Run-time mechanisms for property preservation in high-integrity real-time systems*

Juan Zamorano
Universidad Politécnica de Madrid (UPM), Spain
jzamora@datsi.fi.upm.es

Juan A. de la Puente
Universidad Politécnica de Madrid (UPM), Spain
jpuente@dit.upm.es

Jérôme Hugues
GET-Télécom Paris – LTCI-UMR 5141 CNRS, France
hugues@infres.enst.fr

Tullio Vardanega
University of Padova, Italy
tullio.vardanega@math.unipd.it

ABSTRACT

Classical real-time kernels tend to leave to the application level the burden of policing those stipulations that the designer deems crucial to warrant the correct operation of the system. In fact, in the general case, there exist forms of reflective computing at application level that may be happy with that arrangement. Where "continuity of proof" and "preservation of properties" are central to the development paradigm instead permissive kernels are arguably inferior to proactive execution platforms which are capable of: (i) policing the critical stipulations; and (ii) preventing their violations at run time. In this short paper we illustrate some constructive principles of an execution environment that follows the latter paradigm.

1. INTRODUCTION

One distinctive objective pursued by the ASSERT project (cf. footnote 1) is to attain "preservation of properties" throughout the entire development process, from system modeling down to run-time execution. A crucial ingredient to attain this objective entails the design and implementation of an execution environment that exhibits two fundamental features:

1. to ensure that the run-time entities (e.g. threads, locks, queues, etc.) that are deployed to implement the system model do have *exactly* the same semantics as was assumed in the verification and validation of the system model
2. to ensure that the run-time attributes which decorate all elements of the system model are also attached, without semantic distortion, to the run-time entities that implement them and, when they designate *stipulations* (e.g. a given budget, whether in time for execution, in size for storing, or in bandwidth for communication) they be actively policed during execution so that no single violation of them may either occur or go unnoticed.

*This work has been funded in part by the Sixth Framework Programme of the European Commission under project FP6-IST-2004 004033 (ASSERT).

Meeting requirement 1 in a guaranteed manner is considerably facilitated if visibility of the kernel API is hidden away from the designer so that calls to it can only be issued at places strictly controlled by the development process. (In the context of Model-Driven Engineering, for example, those places would be determined by the model transformation logic as opposed to by manual programming, as it is still often the case in the development of high-integrity real-time systems.)

To satisfy requirement 2, instead, the execution platform that we have designed for use in ASSERT had to be rigidly inflexible in hosting, executing and actively policing the run-time behaviour of the allowable run-time entities. In order to underline its distinctive character, which sets it aside from classical real-time kernels (which tend to be permissive when it comes for the policing of stipulated run-time behaviour), we have chosen to name our execution platform: "Virtual Machine" (VM), with a connotation that intends to evoke the correct interpretation of an intended semantics and its active enforcement at run time.

The ASSERT VM concept exhibits a number of important characteristics:

a) it is a run-time environment that only hosts and supports "legal" entities, i.e., those that are explicitly retained as the target of the automated model transformation process which realizes the system specification (the equivalent of a PIM) in terms of correct-by-construction aggregates and interconnections of allowable run-time entities (the equivalent of a PSM);

b) it provides run-time services that aid those run-time entities to actively preserve their designated proper-

ties; mechanisms and services of interest allow for instance to:

- accurately measure the actual execution time that can be attributed to individual threads of control
- attach and replenish a monitored execution time budget to a thread, and then prompt an alarm when the thread exceeded its time budget
- segregate threads into distinct groups, attaching a monitored budget to individual groups, to be handled in the same way as for threads
- enforce the minimum inter-arrival time stipulated for sporadic threads
- build fault containment regions around individual threads and groups thereof
- attain distribution and replication transparency in inter-thread communication.

c) it is bound to a compilation system that only produces executable code for "legal" entities and rejects the non-conforming ones; run-time checks provided by the virtual machine shall cover the extent of enforcement that cannot be exhaustively achieved at compile and link time

d) it realizes a concurrent computational model provably amenable to static analysis; the model must permit threads to interact with one another (directly, by some form of synchronization, and indirectly, by preemptive interference) in ways that do not incur non-determinism.

It is worth noting that the concept that underpins the VM arguably goes beyond the current state of the art (cf. e.g. [1]) in that it incorporates more than just overrun detection, but rather a whole range of features and mechanisms that actively "police" the continued compliance of the system behaviour at run time to its specification. In the remainder of this short paper we shall briefly discuss some of the most noteworthy features of the ASSERT VM in this respect.

2. DISTINGUISHING FEATURES OF THE VIRTUAL MACHINE

In order to comply with the required characteristics, the ASSERT VM semantics is based on the Ravenscar computational model [2], a concurrency model enabling predictable behaviour. Legal entities at the program code level are those accepted by an Ada 2005 compiler restricted by the Ada Ravenscar profile [9].

2.1 Activation of sporadic tasks

A foremost property to be preserved for any sporadic task is the minimum interval time that is stipulated to occur between any two successive activations (a.k.a. minimum interarrival time). This property is essential in ensuring the feasibility of response-time analysys, for its violation may result in unexpectedly high interference for lower-priority tasks [3].

The ASSERT system generation process enforces this property by using an Ada pattern based on a `delay until` statement, as shown in [7]. The semantics of this statement is supported by the VM using a delay queue, in a similar way as was done in the ORK kernel [11].

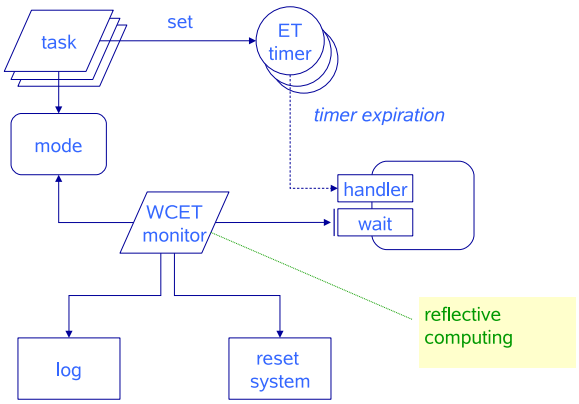

Figure 1: Execution-time monitoring.

2.2 Execution-time monitoring

Execution-time monitoring is a means to prevent execution-time overruns from occurring. Overruns may be caused by erroneous estimation of the worst-case execution time (WCET) of a task, or by some misbehaviour resulting in the task executing for longer than stipulated. In either case appropriate corrective actions must be taken to preventing the offending task from jeopardizing the temporal behaviour of other tasks in the system.

Execution-time monitoring in the ASSERT VM is based on execution-time timers, a specific feature of the new Ada 2005 standard [9]. Each task has an execution-time clock which only advances while the task is actually executing. Based on it, an execution-time timer can be defined which can be set to a time interval or to an absolute value. If the timer expires, a protected procedure handler is executed in a way similar to an interrupt handler.

Execution-time timers are not currently allowed in the Ravenscar Profile [9]. However, in view of their strategic interest to the objectives of the project, the ASSERT VM implements a minimal extension to the Ravenscar Profile by including at most one statically declared timer per task. The code archetypes for execution-time monitoring (cf. listing 1 for an example) use one such timer to detect WCET overruns.

When an overrun is detected, a high-priority monitoring task is released to recover from the error (cf. figure 1). Various forms of corrective measures can be contemplated within the confines of the Ravenscar Profile: our current orientation is to force a mode change to the offending task, which will take effect at the next activation, assuming that the task will end its current violating execution without needing external intervention. (While this assumption may be over-optimistic in the general case, we reckon it is not in the case of ASSERT which pursues a zero-programming development paradigm and thus should be less exposed to erroneous application code.)

2.3 High-integrity distribution

Adding distribution features to High-Integrity (HI) systems such as those targeted by ASSERT requires first to categorize the possible causes of software failures that may stem from distribution, and then to define the set of preventive

Listing 1: WCET overrun detection.

```
My_Identity : aliased constant Task_Id := Periodic_Task'Identity;
WCET_Timer  : Ada.Execution_Time.Timers.Timer(My_Identity'Access);

task body Periodic_Task is
  Next_Activation : Ada.Real_Time.Time := Epoch;
begin
  loop
     WCET_Timer.Set_Handler
       (In_Time=>My_WCET_Budget,
        Handler=>My_Monitor.Overrun_Handler'Access);
    delay until Next_Activation;
    Do_Actual_Work;
    Next_Activation := Next_Activation + My_Period;
  end loop;
end Periodic_Task;
```

measures which may exclude them. We in particular selected three typical concerns of distribution middleware and took special care in addressing them in a HI setting by combining run-time mechanisms and modeling artifacts: 1) no dynamic memory allocators; 2) no dynamic skeleton dispatchers; 3) prevention of overruns on both the client and the server side.

2.3.1 No dynamic memory allocators

Dynamic memory allocation typically occurs when the developer does not know beforehand how much memory it takes to perform the required actions. In a middleware setting, this situation usually arises when buffers must be allocated to: handle incoming requests; initialize connections; or store internal data to manage internal state information.

In fact, all of those situations and the relevant needs can be deduced from a careful analysis of the application model interfaces and the intended system topology. Further assistance of course comes from imposing suitable restrictions on those elements:

1. interfaces are restricted to only use types of bounded size so that the maximum size of buffers needed to support each remotely invokable function can be computed statically
2. prior knowledge about the application topology (i.e., direction and number of connections) permits to statically precompute the required tables of naming information and to store them at elaboration time at all places where they are needed.

By combining restrictions (1) and (2) one becomes able to statically allocate all required resources at compile time, thereby renouncing the need for run-time allocation.

2.3.2 No dynamic skeleton dispatchers

Skeleton dispatchers are required to map network messages onto the call to designated local procedures. To avoid incurring unpredictability in both time and space we infer from the system model the information required to allocate static arrays of dispatchers so that the incoming requests can be directly marshalled into one index in the store. It is worth noting that the latter step is performed in $O(1)$ time, in contrast with standard CORBA middleware, which requires a string-to-index mapping and thus incurs the overhead of hashing [8].

2.3.3 Preventing overruns

The prevention of overruns (which in a distributed context may result in denial of service situations) is a major safety problem. Situations of this kind happen when either a client sends too much data (the "babbling client"), or a server receives more requests than it is able to process. These situations are handled by transport-level primitives of the ASSERT VM middleware respectively by:

1. preventing the client to send more than a given N number of requests per time unit, for a grand total of k bytes in that time interval, using a simple timer and an a byte budget counter;
2. stopping the server from being too responsive to incoming requests, by disabling the monitoring of a I/O source, or by ignoring specific connections in case the device permits it;
3. ensuring that only authorized clients interact with the server. The server will silently ignore all requests coming from other clients.

Measures 1 and 2 are supported by the transport stack, which maintains a table of resource consumption and uses monitoring techniques similar to those discussed in section 2.2 for task execution time. The exact definition of what corrective actions should be performed in the face of a violation follows from thorough analysis of the application needs, with the intent of permitting nominal operation to continue undisturbed while also ensuring that violations are trapped and not permitted (to continue) to occur.

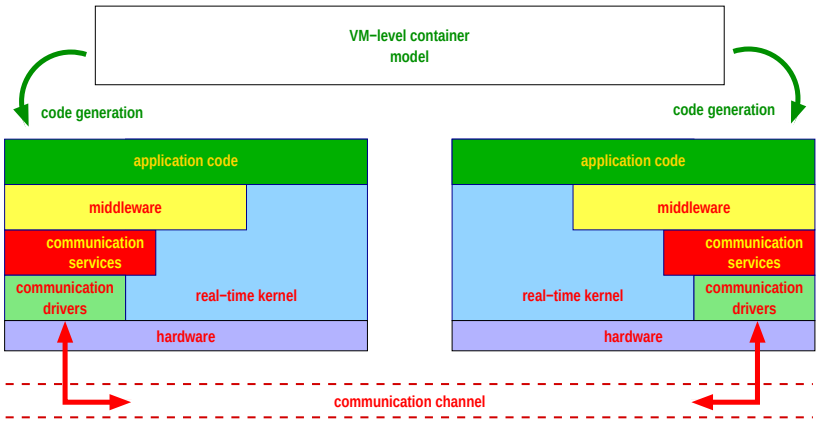

Figure 2: Top-level VM architecture.

Measure 3 simply entails a run-time check to be performed into the name tables built for each node.

3. IMPLEMENTATION STATUS AND LESSONS LEARNED

An implementation of the ASSERT VM for the LEON2 computer architecture [5] has been almost completely achieved to date, in full keeping with the principles discussed in this paper. The VM architecture includes (cf. figure 2):

- A real-time kernel realized as an evolution of ORK [10] and integrated with the GNAT for LEON compiler[1].
- A communications stack for the SOIS Message Transfer Service (MTS) [4].
- A middleware layer based on PolyORB-HI [6].

The completion of the implementation is due in early June 2007 and its use in a suite of end-of-project industrial case studies shall start immediately after that, with culmination in an integrated public demonstration scheduled for late November 2007.

Earlier versions of the ASSERT VM were distributed to the industrial partners in the project and used to explore the implementation of exploratory elements of the intended case studies. Feedback from industrial use is very encouraging. The implementation effort required to date and expected for completion has proven to be ordinate and, thus perfectly acceptable. The early performance figures (with regard to timing and sizing particularly) also seem encouraging though a more thorough assessment of them will most certainly come from the planned case studies.

Overall the message we wish to bring to the reader is that the strategic direction we have taken in the project seems not only intellectually attractive but it also appears to be relevant to industry as well as practical to engineer and use.

[1] `www.adacore.com`

4. REFERENCES

[1] S. Brandt, S. Banachowski, C. Lin, and T. Bisson. Dynamic Integrated Scheduling of Hard Real-Time, Soft Real-Time, and Non-Real-Time Processes. In *Proceedings of the Real-Time Systems Symposium*, pages 396–409. IEEE, December 2003.

[2] A. Burns, B. Dobbing, and T. Vardanega. Guide for the use of the ada ravenscar profile in high integrity systems, 2003.

[3] A. Burns and A. J. Wellings. *Real-Time Systems and Programming Languages*. Addison-Wesley, 3 edition, 2001.

[4] Consultative Committee for Space Data Standards (CCSDS). *CCSDS Spacecraft On-board Interface Services Green Book – CCSDS 830.0-G-0.4*, Dec. 2004. Draft.

[5] Gaisler Research. *LEON2 Processor User's Manual*, 2005.

[6] J. Hugues, B. Zalila, and L. Pautet. Middleware and tool suite for high integrity systems. In *Proceedings of RTSS-WiP'06*, pages 1–4, Rio de Janeiro, Brazil, December 2006. IEEE.

[7] J. A. Pulido, S. Urueña, J. Zamorano, and J. A. de la Puente. Handling temporal faults in Ada 2005. In N. Abdennadher and F. Kordon, editors, *Reliable Software Technologies — Ada-Europe 2007*, number 4498 in LNCS, pages 15–28. Springer-Verlag, 2007.

[8] I. Pyarali, C. O'Ryan, D. Schmidt, N. Wang, W. Kachroo, and A. Gokhale. Applying optimization principle patterns to desgin real-time ORBs. In *Proceedings of the 5th USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*. USENIX, 1999.

[9] S. T. Taft, R. A. Duff, R. L. Brukardt, E. Plöedereder, and P. Leroy, editors. *Ada 2005 Reference Manual. Language and Standard Libraries. International Standard ISO/IEC 8652/1995(E) with Technical Corrigendum 1 and Amendment 1*. Number 4348 in Lecture Notes in Computer Science. Springer-Verlag, 2006.

[10] S. Urueña, J. A. Pulido, J. Zamorano, and J. A. de la Puente. Adding new features to the Open Ravenscar Kernel. In *1st International Workshop on Operating*

Systems Platforms for Embedded Real-Time Applications (OSPERT 2005), Palma de Mallorca, Spain, July 2005.

[11] J. Zamorano, J. F. Ruiz, and J. A. de la Puente. Implementing Ada.Real_Time.Clock and absolute delays in real-time kernels. In A. Strohmeier and D. Craeynest, editors, *Reliable Software Technologies — Ada-Europe 2001*, number 2043 in LNCS, pages 317–327. Springer-Verlag, 2001.