Toward a Run-Time Verification Framework for Real-Time Safety-Critical Systems

Geoffrey Nelissen
David Pereira
Luis Miguel Pinho

CISTER-TR-151011

2015

Toward a Run-Time Verification Framework for Real-Time Safety-Critical Systems

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

No Abstract – two page paper.

Toward a Run-Time Verification Framework for Real-Time Safety-Critical Systems

Geoffrey Nelissen, David Pereira, Luís Miguel Pinho
CISTER/INESC-TEC
Polytechnic Institute of Porto, Portugal
{grrpn, dmrpe, lmp}@isep.ipp.pt

I. Rationale

With the advent of always more complex computing platforms (e.g., multicore processors, many-core accelerators, network on chips, 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 extra-functional (e.g., task deadlines and minimum inter-arrival times) specifications became a big challenge. Static verification has proven limited either because of the state 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 difficult and 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.

The real-time community relies on models (e.g., periodic, sporadic, self-suspending, parallel, constrained or implicit deadline tasks) to reason about extra-functional properties. Yet, those models are based on assumptions on the application properties (e.g., worst-case execution time, minimum inter-arrival time, worst-case blocking or self-suspending time). State-of-the-art analyses provide approximations for those properties. However, they are usually highly pessimistic due to the difficulty of accurately modelling the interactions between the tasks and the computing platform. Probabilistic approaches have also been proposed but come with the inherent limitation that, however low, the analysis outputs a probability that the extra-functional property may not be respected at run-time.

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 and input scenarios. Therefore, successfully passing all the tests does not provide any guarantee that the system is bug-free or that it will respect all its specifications after deployment.

Run-time verification is a complementary solution to those presented above. It consists in adding *monitors* in the system that gather information during the execution and check if the monitored applications behave according to predefined specifications. In case of a detected misbehaviour, counter-measures can be triggered to avoid or mitigate the impact of a failure, for example, by triggering an execution-mode change, modifying the period or the priority of some tasks, adapting the task mapping on the cores or simply deactivating non-critical tasks. Because monitors use real data generated at run-time, run-time verification allows to check if the assumptions made during the analysis and development phase are respected at run-time. Hence, solutions can now be implemented to react in case of non-allowed deviations. Although previous works on run-time verification exist [1]–[5], they are usually unsuited to real-time safety-critical applications. Yet, run-time monitoring, *dynamic reconfiguration* and *graceful degradation*, are all recommended by safety standards [6]–[8] to improve the reliability of safety-critical systems.

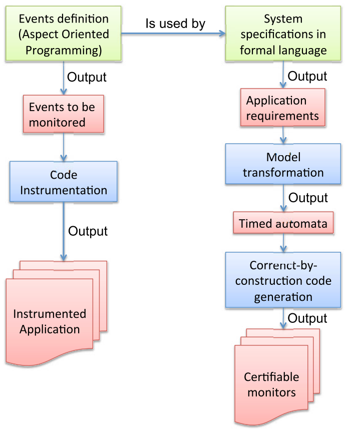
II. Presented Framework

In this talk, we will present a run-time verification framework for real-time safety-critical systems currently in development at CISTER. The different steps of the proposed approach are presented in Figure 1a. Three main challenges will be discussed as presented below.

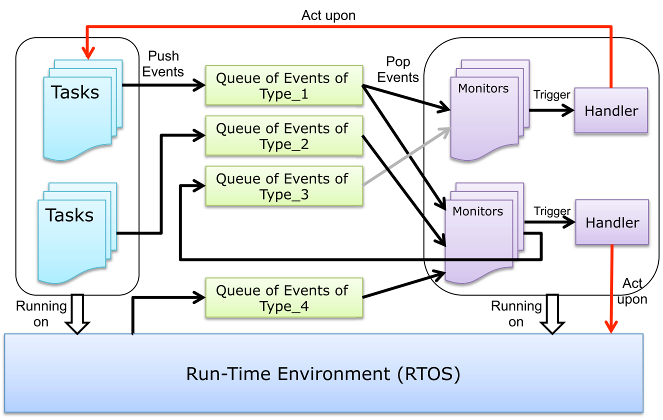
A. Code Instrumentation

The first aspect addressed by the proposed framework is the generation of *events* that may be used by the run-time monitors to check properties. An *event* is defined as a set of conditions that must be encountered during the system execution. In a software application, an event corresponds to the passage by a specific point in the code. We refer to an *event instance* as the realisation of the associated event. An event instance is therefore characterised by a unique timestamp.

The system designer can define a set of custom events by specifying the program locations at which they should be generated. Data (e.g., task ID, current execution state or value of a counter) that must be recorded with each event instance can also be specified. This step can be done relying on a light version of Aspect Oriented Programming [9], [10]. Each event definition is



(a) Run-time verification framework overview.


Act upon
Push Events
Tasks
Queue of Events of Type_1
Pop Events
Monitors
Trigger
Handler
Queue of Events of Type_2
Queue of Events of Type_3
Tasks
Monitors
Trigger
Handler
Queue of Events of Type_4
Act upon
Running on
Running on
Run-Time Environment (RTOS)

(b) Reference architecture for runtime monitoring.

automatically translated in a few lines of code that record the event instances together with their timestamps and potentially associated data. Those probes are then injected at the correct location in the application code.

The overhead generated by the recording of each event instance is constant and independent on the number of monitors using them, the number of tasks in the application or the number of event generated by the system.

B. *Monitor Generation*

The monitor generation starts by the definition of specifications the system must comply with. The specifications use the events defined in the previous section. They are written using non-ambiguous formal languages such as time regular expressions, linear temporal logic and finite state machines. Those already well studied languages are extended so as to consider properties specific to real-time systems such as jitter on temporal properties or the execution time of a job or part of a job.

The system designer can associate actions with the success or failure of each specification. For instance, one can trigger the deactivation of some tasks when a job misbehaves, thereby following the *graceful degradation* approach recommended by safety-related standards [6]–[8].

Each specification is automatically translated in a complex timed automata. This intermediate representation is then used as an input for a correct-by-construction code generation whose output is a periodic task reading the event instances recorded by the application and verifying whether the system respect the specification.

C. *Integration in Real-Time Safety-Critical Systems*

A novel run-time monitoring architecture specifically targeting real-time safety-critical applications (see Figure 1b) will be presented in details during the talk. This architecture that defines how monitors and monitored application interact has been designed to enforce both reliability and efficiency. More specifically, it covers the following aspects:

- It allows an independent and composable development, where individual components remain unchanged when considered in isolation and after their integration with all the other components constituting the system.
- It enforces time and space partitioning, in order to isolate monitors and monitored applications, as this ensures that the newly introduced monitors do not modify the behaviour of the monitored applications, and that a fault happening in the monitored application does not propagate to the monitors.
- It ensures efficiency and responsiveness, since an efficient implementation is required to ensure that information about event realisations reach the monitors as soon as needed.

Each monitor is implemented as a periodic task reading event instances from circular buffers. There is one such buffer per event type and only one task can record event instances in a buffer. This restriction ensures a complete time isolation between tasks but requires the monitors to reorder event instances of different buffers before to be able to use them. The buffers play the role of an interface between the monitors and the event generators, thereby allowing a complete space partitioning (e.g., partitioned scheduling and memory partitioning) between monitors and monitored application.

Acknowledgements

This work was partially supported by National Funds through FCT (Portuguese Foundation for Science and Technology) and by ERDF (European Regional Development Fund) through COMPETE (Operational Programme 'Thematic Factors of Competitiveness'), within project FCOMP-01-0124-FEDER-037281 (CISTER), and by National Funds through FCT and the EU ARTEMIS JU funding, within projects ref. ARTEMIS/0003/2012, JU grant nr. 333053 (CONCERTO) and ref. ARTEMIS/0001/2013 (JU grant nr. 621429 - EMC2).

References

[1] H. Barringer, K. Havelund, D. Rydeheard, and A. Groce, "Runtime verification," ch. Rule Systems for Runtime Verification: A Short Tutorial, pp. 1–24, 2009.

[2] F. Chen and G. Roşu, "Mop: An efficient and generic runtime verification framework," in *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA, pp. 569–588, 2007.

[3] P. Meredith and G. Roşu, "Runtime verification with the rv system," in *Proceedings of the First International Conference on Runtime Verification*, pp. 136–152, 2010.

[4] M. Kim, M. Viswanathan, S. Kannan, I. Lee, and O. Sokolsky, "Java-mac: A run-time assurance approach for java programs," *Form. Methods Syst. Des.*, vol. 24, pp. 129–155, Mar. 2004.

[5] K. Sen, G. Rosu, and G. Agha, "Runtime safety analysis of multithreaded programs," *SIGSOFT Softw. Eng. Notes*, vol. 28, pp. 337–346, Sept. 2003.

[6] IEC61508, *Functional safety of electrical/electronic/programmable electronic safety-related systems*. IEC, 2010.

[7] DO-178C, *Software Considerations in Airborne Systems and Equipment Certification*. RTCA, Inc., 2011.

[8] ISO26262, *Road vehicles Functional safety*. ISO, 2011.

[9] G. Kiczales, "Aspect-oriented programming," *ACM Comput. Surv.*, vol. 28, Dec. 1996.

[10] I. Kiselev, *Aspect-Oriented Programming with AspectJ*. Indianapolis, IN, USA: Sams, 2002.