Ada 2005 Code Patterns for Metamodel-Based Code Generation [*]

José A. Pulido, Juan A. de la Puente
Universidad Politécnica de Madrid (UPM), Spain
{pulido,jpuente}@dit.upm.es

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

Matteo Bordin, Tullio Vardanega
Università di Padova, Italy
{mbordin,tullio .vardanega}@math.unipd.it

Abstract

In this paper we discuss the issues we have explored, as part of the ASSERT project, in the definition and implementation of Ada coding patterns for the support of the automated code generation stage of a comprehensive approach to model-driven development for high-integrity systems.

1. Introduction

In the ASSERT project we explored the feasibility, practicality and performance of a Model-Driven development (MDA) approach targeted to high-integrity applications. A crucial element of our vision relied on the definition and exploratory implementation of a strategy for the automated transformation of the system model down to source code and to execution in a manner capable of warranting the preservation of the properties stipulated at model level.

To support system modeling we introduced two types of containers that are distinct for use but very strictly related when it comes to applying model transformation. One type of container, which we call "*application level container*" addresses functional modeling, where the level of expression ought to be abstracted away from the intricacies of hard real-time concurrency (much in keeping with the platform-independent model, PIM, of MDA). The other type of container, which we name "*virtual machine level container*", designates entities that exist at run time with an intended semantics that must be actively preserved by the execution platform. In the following we shall denote the former by "*APLC*" and the latter by "*VMLC*". We then use the term "*virtual machine*" (VM) to express the kind of execution platform we require to warrant property preservation in the trajectory from model to execution.

In our vision, designers operate on APLC only. Such containers however are not executable, while VMLC are. There must therefore exist some logic that transforms the former in the latter in a trustworthy manner, without any semantic distortion and with active preservation and enforcement of properties. Such a transformation takes the model from a higher level of abstraction to a lower one, closer to implementation and execution. Hence we regard it as a "vertical transformation", as opposed to other forms of model transformation in which the representation formalism changes while the level of abstraction need not. Our principle of vertical transformation rests on the postulate that the attributes set on APLC (which denote component interfaces and relations among them) consistently map to groups of interconnected VMLC endowed with the required interfaces.

The platform must be rigidly inflexible (as opposed to permissive) in hosting, executing and actively policing the run-time behaviour of the container entities that may legally exist at its level. In fact, our VM concept entails the following characteristics:

1. it is a run-time environment that only accepts and supports "legal" entities; the sole legal entities that may be propagated down from model transformation are VMLC; no other run-time entity is permitted to exist and no other can thus be assumed in the model
2. it provides run-time services that assist containers in actively preserving their designated properties; mechanisms and services of interest may for instance aid 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

[*]This work has been funded in part by the IST Programme of the European Commission under project IST-004033 (ASSERT).

(e.g.: an exception) when the thread should exceed 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.

3. 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 VM shall cover the extent of enforcement that cannot be exhaustively achieved at compile and link time

4. 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.

2. Virtual Machine Architecture

To date, we consider that the ASSERT VM should be constituted by the integration of the following key elements, shown in figure 1.

- *Real-Time kernel*: it assigns CPU time to threads, providing tasking and synchronization primitives. It also supports some mechanisms for device drivers to access the hardware elements of the targets;

- *Middleware*: it provides the necessary abstractions to support communication between nodes of the application. These abstractions are defined as part of the ASSERT Distribution model.

- *Communication subsystem*: it provides low-level support for inter-partition communications relying on the deterministic SpaceWire protocol [6];

All of those elements are individually defined in full compliance with both the Ravenscar profile and the restrictions set by the High-Integrity systems annex of Ada 2005 [11]. The key high-level restrictions that we placed on them are as follows:

2.1. Real-time kernel

The real-time kernel component of the ASSERT VM is an upgraded version of the Open Ravenscar Kernel (ORK) [2, 4]. It provides direct support for the Ravenscar profile [11, D.13] *and* includes the following Ada 2005 extensions:

- global timing events;
- execution-time clocks;
- system-wide (fallback) teask termination handler.

The kernel also supports execution-time timers and group budgets. Although not allowed in the Ravenscar profile, those mechanisms help enforce temporal separation between subsystems with possibly different levels of criticality, which is a strong requirement for the kind of on-board aerospace embedded systems envisaged in the ASSERT project. Execution-time timers and budgets are used to implement hierarchical scheduling of subsystems as a means to provide temporal isolation [9]. The kernel only allows one execution-time timer per task, as suggested in previous IRTAW discussions [5, 3].

The ASSERT VM kernel is integrated with the GNAT compilation system.

2.2. Middleware

The distribution layer of the ASSERT VM (the "middleware") is placed at the interface between the application code and the underlying kernel. It maps high-level interactions onto calls to low-level primitives that support distribution in nearly-transparent manner. The subset of requirements of the Middleware that affects the kernel is well defined. Those that most impact the design of the Middleware component of the ASSERT VM are as follows:

- The middleware inherits and restricts the process and data models of the real-time kernel and it is therefore able to provide adequate support for timeliness and predictability of service.
- Distribution transparency: the ASSERT VM should rely on the process model and provide the infrastructure and constructs required to provide for distribution transparency. As a result of this provision, the ASSERT designer should not need to modify source code to deploy different configurations of the system.

The communication services that interact with the physical interconnection might also impose some requirements to be fulfilled by the middleware. The analysis of its full impact is currently in progress, due to complete by Q3 2007.

As an adaptation layer, the middleware shall limit its weight to avoid any unjustified overhead, while preserving structural properties and proof preservation.

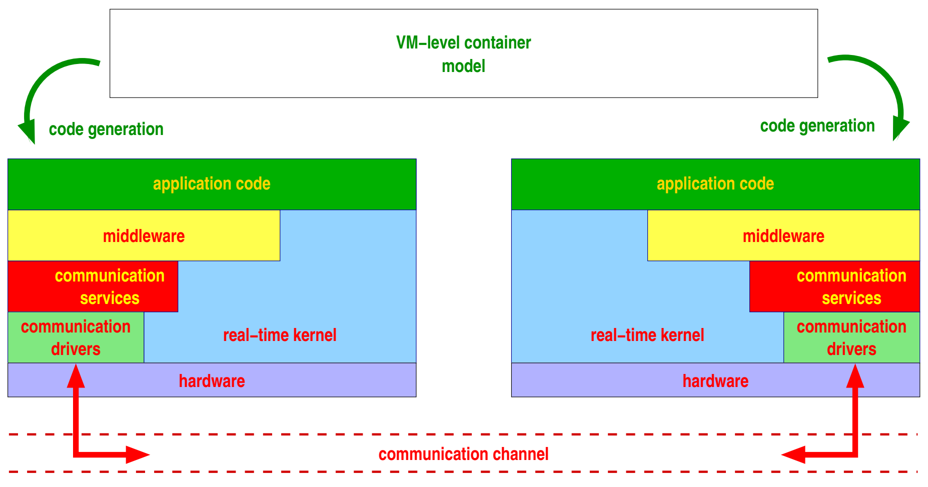

Figure 1. Top-level VM architecture.

3. Code Patterns for local VM-Level Containers

Local VMLC are run-time entities that provide direct support for concurrent real-time behaviour based on the Ravenscar computation model. They are directly implemented by primitive services of the ASSERT VM, using the above described extension of the Ada Ravenscar profile. Four basic types of VMLC have been defined, based on the hard real-time terminal objects of HRT-HOOD [1]: cyclic, sporadic, protected, and passive containers.

VMLC are implemented using some meta-model elements, which are named after HRT-HOOD entities (figure 2):

- Provided and required interfaces,
- OPCS (operation control structure), embodying the functional behaviour of the component;
- OBCS (object control structure), providing synchronization mechanisms;
- Thread, providing the concurrent behaviour of the component.

The real-time and synchronization behaviour of a VMLC is provided by the OBCS and the thread. The OBCS is implemented by an Ada protected object, while the thread is implemented by an Ada task.

Code patterns that represent the common aspects of a class of model elements are called *archetypes*. Examples of archetypes are:

- Cyclic object, with or without deadline and overrun detection;
- Sporadic object, with or without deadline and overrun detection;

Appendix A includes some examples of such archetypes. More complex archetypes can built by combining these code patterns, e.g. a cyclic thread with both deadline miss and WCET overrun detection.

A key issue is how to handle temporal faults. Possible fault handling patterns include:

- *Error logging*. Although useful for testing or low integrity applications, this pattern is not suited for high-integrity tasks as it does not provide any kind of resilience to faults.
- *Second chance*. If there is some slack time available, the execution-time budget of the failing task can be extended for the current job. This is seldom acceptable in high-integrity systems, as it incurs some degree of temporal indeterminacy.
- *Mode change*. This is the most flexible approach in that it permits fault recovery to be performed as and when required.
- *Safe stop*. This strategy is often the only course to take for high-integrity systems.

A more extensive discussion of these schemes can be found in an upcoming paper [8].

4. Extending Patterns for Distribution

In this section we review Ada coding patterns for VMLC in the distributed case.

The support of distribution encompasses a variety issues that range from high-level middleware architecture down

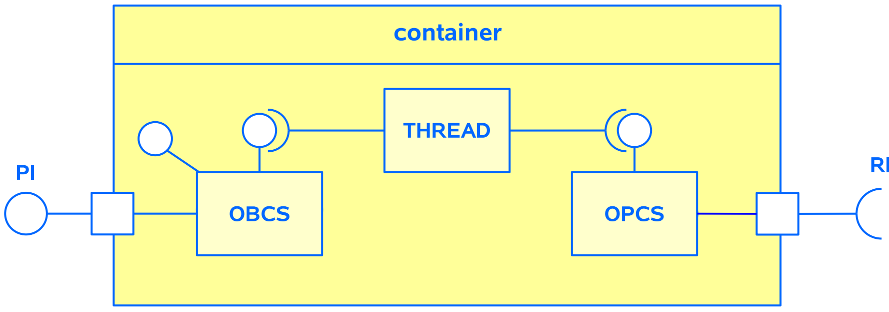

Figure 2. VM-level container meta-model

to implementation concerns. Some of those issues are already discussed from an Ada perspective in PolyORB [13]; GLADE [7] and RT-GLADE projects. For the purposes of this paper we focus on interaction patterns, which are the most critical elements to warrant system analyzability.

We make the following hypotheses: a high-level model describes the deployment of the system; this view is used to build and dimension all tasks and buffers, in-place marshallers are built; a deterministic protocol is used [1]. Hence, we focus on how interaction patterns defined in the Ravenscar profile can be mapped onto interaction suitable for distribution.

We assume interactions are asynchronous and one-way, which is a sensible choice for Distributed Real-Time systems for it reduces global blocking time. Besides, this is compatible with typical requirements from critical systems and typical scheduling techniques such as [12] and its extensions. We assume that a one-to-one relation exists between the priority of the request (the message to be exchanged) and the priority of the processing tasks.

The Ravenscar profile prescribes that inter-task communication take place via protected objects. Extending this configuration to distribution can be as simple as "splitting" the protected object in two separate entities: one client-side activity that formats the request and passes it to the communication subsystem; one server-side activity that is notified a request is arriving. Depending on the enforcement policies of choice, a task can be released from a thread pool, or else a dedicated task is awakened. Let us briefly review each configuration in isolation (figure 3):

- **Client Side**: this configuration is similar to the local case: the client passes its request to the communication subsystem that sends this request to the remote node. Intermediate steps are required to marshall the payload, and to build proper messages. Depending on the protocol stack, an intermediate protected object might be placed between client code and the protocol-specific threads.

[1]Please refer to the authors bibliography for more details on these hypotheses.

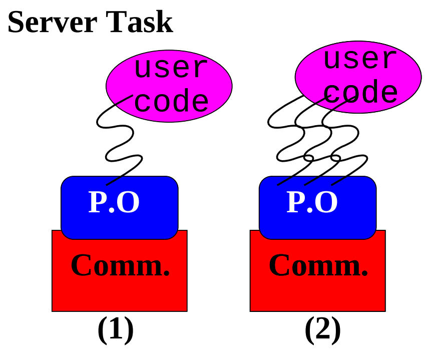

Figure 3. Request Handling Strategies

- **Server Side**: on the server side, the request is received by the communication subsystem and stored in some internal buffer. It is safe to assume that this buffer is embodied in a protected object (PO). Several scenarios are then possible:
  - *One PO per receiving task (1)*: this scenario is similar to the local case where at most one task is waiting per protected object;
  - *One PO plus a pool of tasks (2)*: this scenario implies that a task is waiting for the next request while some other tasks are either processing jobs or idling, or else blocked on a task-specific PO. This arrangement corresponds to the *Leader/Followers* or *Thread Pool* patterns [10]. In such a scenario it may be convenient to use different priorities for I/O and message sending as well as for request processing tasks, for example to discriminate between urgency and priority.

Furthermore, one must select patterns to dispatch to the correct subprogram, with two strategies *one PO per subprogram*, or *one PO shared by several subprograms* and skeleton-like code to dispatch to the correct subprogram.

Each strategy has been implemented and made deterministic separately by using local Ravenscar patterns and a transport library. Those two elements can then be combined to form four different configurations. The selection of one particular configuration has to be made depending on two antonymous criteria: footprint size of the system vs. ease of analysis. At the time of this writing the ASSERT project team is still evaluating the costs and trade-offs of each configuration in term of pessimistic WCET and memory overhead.

5. Conclusions and future work

In this paper we have briefly outlined the motivations and the strategy we have adopted for the definition of Ada 2005 coding patterns for the support of the code generation stage of an advanced model-driven development infrastructure. We are presently approaching the final implementation stage of our vision and we shall soon have performance figures useful to ascertain the viability of the approach we pursue.

References

[1] A. Burns and A. Wellings. HRT-HOOD: A design method for hard-real-time. *Real-Time Sytsems*, 6(1):73–114, 1994.

[2] J. A. de la Puente, J. F. Ruiz, and J. Zamorano. An open Ravenscar real-time kernel for GNAT. In H. B. Keller and E. Ploedereder, editors, *Reliable Software Technologies — Ada-Europe 2000*, number 1845 in LNCS, pages 5–15. Springer-Verlag, 2000.

[3] J. A. de la Puente and J. Zamorano. Execution-time clocks and Ravenscar kernels. *Ada Letters*, XXIII(4):82–86, December 2003. Proceedings of the 12th International Ada Real-Time Workshop (IRTAW12).

[4] J. A. de la Puente, J. Zamorano, J. F. Ruiz, R. Fernández, and R. García. The design and implementation of the Open Ravenscar Kernel. *Ada Letters*, XXI(1), 2001. Proceedings of the 10th International Real-Time Ada Workshop.

[5] B. Dobbing and J. A. de la Puente. Session report: Status and future of the Ravenscar profile. *Ada Letters*, XXIII(4):55–57, December 2003. Proceedings of the 12th International Real-Time Ada Workshop (IRTAW 12).

[6] ESA/ESTEC. ECSS-E-50-12A SpaceWire - Links, nodes, routers and networks. Technical report, European Space Agency, 2003.

[7] L. Pautet and S. Tardieu. GLADE: a Framework for Building Large Object-Oriented Real-Time Distributed Systems. In *Proceedings of the 3rd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'00)*, Newport Beach, California, USA, June 2000. IEEE Computer Society Press.

[8] 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.

[9] J. A. Pulido, S. Urueńa, J. Zamorano, T. Vardanega, and J. A. de la Puente. Hierarchical scheduling with Ada 2005. In M. G. H. Lus Miguel Pinho, editor, *Reliable Software Technologies - Ada-Europe 2006*, volume 4006 of *LNCS*. Springer Berlin / Heidelberg, 2006. ISBN 3-540-34663-5.

[10] D. C. Schmidt and F. Buschmann. Patterns, frameworks, and middleware: their synergistic relationships. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 694–704, Washington, DC, USA, 2003. IEEE Computer Society.

[11] 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.

[12] K. Tindell. Holistic schedulability analysis for distributed hard real-time systems. Technical report, University of York, 1993.

[13] T. Vergnaud, J. Hugues, L. Pautet, and F. Kordon. PolyORB: a schizophrenic middleware to build versatile reliable distributed applications. In *Proceedings of the 9th International Conference on Reliable Software Techologies Ada-Europe 2004 (RST'04)*, volume LNCS 3063, pages 106 – 119, Palma de Mallorca, Spain, June 2004. Springer Verlag.

A. VM-level container archetypes

The code templates in the next pages illustrate the approach use in building component archetypes. More complex archetypes cab be built using this basic set.

Pattern 1. Cyclic thread archetype with deadline miss detection.

```
Deadline_Miss : Timing_Event;
task type Cyclic_Thread(P : Priority; T : Time_Span; D : Time_Span) is
  pragma Priority(P);
end Cyclic_Thread;
task body Cyclic_Thread is
  Next_Time : Time := Clock;
  Deadline_Missed : Boolean;
begin
  loop
    Deadline_Miss.Set_Handler(Next_Time + D, Deadline_Miss_Handler'Access);
    delay until Next_Time;
    OPCS.Activity; --  periodic job
    Next_Time := Next_Time + T;
  end loop;
 end Cyclic_Thread;
```

Pattern 2. Cyclic thread archetype with WCET overrun detection.

```
task type Cyclic_Thread(P : Priority; T : Time_Span; D : Time_Span; C : Time_Span) is
  pragma Priority(P);
end Cyclic_Thread;
task body Cyclic_Thread is
  Next_Time : Time := Clock;
  Id : aliased constant Task_Id := Current_Task;
  WCET_Timer : Ada.Execution_Time.Timers.Timer(Id'Access);
begin
  loop
    Set_Handler(WCET_Timer, C, Overrun_Handler'Access);
    delay until Next_Time;
    OPCS.Activity; -- periodic job
    Next_Time := Next_Time + T;
  end loop;
end Cyclic_Thread;
```

Pattern 3. Sporadic thread archetype with minimal inter-arrival time enforcement.

```
OBCS : Sporadic_OBCS(Ceiling);
task type Sporadic_Thread (P : Priority; T : Time_Span) is
  pragma Priority(P);
end Sporadic_Thread;
task body Sporadic_Thread is
  Next_Time : Time := Clock;
begin
  loop
    delay until Next_Time;
    OBCS.Get_Request;
    OPCS.Activity; -- sporadic job
    Next_Time := Next_Time + T;
  end loop;
end Cyclic_Thread;
procedure Start is
begin
  OBCS.Put_Request;
end Start;
```