avoids the need of synchronisation mechanisms since concurrent writing in the same buffer becomes impossible.

With the same objective of reducing the impact of the code instrumentation on the application performances, the buffers are circular. That is, a task writing an event in the buffer can never be blocked due to the buffer being full. In such a situation, the oldest event instance in the buffer will simply be overridden with the new data. It is therefore the role of the monitor developer to make sure that (i) the buffer is large enough and (ii) the monitors are reading events frequently enough to avoid missing any meaningful information.

4.2 Monitor implementation

In the proposed reference run-time monitoring architecture, a monitor is a task which is periodically activated. At each activation, a monitor calls a monitoring procedure, which is provided by the monitor designer. A monitoring procedure is usually implemented as a finite state machine. It reads information about some specific events and makes its state evolve. If the state machine reaches an erroneous state, safe-guarding measures may be activated. The generation of the monitoring procedure is however out of the scope of this paper as it relates to the general run-time verification problem rather than the specific run-time monitoring issue discussed in this paper. The interested reader may consult the following (non-exhaustive) list of references for further information on that topic [2–4, 11, 18].

Note that, because the monitors are implemented as periodic tasks, their activation is completely independent from the execution of the monitored application. Unlike the solutions based on Aspect Oriented Programming and similar technologies, the execution of the monitored application is not delayed by the execution of the monitor whenever an event instance is generated. Instead, the monitor is periodically activated and checks all the meaningful information on events that may have been realised during its last period of inactivity. Therefore, implementing a monitor as an independent task rather than inserting code in the application, keeps the properties of the monitored application unaffected by the execution of the monitors, thereby allowing for the independent and composable development of monitors and monitored applications.

However, because the monitors are isolated in time from the monitored application (i.e., their executions are independent), there is a delay between the realisation of an event and its treatment by the monitor. This latency impacts the time needed to react to a system fault. However, because the monitors are implemented as periodic tasks, and under the assumption that the system is schedulable, this latency can be upper-bounded. Let T be the period of monitor `Mon1` and let t_0 be the beginning of a period of `Mon1`. `Mon1` can be executed at any time between t_0 and $t_0 + T$. A second execution of `Mon1` will then happen between $t_0 + T$ and $t_0 + 2 \times T$. In the worst-case scenario, `Mon1` executes right at t_0 and then at the end of its next period, that is, right before $t_0 + 2 \times T$. If

an event instance is generated just after the first execution of `Mon1`, it will be treated only during the second execution of the monitor at time $t_0 + 2 \times T$. The maximum latency can thus be upper-bounded by $2 \times T$ (i.e., twice the period of the monitor). As a consequence, the responsiveness of the monitor can be configured by modifying its period of activation.

4.3 Events ordering

One of the major advantages of the architecture depicted on Fig. 2 is also one of its major issues. Because different events are stored in different buffers, instances of different events are not ordered (w.r.t. their timestamps). Yet, the order in which events happen is often a key property to detect faults, bugs, deficiencies or other execution anomalies. Since a monitor may need information from more than one event, it becomes its job to reorder event instances stored in different buffers before to treat them. In the proposed reference architecture, the monitor performs this task by using *synchronised buffer readers*. The synchronised buffer readers are instantiated inside the monitors. Synchronised buffer readers associated with a same monitor share a synchronisation variable. This variable stores the timestamp of the last event instance read by the monitor in any buffer. This information is used by all the synchronised event readers to determine which is the next event instance that must be considered in their buffer. The use of the synchronisation variable by the synchronised event readers is summarised in the pseudo-code provided in Algorithm 1.

Algorithm 1: Pseudo-code of the function reading an event in a synchronised buffer reader.

```
Data: Synch_Variable; Current_Index; End_Buffer
1 while Current_Index ≠ End_Buffer do
2 |   if Buffer[Current_Index].timestamp ≥ Synch_Variable then
3 |   |   Synch_Variable ← Buffer[Current_Index].timestamp;
4 |   |   Current_Index ← Current_Index+1;
5 |   |   return Buffer[Current_Index−1];
6 |   end
7 |   Current_Index ← Current_Index+1;
8 end
9 return "Buffer empty";
```

When accessing its buffer, the synchronised event reader looks for the first event instance stored in the buffer with a timestamp larger than or equal to the timestamp saved into the synchronisation variable. If such an element exists, the timestamp of that event instance is saved in the synchronisation variable and the event instance information are sent back to the monitor. Otherwise, the event reader keeps the synchronisation variable unchanged and notify the monitor that the buffer is empty.

Note that because a monitor is implemented as a sequential task and because all the synchronised event readers sharing the same synchronisation variable are associated with the same monitor, only one synchronised event reader is called at a time. Therefore, there cannot be concurrent accesses to the synchronisation variable and no access protection mechanism is hence required.

4.4 Isolation

The architecture described above is particularly well designed to achieve the complete isolation of the monitors from the monitored application. Indeed, each monitor is an independent task which can be restricted to its own memory address space, interacting with the application only through the unidirectional communication channels implemented by the event buffers.

Furthermore, if the system was deployed on a multicore platform, monitors could be executed on their own core(s), in parallel with the monitored application thanks to the multi-threaded approach promoted in this architecture.

4.5 Additional remarks

As depicted on Fig. 2, the proposed architecture is quite flexible and thus allows task of the monitored application, but also the run-time environment (usually an operating system) and the monitors themselves to generate events and push them in their own buffers. Since the monitors are isolated from the rest of the system, they do not know anything about the architecture of the monitored application. Therefore, from the viewpoint of one monitor, another monitor generating events is seen as any other task pertaining to the monitored system. This paves the way to the definition of parallel and hierarchical verification frameworks, in which big monitoring procedures would be decomposed in a set of smaller monitors that would implement the overall monitoring functionality by exchanging information through event instances.

5 Library implementation and usage

A prototype of the monitoring architecture described in Section 4 has been implemented in Ada.

The buffer is implemented as a generic package. Each instance of that package is associated with a specific event. As shown in the example code below, it requires the definition of two generic variables; the length of the circular buffer and the type of an additional data that can be saved together with the timestamp of each event instance. This data could be the identifier of a task, the value of a counter or any other meaningful information for the monitors using it. For safety reasons, and to keep external components from altering the buffer content without using a buffer writer as prescribed in the reference architecture, the package is entirely private, leaving no interface for the external world to interact with it.

```
1 package Buffer_1 is new
2     Buffer(Event_Info_Type => Integer, Buffer_Length => 20);
```

Implementing the buffers as a generic package rather than a generic type (e.g., a record) may look unconventional. However, it allows us to implement the buffer writer as a generic child package, thereby permitting the package to check that there is only one buffer writer per buffer instance.

In order to be fully compliant with the proposed reference architecture, the event writers should be instantiated as described in the example code provided below, limiting the scope of the event writer to one task only.

```
1  task T1 with Priority => 10;
2  task body T1 is
3      package Buffer1_Writer is new Buffer_1.Writer;
4      T : Ada.Real_Time.Time;
5  begin
6      loop
7          T := Ada.Real_Time.Clock;
8          Buffer1_Writer.Write(Data => 1, TimeStamp => T);
9          (...)
10     end loop;
11 end T1;
```

The event writer interface provides only one procedure, which allows to write an event instance in the associated buffer. The `Write` procedure (see line 8 in the example code above) takes two parameters; the timestamp of the event instance and a data that provides extra-information for the monitors.

The skeleton of any monitor is implemented in the generic `Monitor` package. It implements the synchronised event readers and the periodic monitoring task as two private nested packages. This isolates the monitor from the external world, forbidding any external component to generate new monitoring tasks or synchronised event readers that could mess with the synchronisation variable of the monitor and hence with its correct behaviour.

To generate a new monitor, the system designer must first create a new child package to the `Monitor` package. It is in that new package that the monitoring procedure must be implemented and the periodic monitoring task calling that procedure instantiated. An example of such monitor definition is provided below.

```
1  generic
2  package Monitor.Procedure1 is
3      (...)
4  end Monitor.Procedure1;
5
6  package body Monitor.Procedure1 is
7      procedure MonProc1 is
8          Data : Integer;
9          Timestamp : Ada.Real_Time.Time;
10         IsEmpty, HasGap : Boolean;
```

```
11          package B1_Reader is
12              new Synchronized_Reader(Buffer_1);
13          (...)
14      begin
15          B1_Reader.Pop(Data, Timestamp, IsEmpty, HasGap);
16          (...)
17      end MonProc1;
18
19      package MonTask1 is
20          new Monitoring_Task(Monitoring_Procedure => MonProc1);
21  end Monitor.Procedure1;
```

The implementation of the monitoring procedure is left completely to the monitor designer. This procedure may instantiate synchronised readers — as exemplified at line 13 of the code above — to access the content of some specific event buffers. The `Synchronised_Reader` package takes the buffer from which it must read as a generic argument, and provides two procedures to extract information from that buffer: `Pop` and `Get`. `Pop` implements the basic reading procedure presented in the pseudocode of Algorithm 1. It finds the first event instance with a timestamp larger than the value saved in the synchronisation variable, updates the synchronisation variable and sends the data and timestamp associated with the found event instance back to the monitoring procedure. `Get`, however, also finds and sends back the information associated with the first meaningful event instance in the buffer but does not update the synchronisation variable. The monitoring procedure can use `Get` in order to read multiple times the same event instance or to first compare the timestamps and data of events stored in different buffers before deciding which event must actually be considered. `Pop` and `Get` may set the two booleans given as parameters in order to notify if the buffer is empty and if there is a gap in the trace stored in the buffer due to the circular nature of that buffer.

Once the monitoring procedure is implemented, the monitor designer must instantiate a monitoring task as described on line 21 of the example code provided above. This task will periodically call the monitoring procedure. Note that only one task can be instantiated per monitor. If multiple instantiations were attempted, then an exception would be raised.

Finally, the integration of the monitor in the system is done as shown below, defining the period and the priority of the monitoring task as the values of the two generic arguments of the `Monitor` package.

```
1  package Monitor1 is new Monitor(Period=>100, Priority=>61);
2  package Monitor1_Proc is new Monitor1.Procedure1;
```

6 Experimental Results

In order to evaluate the overhead generated by our run-time monitoring architecture, we conducted a set of experiments varying the number of monitored

Tasks-Monitors-read/write	1-1-w	1-5-w	1-10-w	5-1-w	10-1-w	1-1-r	1-5-r	1-10-r	5-1-r	5-5-r
Average	0.74	0.69	0.58	0.62	0.82	0.53	0.81	0.70	0.94	0.83
Standard Deviation	0.48	0.53	0.5	0.49	0.52	0.50	0.54	0.52	0.34	0.47
Maximum	1	2	2	1	2	1	2	2	2	2

Table 1: Experimental results (values in μs).

tasks (from 1 to 10), event buffers (one per task) and monitors (from 1 to 10) reading from a same buffer. For each configuration, the time needed to read and write 100 event instances in their buffers has been monitored. The average value, maximum value and the standard deviation have been computed for each experiment. The experimental systems were implemented using the Ada prototype of the architecture presented in the previous section and compiled using the `gnat` toolchain. The experiments were performed on a quad-core Intel i7 processor cadenced at 2.3GHz with 8GB of RAM memory and a 500 GB Solid State SATA Drive. The results of the experiments are presented in Table 1. As a consequence of the absence of protection mechanisms in the reference architecture, the variation on the cost of writing and reading event instances is small and independent of the number of monitors and tasks generating events. Consequently, the maximum overhead induced by the monitoring architecture can be precisely calculated and hence included in the timing analysis of the application.

7 Conclusion

After the study of the requirements of safety critical systems and the extraction of the limitiations in the solutions implemented in the existing run-time verification frameworks, we presented a novel architecture for the implementation of a safe and reliable run-time monitoring framework. We exposed how this new design, whilst remaining simple and efficient, allows for the independent and composable development of monitors and monitored applications, and improves the safety and reliability of the overall system by making possible the complete isolation of the monitors from the rest of the system. As a consequence, a fault in the monitored application cannot propagate anymore to the monitors checking its correct behaviour. A first implementation of the proposed reference architecture together with experimental results were presented in the paper, thereby proving the feasibility of the promoted approach.

Acknowledgement. This work was partially supported by National Funds through FCT/MEC (Portuguese Foundation for Science and Technology) and when applicable, co-financed by ERDF (European Regional Development Fund) under the PT2020 Partnership, within project UID/CEC/04234/2013 (CISTER Research Centre); also by, FCT/MEC and the EU ARTEMIS JU within projects ARTEMIS/0003/2012 - JU grant nr. 333053 (CONCERTO) and ARTEMIS/0001/2013 - JU grant nr. 621429 (EMC2).

References

1. Barringer, H., Goldberg, A., Havelund, K., Sen, K.: Rule-based runtime verification. In: Steffen, B., Levi, G. (eds.) VMCAI. Lecture Notes in Computer Science, vol. 2937. Springer (2004)
2. Barringer, H., Havelund, K., Rydeheard, D., Groce, A.: Runtime verification. chap. Rule Systems for Runtime Verification: A Short Tutorial, pp. 1–24 (2009)
3. Bauer, A., Leucker, M., Schallhart, C.: Runtime verification for ltl and tltl. ACM Trans. Softw. Eng. Methodol. 20(4), 14:1–14:64 (Sep 2011)
4. Chen, F., Roşu, G.: Mop: An efficient and generic runtime verification framework. In: Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications. pp. 569–588. OOPSLA (2007)
5. Coady, Y., Kiczales, G., Feeley, M., Smolyn, G.: Using aspectc to improve the modularity of path-specific customization in operating system code. SIGSOFT Softw. Eng. Notes 26(5), 88–98 (Sep 2001)
6. d'Amorim, M., Havelund, K.: Event-based runtime verification of java programs. SIGSOFT Softw. Eng. Notes 30(4), 1–7 (May 2005)
7. Delgado, N., Gates, A.Q., Roach, S.: A taxonomy and catalog of runtime software-fault monitoring tools. IEEE Trans. Softw. Eng. 30(12), 859–872 (Dec 2004)
8. Drusinsky, D.: The temporal rover and the atg rover. In: Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification. pp. 323–330 (2000)
9. Havelund, K.: Runtime verification of c programs. In: Proceedings of the 20th IFIP TC 6/WG 6.1 International Conference on Testing of Software and Communicating Systems: 8th International Workshop. pp. 7–22. TestCom '08 / FATES '08 (2008)
10. Havelund, K., Roşu, G.: An overview of the runtime verification tool java pathexplorer. Form. Methods Syst. Des. 24(2), 189–215 (Mar 2004)
11. Havelund, K., Rosu, G.: Synthesizing monitors for safety properties. In: In Tools and Algorithms for Construction and Analysis of Systems (TACAS02. pp. 342–356. Springer (2002)
12. Kiczales, G.: Aspect-oriented programming. ACM Comput. Surv. 28(4es) (Dec 1996)
13. Kim, M., Viswanathan, M., Kannan, S., Lee, I., Sokolsky, O.: Java-mac: A run-time assurance approach for java programs. Form. Methods Syst. Des. 24(2), 129–155 (Mar 2004)
14. Kiselev, I.: Aspect-Oriented Programming with AspectJ. Sams, Indianapolis, IN, USA (2002)
15. Konur, S.: A survey on temporal logics for specifying and verifying real-time systems. Front. Comput. Sci. 7(3), 370–403 (Jun 2013)
16. Leucker, M., Schallhart, C.: A brief account of runtime verification. The Journal of Logic and Algebraic Programming 78(5), 293 – 303 (2009), the 1st Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS07)
17. Meredith, P., Roşu, G.: Runtime verification with the rv system. In: Proceedings of the First International Conference on Runtime Verification. pp. 136–152 (2010)
18. Sen, K.: Generating optimal monitors for extended regular expressions. In: In Proc. of the 3rd Workshop on Runtime Verification (RV03), volume 89 of ENTCS. pp. 162–181. Elsevier Science (2003)
19. Sen, K., Rosu, G., Agha, G.: Runtime safety analysis of multithreaded programs. SIGSOFT Softw. Eng. Notes 28(5), 337–346 (Sep 2003)