COrDeT - Cannes Study
Non-Functional Code Generation

Output of WP602

Written by:	**Organization**	**Approval Status**
Marco Panunzio	UPD	30/05/2008
Tullio Vardanega	UPD	30/06/2008
Verified by:		
Tullio Vardanega	UPD	18/08/2008
Approved by:		
Gerald Garcia	TAS-F/Cannes	pending
Andreas Jung	ESA/ESTEC	pending

Document Change Record

Issue/Revision	Date	Change Record	Author
0.0	30/05/2008	Initial version	Marco Panunzio (UPD)
0.1	30/06/2008	First internal release	Tullio Vardanega (UPD)
0.2	11/07/2008	First external release	Tullio Vardanega (UPD)
0.3	18/08/2008	Revised after partners' review	Tullio Vardanega (UPD)

Abstract

This document is a guide to the current RCM code generation that covers the non-functional parts of the software architecture of a system design with the HRT-UML/RCM methodology defined in the ASSERT project. The generation strategy is designed to be able to accommodate the seamless insertion of functional (algorithmic) code either hand-coded or else produced by other generation means, so long as in compliance with the HRT-UML/RCM restrictions. Those restrictions are discussed in COrDeT Report WP503 which readers are advised to read first, *before* approaching the core of this document.

After a short introduction in which we recall the key concept of the RCM methodology, we discuss code generation. In particular we examine the code archetypes used by the generation engine, the complete structure of the various types of Virtual Machine-level Containers and the generation of Application-level Containers. A small example is used to comment various aspects of the generated code.

Acknowledgments. The authors of the document gratefully acknowledge the contribution of Matteo Bordin, former member of the ASSERT team at UPD, who was the principal designer of the current code generation strategy, for all the hints and the discussions about the code generation as reported in this document.

Contents

Chapter 1

Introduction

1.1 Overview of AP-level and VM-level Containers

Application-level Containers (APLC for short) are the main design entities in the RCM methodology. APLC are components that embed cohesive services and functional states and expose them in a controlled way through an interface.

An APLC exposes two kind of interfaces: the *provided interface* (PI) and the *required interface* (RI). The PI specifes the services that the APLC offers to other APLC components. The signature of these operations denotes ”what” is offered, whereas a set of other attributes *declaratively* determine the ”how”. The ”how” attributes determine for example whether any synchronization protocol is provided to protect the execution of an interface invocation in the face of concurrency or that the operation is executed by a dedicated thread of control on the callee side. In contrast, the RI specifies what the component needs from others in order to discharge its duty to the system. An RI is similar in nature to a PI, except that a PI specifies the services offered to others by the component of interest, whereas the RI specifies its needs.

Relations drawn between RI and PI are subject to compliance checks so that a PI satisfies an RI if and only if all the ”what” and ”how” wishes are matched by ”what” and ”how” obligations exposed by the corresponding PI. Attributes are set on the interfaces and not on the relations among them. In other words, it is the interface that statically determines the semantics of the invocation.

This choice intentionally differs from other model formalisms in which some attributes of an interface invocation determine the semantics of it. The wisdom of our choice is in that static analysis is considerably facilitated (for an acceptable loss of expressive power) when the invocation semantics is a static attribute of the *provided* interface instead of being a dynamic attribute of the invocation.

A functional specification is attached to each interface operation to determine its *sequential* behavior. The way an invocation to a PI operation actually activates a transition in the state machine that describes the behaviour of that operation is determined by the attributes attached to the method itself (the ”how”).

Figure 1.1 complements with an example what has been just exaplained. We are partially modeling a tiny producer-consumer system. The designer specifies the sequential behavior of the system with a formalism that includes interfaces, classes, state machines. We define an interface for the consumer (IConsumer) and an interface and a concrete class for the producer (IProducer and Producer respectively). In method Produce of class Producer we need to use method consume of interface IConsumer, and we specify this need in the design of the class. Then we define an APLC for the producer. We embed in this component a *functional state* that is typed to the class Producer.

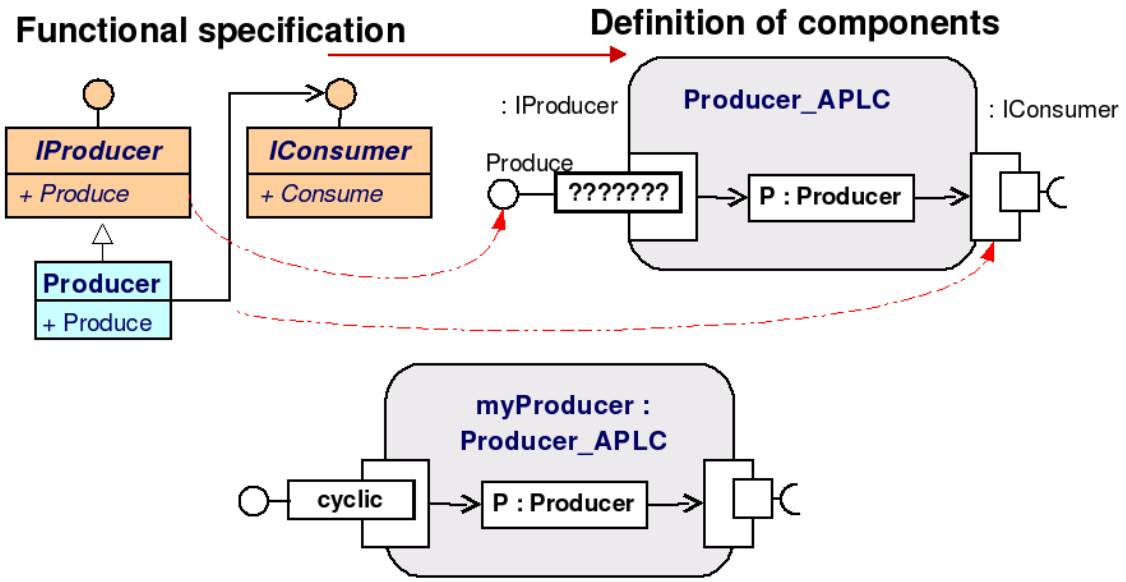

Figure 1.1: Modeling of AP-level Containers.

The resulting PI and RI of the APLC are automatically calculated since they directly derive from the functional specification of the embedded state. The PI exposes the *Produce* method, which in turn requires a method *Consume* exposed by interface IConsumer. That need is correctly reproduced in the RI of the Producer APLC.

The designer then completes the definition of the APLC operations specifying how its services are performed. In the example we specify that the operation *Produce* is to be executed periodically (cyclic concurrent kind).

APLC are platform-independent components used by the designer to specify the solution to the system problem. In fact the reader can note that neither references to any programming language nor to a concurrency model have been made in the previous example. For instance, the concurrent kind of the operation is only specified declaratively, but no implementation details on how that concurrent behaviour is achieved in the final system is provided by the designer, coherently with Model-Driven Engineering (MDE), which is the software engineering approach that inspires our methodology.

Being free of implementation details, APLC are defined in the space of the Platform Independent Model (or PIM, in the terminology of Model Driven Architecture). In the RCM development process, the PIM is *automatically* transformed in a Platform Specific Model (PSM), which conversely specifies all the information needed to implement the system. In RCM, the automatic transformation is instructed by a set of rules that attach to each Application-level Container one or more Virtual Machine-level Containers. Virtual Machine-level Containers (VMLC for short) are *platform-dependent* entities that implement the concurrent semantic requirements attached to the APLC services, in the form of a specific computational model.

In our design process we chose to use the Ravenscar Computational Model [BDV03], which emanates directly from the Ravenscar Profile [BDR98] of the Ada language [ISO05].

APLC and VMLC thus belong to distinct abstraction levels: the former provide a platform-independent specification of reusable software components, the latter ”implements” the APLC in manners that provably abide by the chosen computational model.

The general structure of a VMLC comprises the following three entities:

- the Operation Control Structure (OPCS), the primitive container that realizes one or more PI operations (as many as operate on one and the same functional state of the encompassing container)
- the Object Control Structure (OBCS), which embeds an agent of the synchronization protocol specified in the relevant PI attributes
- the Thread, which is a thread of control that executes the invoked PI operations in coordination with the synchronization protocol agent embedded in the OBCS.

Not all VMLC need to exhibit all those three constituents. The allowable structure of each possible kind of VMLC is discussed in chapter 2.

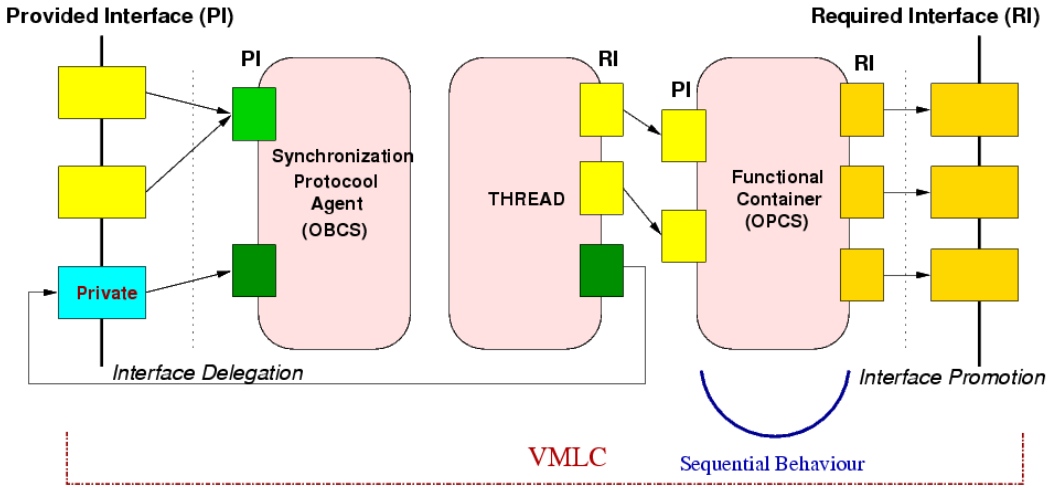

Figure 1.2: General structure of an APLC realized by a single VMLC.

The general structure of an APLC and its implementing VMLC is depicted in figure 1.2. Often multiple VMLC are attached to a single APLC, but for the sake of simplicity we present in that figure the simplest case (1 APLC realized by 1 VMLC).

Any legal vertical transformation effectively delegates the PI of an AP-level container to the matching PI of the target aggregate of VMLC. Similarly, it must ensure that the RI that those VMLC promote does match the RI of the corresponding APLC. The reader should recall that the intra-component relation between PI and RI is defined while specifying the functional behaviour of the system (which resides in the OPCS).

In the RCM the following types of VM-level containers are estabilished:

1. *Passive VMLC*: it is a primitive run-time entity with PI void of any synchronisation protocol.
2. *Protected VMLC*: it is a primitive run-time entity with synchronization control on access to its PI, following the ”Priority Ceiling Protocol” (PCP) [SLR86] (or, equivalently, the ”Stack Resource Policy” (SRP) [Bak91] with dispatching policies other than fixed-priority preemptive were used in the relevant partition). The use of any of those synchronization protocols warrants structural absence of deadlocks induced by cumulation of resources, minimisation of priority inversion and occurrence of blocking time at most one time per thread activation.

3. *Cyclic VMLC*: it is an aggregate of run-time entities that includes a thread of control which issues jobs at a fixed constant rate. The event that triggers the activation of a job of that thread is produced directly by the system clock.

4. *Sporadic VMLC*: it is an aggregate of run-time entities that includes a thread of control which executes job sporadically, i.e., with a stipulated minimum separation time between successive invocations, which is also guaranteed at run time. The event that triggers the activation of a job of that thread is produced by software (an interrupt handler or some other thread).

This general structure depicted in fig. 1.2 is the result of several architectural choices.

The reader should note that there is a single access point for the APLC and its underlying set of VMLC, that is the APLC *provided interface*: thus we earn encapsulation of the overall aggregation of entities. Each service of the PI is further equipped with a specific visibility attribute, which may further restrict the accessibility of the service to other entities that populate the system.

The three primitive structures that we have identified (OBCS, Thread, OPCS) have specific design goals, and their separation promote and facilitate factorization. The (optional) OBCS caters for protection against concurrent access to the operation and in deferred operation provides the mechanisms to *reify* operation invocations in a manner akin to what is done in classical middleware. The (optional) Thread is an executor of reified requests of invocations. For this reason, a Thread always requires an OBCS in the same VMLC from which to fetch the requests of execution. The OPCS is the entity where the sequential behaviour of the VMLC resides. That sequential behaviour is specified in the Functional View using UML2 formalism: interfaces, classes, state machines. The static structure of the OPCS simply exposes the PI and (possibly void) RI as specified in its class definition. What it is important is that the PI of the OPCS is *never* exposed to the system, but it is only reachable in a controlled form through the specification delegation chain that proceeds from the PI of the APLC. This property permits effective protection against concurrent access (which is provided by the OBCS) and the execution by a thread on the callee side (which is provided by the Thread). In theory, the OPCS can be used by multiple distinct VMLC and the sought separation from the synchronization agent and the executor becomes very valuable; in fact the *same* OPCS definition can be freely used in a threaded VMLC, in a protected VMLC and in a passive VMLC without any change to the sequential code.

Finally, the reader can note that the RI of the APLC is completely determined by the RI of its included OPCS: those RI are the sole RI of all the three primitive VMLC components that are promoted to the APLC level and become functional needs that must be completely satisfied in order for the APLC to discharge its services to the system.

We can now return to the allowable operations of APLC. In particular we examine the possible choices of concurrent semantics that each operation may exhibit:

1. *unprotected operation*: an operation that does not provide any protection against concurrent access and it is directly executed by the caller;

2. *protected operation*: an operation that is directly executed by the caller and that is subject to a synchronization protocol to govern concurrent accesses;

3. *cyclic operation*: an operation that is to be executed with a fixed period by a dedicated thread of control on the side of the callee; an operation of this kind can have no software caller;

4. *sporadic operation*: an operation that is to be executed sporadically, that is with a guaranteed minimum separation time between two subsequent executions; the operation is invoked by a software caller but it is execution by a dedicated thread of control on the side of the callee; the execution of an operation marked

"sporadic" is triggered by a condition (a guard, in technical terms) becoming true; the specification of the condition is associated with the specification of the "sporadic" attribute;

5. *modifier operation*: an operation that causes the executor thread to take an alternative behavior to the nominal one, whether cyclic or sporadic. In the current version of the code generation, the alternative behavior is one-off, i.e., executed only once before resuming nominal operation. It is easy to see however that modifier operations are also the most natural means for the implementation of *mode changes*.

The model-to-model transformation that instantiates VMLC to realize APLC uses the concurrent semantics of each operation (as well as information about the members of the functional states accessed by individual operation) to determine which *type* and *how many* VMLC are required to implement the intended concurrent semantics.

In domains like high-integrity real-time systems, the need often arises to reduce as much as possible the number of threads required at run time, to incur less space and time overhead. In RCM, modifier operations help meet this requirement. Modifier operations (or simply modifiers) are always coupled with a *nominal* operation. For each nominal operation, which may either be marked as *cyclic* or *sporadic*, there always exists a dedicated thread of control designated to perform their execution. Modifier operations represent an alternative sequential behavior to the *nominal* operation, which may be executed, at distinct invocations, by one and the same thread of control. In the current implementation, modifier operations feature exclusively a *one-off* behavior, meaning that an asynchronous request for a modifier operation replaces the *nominal* operation during the next activation of the executor thread. After that activation, the *nominal* operation is executed again.

1.2 Key principles and requirements of RCM

Abstraction and automation. MDE in general promotes reduced time and costs of system production, mainly by raising the abstraction level of the design space and enabling the automated generation of (portions of) the system code. In RCM the designer devises the solution to the system problem by: creating APLC; decorating their contractual interfaces; and binding them to one another so that RI as satisfied by matching PI. The designer need not decompose the system in low-level primitive entities: automatic model transformations generates intermediate artefacts which comply with the chosen computational model and are used to feed model-based analysis and the code generation engine, which generates the code for the concurrent architecture.

Separation of concerns. The RCM promotes strict separation between functional modeling and architectural/-concurrent modeling. The functional specification of the system shall be intentionally void of any concurrent and time-related semantics. This form of separation of concerns is sought especially to facilitate reuse of the functional specification across distinct concurrent and distribution architectures.

Types and instances. APLC (and VMLC) are type based and support multiple instantiation. Thanks to this feature, RCM places a clear step ahead of the object-centric design space typical of other methodologies like for example HRT-HOOD. The dichotomy of types and instances has to be properly expressed in the design space and recognisable in the source code. Even if APLC and VMLC are considered as types, they do not feature a complete object-oriented nature: inheritance and methods overriding are intentionally removed from the Interface View. Conversely, full object-orientation is supported in the Functional View for the definition of the sequential behavior of the system.

Traceability. All the entities present in the PSM should be present in the automatically generated source code so as to allow complete traceability. This implies that APLC types, APLC instancies, VMLC types and VMLC instancies should *all* be mapped in the source code.

Chapter 2

Mapping of VM-level Containers

In this chapter we focus on the realisation of VM-level Containers according to the current code generation strategy. First, we examine the general structure of an OPCS, which encapsulates the sequential behaviour of system components. Subsequently, we explore a set of code archetypes, which are fragments of code used to factorize the common behaviours that can be typically encountered in a real-time system (a task with cyclic activation pattern, a sporadic task, etc..). Finally, we examine how archetypes are assembled to form the structure of a VMLC, as introduced in chapter 1. In order to simplify the presentation and the discussion, the code archetypes shown in this chapter do not make any provisions for the monitoring of execution time nor for handling violation events. Consideration of those features however forms integral part of the RCM code generation logic.

2.1 Structure of the OPCS

Let us commence by examining how an OPCS is mapped to code in the current strategy of code generation. In particular we first review the mapping of Interfaces specified in the Functional View.

Listing 2.1: Interface mapping

```
1  package <Interface_Type>s is
2     -- for the sake of readability assume <Interface_Type> is ''IProducer''
3     type IProducer is interface;
4     type IProducer_Ref is access all IProducer'Class;
5     type IProducer_Static_Ref is access all IProducer;
6
7     type IProducer_Arr is
8        array(Standard.Integer range <>) of IProducer_Ref;
9     type IProducer_Arr_Ref is access IProducer_Arr;
10
11    --  for each interface operation
12    procedure <Operation_Name> (This : in out IProducer;
13                                <Param1_Name> : in <Param1_Type>;
14                                <ParamN_Name> : in <ParamN_Type>) is abstract;
15 private
```

```
16 end <Interface_Type>s;
```

The mapping is straightforward. For each Interface, a dedicated package (with the 's' character appended to the Interface name) is created. In this package, we define the interface, as well as access types to it. Each operation in the interface definition is also generated as shown on line 10 of listing 2.1.

Now we examine the mapping of classes defined in the Functional View, which can be used as the OPCS of VMLC.

Listing 2.2: Generic structure of the OPCS (spec)

```
1  package <OPCS_Type>s is
2     -- for the sake of readability assume <OPCS_Type> is ''Producer''
3     type Producer is new Controlled and <OPCS_SuperType>s.<OPCS_SuperType>
4        with private;
5     type Producer_Ref is access all Producer'Class;
6     type Producer_Static_Ref is access all Producer;
7
8     type Producer_Arr is array(Standard.Integer range <>) of Producer_Ref;
9     type Producer_Arr_Ref is access Producer_Arr;
10
11    overriding
12    procedure Initialize (This : in out Producer);
13
14    -- for each operation in PI
15    procedure <Operation_Name> (This : in out Producer;
16                                <Param1_Name> : in <Param1_Type>;
17                                <ParamN_Name> : in <ParamN_Type>);
18
19    -- for each attribute
20    procedure Set_<Attribute_Name> (This : in out Producers.Producer;
21                                    <Var_Name> : <OPCS_Type1>s.Producer);
22 private
23    type Producer is new Controlled and
24       <OPCS_SuperType>s.<OPCS_SuperType> with record
25       --  for each primitive attribute
26       <Attribute_Name> : <Primitive_Type>s.<Primitive_Type>;
27       -- for each non-primitive attribute
28       <Attribute_Name1> : <OPCS_Type1>s.<OPCS_Type1>_Ref;
29    end record;
30 end <OPCS_Type>s;
```

Similarly to what we saw for Interface specification, the OPCS declaration is placed in a package named after the name of the OPCS with a trailing 's'. The generated code includes the declaration of the OCPS as a limited, controlled record type and the access types that point to it. (The limitedness and controlled nature of the record type ensures that the objects of that type can only be manipulated by specific operations, which adds to the overall integrity of the code.) All the member attributes present in the class definition are declared in the record type. A setter procedure is defined to initialize those attributes.

Listing 2.3: Generic structure of the OPCS (implementation)

```
1  package body <OPCS_Type_>s is
2     -- for the sake of readability assume <OPCS_Type> is ''Producer''
3     procedure Initialize (This : in out Producer) is
4     begin
5        --  user-code here;
6     end Initialize;
7
8     -- for each PI
9     procedure <Operation_Name> (This : in out Producer;
10                                <Param1_Name> : in <Param1_Type>;
11                                <ParamN_Name> : in <ParamN_Type>) is
12    --+ <List of component RI>
13    --+ <List of accessed members>
14    begin
15       -- User-code here --
16    end <Operation_Name>;
17
18    procedure Set_<Attribute_Name> (This : in out Producers.Producer;
19                                    <Var_Name> : <OPCS_Type1>s.<OPCS_Type1>_Ref) is
20    begin
21       --  setters should be invoked only once at system initialization
22       if This.<Attribute_Name> = null then
23          This.<Attribute_Name> := <Var_Name>;
24       end if;
25    end Set_<Attribute_Name>;
26 end <OPCS_Type>s;
```

The generation engine places appropriate hooks where action semantics can be inserted (via manual coding or via interfacing with code generated by foreign tools): the *functional* contract of each operation (accessed members, invoked required interfaces) is generated as source documentation.

2.2 Code archetypes

Code archetypes are used to factorize common behaviour usually found in a real-time system in a shared library of patterns [BV07]. The use of code archetypes improves the compactness of the generated code.

The current RCM code generation strategy uses a set of archetypes: cyclic thread, cyclic thread with modifiers, sporadic thread, and a common functional behaviour for the OBCS.

2.2.1 Cyclic thread

Listing 2.4: Skeleton of a Cyclic Thread

```
1  task type Thread_T(Thread_Priority : Any_Priority;
2                     Interval : Integer) is
3     pragma Priority(Thread_Priority);
4  end Thread_T;
5
6  task body Thread_T is
7     Next_Time : Time := System_Start_Time + Task_Activation_Delay;
8  begin
9     loop
10       delay until Next_Time;
11       -- Perform the operation of the OPCS
12       Next_Time := Next_Time + Milliseconds (Interval);
13    end loop;
14 end Thread_T;
```

The listing shows the code archetype for a cyclic *thread*. After elaboration, the thread is immediately put into suspension until a system-wide start time, which represents the common start time of all threads with 0 phase. Support for thread-specific offsets can easily be incorporated by including a further task attribute valued to a user-level parameter specified in the Interface View. Upon release after suspension, the thread performs the operation specified in its OPCS, then computes the *absolute time* of its next activation, and finally repeat the cycle.

To build a shared library of archetypes that factorize common execution behaviors within ASSERT systems, we need to allow the thread operation to be instantiated on a per-VMLC basis. To this end, we use the *generic* construct of the Ada language. The archetype thus becomes the following:

Listing 2.5: Cyclic Thread

```
1  generic
2     with procedure Cyclic_Operation;
3  package Simple_Cyclic_Task is
4     task type Thread_T(Thread_Priority : Any_Priority;
5                        Interval : Integer) is
6        pragma Priority(Thread_Priority);
7     end Thread_T;
8  end Simple_Cyclic_Task;
9
10 package body Simple_Cyclic_Task is
11    task body Thread_T is
12       Next_Time : Time := System_Start_Time + Task_Activation_Delay;
13    begin
14       loop
15          delay until Next_Time;
16          -- Perform the operation of the OPCS
17          Cyclic_Operation; -- obviously parameterless for a cyclic thread!
```

```
18          Next_Time := Next_Time + Milliseconds (Interval);
19       end loop;
20    end Thread_T;
21 end Simple_Cyclic_Task;
```

2.2.2 Cyclic thread with modifiers

Listing 2.6: Cyclic Thread with Modifiers

```
1  generic
2     with procedure Cyclic_Operation;
3     with procedure Get_Request(Req : out Request_Descriptor_T);
4  package Cyclic_Task_ATC is
5     task type Thread_T(Thread_Priority : Any_Priority;
6                        Interval : Positive) is
7        pragma Priority(Thread_Priority);
8     end Thread_T;
9  end Cyclic_Task_ATC;
10
11 package body Cyclic_Task_ATC is
12    task body Thread_T is
13       Req_Desc : Request_Descriptor_T;
14       Next_Time : Time := System_Start_Time + Task_Activation_Delay;
15    begin
16       loop
17          delay until Next_Time;
18          Get_Request (Req_Desc);
19          case Req_Desc.Request is
20             when NO_REQ =>
21                --nominal operation
22                Cyclic_Operation;
23             when ATC_REQ =>
24                -- modifier operation
25                My_OPCS(Req_Desc.Params.all); -- may take parameters!
26             when others =>
27                -- error handling
28          end case;
29          Next_Time := Next_Time + Milliseconds(Interval);
30       end loop;
31    end Thread_T;
32 end Cyclic_Task_ATC;
```

The listing shows the code archetype for a cyclic thread with modifiers. As in the archetype of section 2.2.1, the thread executes with a fixed *period* the cyclic operation denotes as the *nominal operation*). The nominal operation is passed as an instantiation parameter to the encompassing *generic* unit.

Additionally, this thread is able to receive requests for the execution of alternative operations, termed *modifier operations* or simply *modifiers*. While the cyclic nominal operation cannot have any parameter since it is not invoked by software, modifiers instead can (cf. listing 2.6, line 23), since they indeed are invoked by software. The invocation parameters are not directly handled in the code of the thread, but instead are stored in a request descriptor.

Each time the thread is put in the running state after resuming from suspension, it inspects a request queue held in its OBCS, searching for possible *asynchronous* requests of execution posted in the meanwhile by some callers. If no such requests are found, the thread executes its nominal operation. Otherwise, if there are pending requests, the OBCS returns the first request from the queue and the thread executes the appropriate *modifier operation* on it. In this manner, the thread *skips* for one activation the execution of the nominal operation. In this section we intentionally omit the scrutiny of Request Descriptors. Suffice it to say for now that they embed the parameters needed to perform the requested operation.

2.2.3 Sporadic thread

Listing 2.7: Sporadic Thread

```
1  generic
2     with procedure Get_Request(Req : out Request_Descriptor_T);
3  package Sporadic_Task is
4     task type Thread_T(Thread_Priority : Any_Priority;
5                        Interval : Integer) is
6        pragma Priority (Thread_Priority);
7     end Thread_T;
8  end Sporadic_Task;
9
10 package body Sporadic_Task is
11    task body Thread_T is
12       Req_Desc : Request_Descriptor_T;
13       Next_Time : Time := System_Start_Time + Task_Activation_Delay;
14       Release : Time;
15       id : aliased Task_Id := Current_Task;
16    begin
17       loop
18          delay until Next_Time;
19          Get_Request(Req_Desc, Release);
20          case Req_Desc.Request is
21            when START_REQ | ATC_REQ =>
22               -- nominal or modifier operation
23               My_OPCS (Req_Desc.Params.all);
24            when NO_REQ =>
25               -- intentional idling
26               null;
27            when others =>
28               -- error handling
29          end case;
30          Next_Time := Release + Milliseconds (Interval);
```

```
31        end loop;
32     end Thread_T;
33  end Sporadic_Task;
```

The listing shows the code archetype for a thread with sporadic behaviour. The archetype is very similar to the cyclic thread with modifiers.

As in the previous case the thread is put in the suspended state until the time of first system-wide activation. Then the thread calls the single entry of its OBCS (*Get_Request*) to probe for execution requests. When at least one request is pending in the queue held in the OBCS, the guard to the entry is open and the entry call returns with the first request descriptor from the queue. Otherwise the calling thread is blocked until an execution request is posted. On resuming execution, the thread performs the required operation and then computes the next earliest time of activation and suspends until then. It is a distinct requirement on sporadic threads that subsequent jobs of theirs must be spaced by at least some minimum interarrival time (MIAT). The code generation strategy enforces that requirement by having the thread suspend between successive executions until an absolute time no earlier than the return time from *Get_Request* plus the MIAT value stipulated in the Interface View.

At line 19 of listing 2.7, parameter *Release* is passed to procedure *Get_Request* and it is updated during the execution of the delegation chain of *Get_Request* so as to store the actual release time of the activation of the thread.

The reader can appreciate the difference between the cyclic and sporadic behavior of the corresponding two archetypes: while the former is released and executes at each time instant that is a multiple of its period, the latter executes only after the stipulated MIAT has arrived and when there is at least a pending execution request in the OBCS.

2.2.4 Functional behaviour of the OBCS

The code generation strategy defines two distinct types of *synchronization behaviour* for the OBCS: the one that is used for the OBCS of Cyclic VMLC (*Cyclic_OBCS*) and the latter, for the Sporadic VMLC (*Sporadic_OBCS*).

The distinct behaviours are defined using two Ada types that inherit from a common abstract type (*OBCS_T*).

The following listing shows the specification and implementation of the Sporadic OBCS.

Listing 2.8: Specification of OBCS and Sporadic OBCS

```
1  type Request_T is (NO_REQ, ATC_REQ, START_REQ);
2
3  type Param_Type is abstract tagged record
4     In_Use : Boolean := False;
5  end record;
6  type Param_Type_Ref is access all Param_Type'Class;
7
8  type Request_Descriptor_T is record
9     Request : Request_T;
10    Params : Param_Type_Ref;
11 end record;
12
13 -- Abstract interface of OBCS
14
15 type OBCS_T is abstract new Controlled with null record;
```

```
16 type OBCS_T_Ref is access all OBCS_T'Class;
17
18 procedure Put (Self : in out OBCS_T;
19                Req : Request_T;
20                P : Param_Type_Ref) is abstract;
21
22 procedure Get (Self : in out OBCS_T;
23                R : out Request_Descriptor_T) is abstract;
24
25 -- Concrete type of sporadic OBCS
26
27 Sporadic_OBCS (Size : Integer) is new OBCS_T with record
28    START_Param_Buffer : Param_Arr (1..Size);
29    START_Insert_Index : Integer;
30    START_Extract_Index : Integer;
31    START_Pending : Integer;
32    ATC_Param_Buffer : Param_Arr (1..Size);
33    ATC_Insert_Index : Integer;
34    ATC_Extract_Index : Integer;
35    ATC_Pending : Integer;
36    Pending : Integer;
37 end record;
38
39 overriding
40 procedure Initialize (Self : in out Sporadic_OBCS);
41
42 overriding
43 procedure Put (Self : in out Sporadic_OBCS;
44                Req : Request_T;
45                P : Param_Type_Ref);
46
47 overriding
48 procedure Get (Self : in out Sporadic_OBCS;
49                R : out Request_Descriptor_T);
```

Listing 2.9: Implementation of procedure Put and Get in the sporadic OBCS

```
1  -- Interface operations of sporadic OBCS
2  procedure Put (Self : in out Sporadic_OBCS;
3                 Req : Request_T;
4                 P : Param_Type_Ref) is
5  begin
6     case Req is
7        when START_REQ =>
8           Self.START_Param_Buffer (Self.START_Insert_Index) := P;
9           Self.START_Insert_Index := Self.START_Insert_Index + 1;
10          if Self.START_Insert_Index > Self.START_Param_Buffer'Last then
11             Self.START_Insert_Index := Self.START_Param_Buffer'First;
```

```
12          end if;
13          -- increment the counter of pending requests without overflowing
14          if Self.START_Pending < Self.START_Param_Buffer'Last then
15             Self.START_Pending := Self.START_Pending + 1;
16          end if;
17       when ATC_REQ =>
18          Self.ATC_Param_Buffer(Self.ATC_Insert_Index) := P;
19          Self.ATC_Insert_Index := Self.ATC_Insert_Index + 1;
20          if Self.ATC_Insert_Index > Self.ATC_Param_Buffer'Last then
21             Self.ATC_Insert_Index := Self.ATC_Param_Buffer'First;
22          end if;
23          -- increment the counter of pending requests without overflowing
24          if Self.ATC_Pending < Self.ATC_Param_Buffer'Last then
25             Self.ATC_Pending := Self.ATC_Pending + 1;
26          end if;
27       when others =>
28          -- error handling
29    end case;
30    Self.Pending := Self.START_Pending + Self.ATC_Pending;
31 end Put;
32
33 procedure Get (Self : in out Sporadic_OBCS;
34                R : out Request_Descriptor_T) is
35 begin
36    if Self.ATC_Pending > 0 then
37       R := (ATC_REQ,
38             Self.ATC_Param_Buffer(Self.ATC_Extract_Index));
39       Self.ATC_Extract_Index := Self.ATC_Extract_Index + 1;
40       if Self.ATC_Extract_Index > Self.ATC_Param_Buffer'Last then
41          Self.ATC_Extract_Index := Self.ATC_Param_Buffer'First;
42       end if;
43       Self.ATC_Pending := Self.ATC_Pending - 1;
44    else
45       if Self.START_Pending > 0 then
46          R := (START_REQ,
47                Self.START_Param_Buffer(Self.START_Extract_Index));
48          Self.START_Extract_Index := Self.START_Extract_Index + 1;
49          if Self.START_Extract_Index > Self.START_Param_Buffer'Last then
50             Self.START_Extract_Index := Self.START_Param_Buffer'First;
51          end if;
52          Self.START_Pending := Self.START_Pending - 1;
53       end if;
54    end if;
55    -- the parameter is in use
56    R.Params.In_Use := True;
57    Self.Pending := Self.START_Pending + Self.ATC_Pending;
58 end Get;
```

The Sporadic OBCS embeds two circular buffers: one buffer contains all requests of execution of nominal operations (*START_Param_Buffer*); the other contains all requests of execution of modifier operations (*ATC_Param_Buffer*).

When posting an execution request in the OBCS, procedure *Put* discriminates on the request type to locate the correct destination buffer. When fetching a request from the OBCS, the ATC buffer is inspected *first*. If ATC requests are pending, the first of them is extracted and inserted in a request descriptor which specifies the type of the request. Otherwise the first START pending request is extracted. The default queuing policy is FIFO, which is known to be fair and statically analyzable. Other policies might be contemplated, but none other would be fair.

The code of the Cyclic OBCS is not reported here, but it is quite simple to evoke since it is a simplification of the Sporadic OBCS. It consists in a single circular buffer to store ATC requests. When procedure *Get* is called, down in the delegation chain of the *Get_Request* call made by a thread, it inspects the ATC buffer. If there are pending requests, the first of them is fetched into a request descriptor whose type specifies that an *ATC_REQ* is included. Otherwise a request descriptor with *NO_REQ* request type is generated. The descriptor will determine the execution of the nominal cyclic operation for the current task activation (cf. 2.2.2).

Record member *Self.Pending*, updated at lines 31 and 58 of listing 2.9, is used in the guard expression attached to the entry of the OBCS archetype.

2.3 Complete structure of VMLC

In this section we examine the complete structure of each type of VMLC to review how code archetypes are combined together to form an aggregate of run-time entities which comply with the RCM and exhibit the intended concurrent semantics.

2.3.1 Passive VMLC

A passive VMLC is the simplest type of VMLC. It does not exhibit any concurrent semantics and thus does not require any real-time attribute.

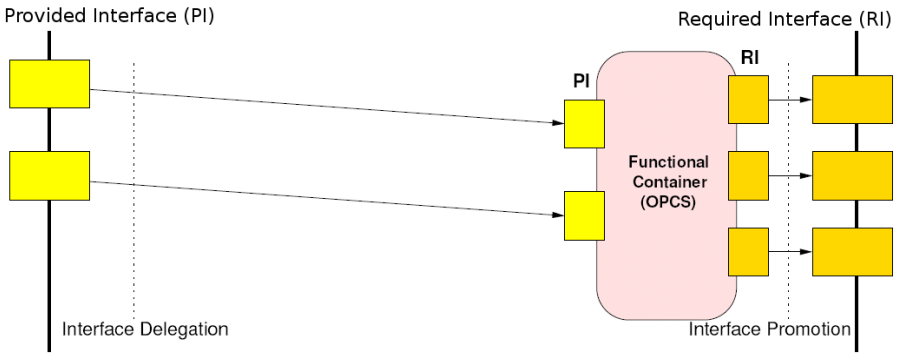

Figure 2.1: Passive VMLC

The structure of a passive VMLC is depicted in fig. 2.1. The VMLC exposes a PI and a RI. The services of the PI simply perform an indirection to the services offered by the OPCS of the VMLC.

In the current code generation strategy the mapping of a passive VMLC to code is thus very straightforward. Figure 2.2 is a graphical representation of the implemented mapping.

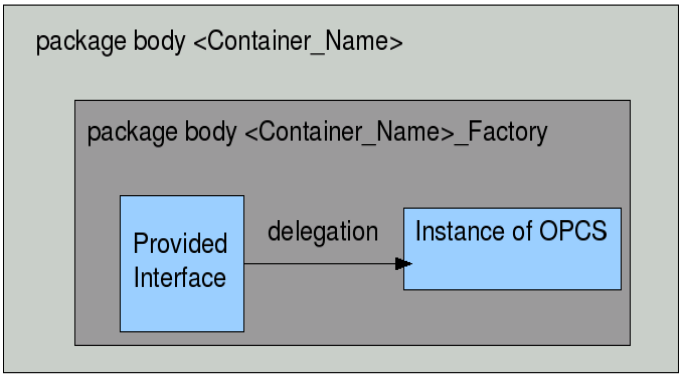

Figure 2.2: Architectural mapping of passive VMLC

Listing 2.10: Specification of passive VMLC

```
1  package <VMLC_Name> is
2    generic
3       My_ID : Deployment.Entity_Type;
4       OPCS_Instance : <OPCS_Type>_Static_Ref;
5       -- <OPCS_Type>_Ref is access type to the class
6       --+ containing the OPCS of the container
7     package <VMLC_Name>_Factory is
8       -- for each operation in the PI
9       procedure/function <OP_Name> (<Parameter_Signature>);
10    private
11      -- ...
12    end <VMLC_Name>_Factory;
13 end <VMLC_Name>;
```

The use of the *generic* construct of the Ada language [ISO05] is used to make the structure of the VMLC parametric on the type of the reference to the specific OPCS to be embedded in the VMLC.

Listing 2.11: Implementation of a passive VMLC

```
1  package body <VMLC_Name> is
2     package body <VMLC_Name>_Factory is
3        -- for each operation in the PI
4        procedure/function <OP_Name> (<Parameter_Signature>) is
5        begin
6           -- simple call indirection
7           OPCS_Instance.<OP_Name>(<Parameter_Values>);
8        end <OP_Name>
9     end <VMLC_Name>_Factory;
10 end <VMLC_Name>;
```

The unit body includes the implementation of all the procedures published in the relevant PI, realized as a simple call indirection to the concrete OPCS instance, which in fact is passed as the actual parameter to the generic instantiation.

This mapping schema is retained for all other types of VMLC. The rationale to this choice is to adopt a recurrent, recognizable pattern through all the code generation strategy, thus making the overall approach more coherent and recognizable. However, this choice does not preclude future optimizations in case the overhead of the indirection was deemed undesirable.

2.3.2 Protected VMLC

A protected VMLC extends the Passive VMLC by interposing a synchronization agent in the form of an OBCS, between the actual PI interface and the OPCS where the sequential code resides. The mapping of protected VMLC to code thus extends the generation pattern provided for passive VMLC.

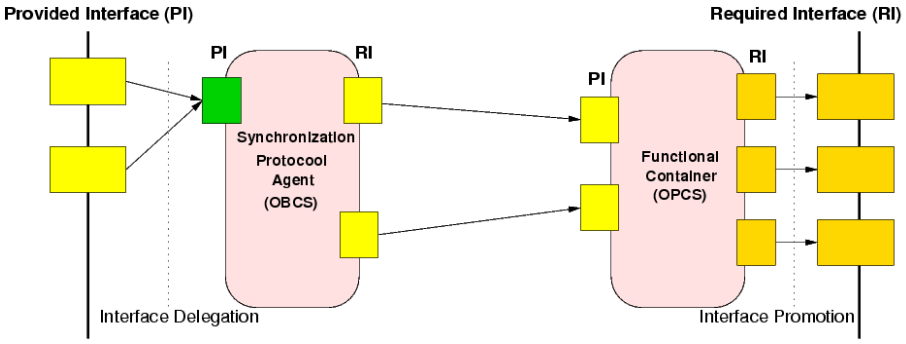

Figure 2.3: Protected VMLC

Figure 2.4 is a graphical representation of the implemented mapping.

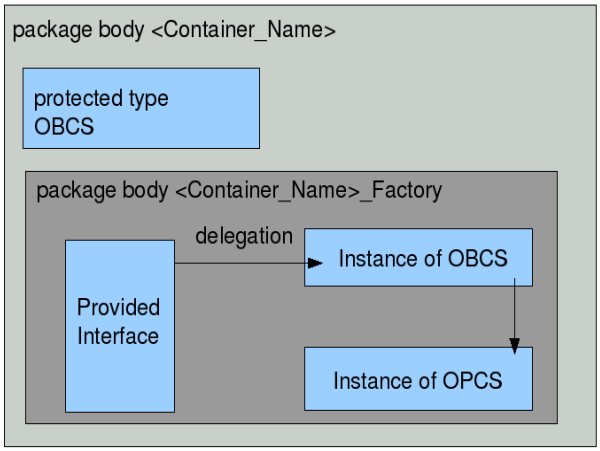

Figure 2.4: Architectural mapping of protected VMLC

Listing 2.12: Specification of Protected VMLC

```
1  package <VMLC_Name> is
2     generic
3        My_ID : Deployment.Entity_Type;
4        Ceiling : Priority;
5        OPCS_Instance : <OPCS_Type>_Static_Ref;
6     package <VMLC_Name>_Factory is
7        -- for each operation in the PI
8        procedure/function <OP_Name> (<Parameter_Signature>);
9     private
10       -- ...
11    end <VMLC_Name>_Factory;
12 private
13   protected type OBCS (Ceiling : Priority;
14                        O : <OPCS_Type>_Ref) is
15      pragma Priority (Ceiling);
16      -- for each operation in PI
17      procedure/function <OP_Name> (<Parameter_Signature>);
18   private
19      OPCS : <OPCS_Type>_Static_Ref := O;
20   end OBCS;
21 end <VMLC_Name>;
```

Listing 2.13: Implementation of Protected VMLC

```
1  package body <VMLC_Name> is
2     protected body OBCS is
3        -- for each operation in the PI
4        procedure/function <OP_Name> (<Parameter_Signature>) is
5        begin
6           -- call indirection
7           OPCS.<Op_Name> (<Parameter_Values>);
8        end <OP_Name>;
9     end OBCS;
10
11    package body <VMLC_Name>_Factory is
12       -- the OBCS instance
13       Protocol : aliased OBCS (Ceiling, OPCS_Instance);
14       -- for each operation in PI
15       procedure/function <OP_Name> (<Parameter_Signature>) is
16       begin
17          -- simple call indirection to the OBCS
18          Protocol.<OP_Name> (<Parameter_Values>);
19       end <OP_Name>;
20    end <VMLC_Name>_Factory;
21 end <VMLC_Name>;
```

The unit body includes the definition of a protected type to realize the required non-functional semantics. The functional behaviour of the protected body is simply a call indirection to its OPCS (line 7 of listing 2.13). Note that the protected type is defined inside the package but outside the generic body: in this manner it is possible to use the same type in multiple instances of the same container, without recurring to compile-time object code duplication, typical of the compilation model of Ada generics. The generic body contains an instance of the protected type (the instantiation of protected objects at library level is an Ada Ravenscar constraint), and a set of call indirections from the PI of the VMLC to the protected object. Each call to the PI of the protected VMLC is thus subject to *Ceiling Locking*, which governs synchronization in protected objects under the Ravenscar Profile.

2.3.3 Threaded VMLC

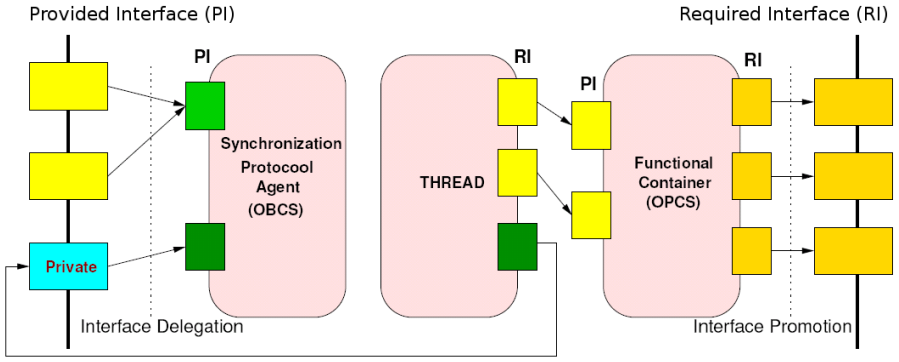

Figure 2.5: Threaded VMLC

Figure 2.5 depicts the structure of a threaded VMLC. A threaded VMLC consists in a queue of preallocated request descriptors that are used up to store the incoming execution requests. The preallocation of request descriptors complies with the *static exhistence* constraints of the RCM. The thread of the VMLC extracts a request from the queue and executes the corresponding functional code. The request queue belongs to the OBCS of the VMLC, which provides for protection against concurrent access (since the queue is accessed by the thread and all the callers of the PI of the VMLC). The general structure of the mapping is depicted in figure 2.6. In the following we comment pn the code of a Sporadic VMLC and subsequently we outline the differences between Sporadic VMLC and Cyclic VMLC.

Sporadic VMLC

Listing 2.14: Specification of Sporadic VMLC

```
1 package <VMLC_Name> is
2    generic
3       My_ID : Deployment.Entity_Type;
4       Thread_Priority : Priority;
5       Ceiling : Any_Priority;
6       MIAT : Standard.Integer;
```

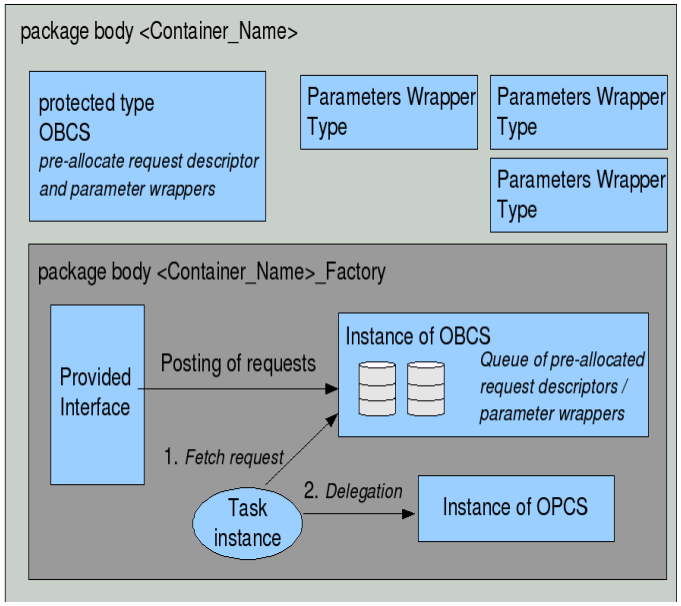

Figure 2.6: Architectural mapping of threaded VMLC

```
7       OPCS_Instance : <OPCS_Type>_Ref;
8    package <VMLC_Name>_Factory is
9       -- for each modifier operation in the PI
10      procedure/function <OP_Name> (<Parameter_Signature>);
11   private
12      -- ...
13   end <VMLC_Name>_Factory;
14 private
15   Param_Queue_Size : constant Standard.Integer := <Queue_Size>;
16   OBCS_Queue_Size : constant Standard.Integer := Param_Queue_Size * 1;
17
18   -- for each operation in the PI
19   type <OP_Name>_Param_T is new Param_Type with
20      record
21         OPCS_Instance : <OPCS_Type>_Static_Ref;
22         -- for each primitive parameter
23         <Param1_Name> : <Param1_Type>;
24         -- for each non-primitive parameter
25         <Param2_Name> : <Param2_Type>_Ref;
26      end record;
27
28   type <OP_Name>_Param_T_Ref is access all <OP_Name>_Param_T;
29   type <OP_Name>_Param_Arr is array(Standard.Integer range <>) of
30      aliased <OP_Name>_Param_T;
31
```

```
32    overriding
33    procedure My_OPCS(Self : in out <OP_Name>_Param_T);
34    -- end for each
35
36    protected type OBCS (Ceiling : Any_Priority;
37                         <OP_Name1>_Params_Arr_Ref_P : Param_Arr_Ref;
38                         <OP_NameN>_Params_Arr_Ref_P : Param_Arr_Ref) is
39       pragma Priority (Ceiling);
40       entry Get_Request (Req : out Request_Descriptor_T,
41                          Release : out Time);
42
43       -- for each modifier operation in the PI
44       procedure <OP_Name> (<Parameter_Signature>);
45    private
46       OBCS_Queue : Sporadic_OBCS (OBCS_Queue_Size);
47       Pending : Standard.Boolean := False;
48       -- for each operation in PI
49       <OP_Name>_Params : Param_Buffer_T (Param_Queue_Size) :=
50          (Size => Param_Queue_Size, Index => <I>,
51           Buffer => <OP_Name>_Params_Arr_Ref_P.all);
52    end OBCS;
53 end <VMLC_Name>;
```

The OBCS of the VMLC is the usual protected type declared in the package body, yet outside the enclosed generic unit, and instantiated inside the latter. The OBCS exposes a method for each PI of the VMLC (cf. figure 2.5). The generic instantiation parameters are the thread priority, the ceiling of the istance of OBCS, the MIAT of the thread and the OPCS instance.

Listing 2.15: Implementation of Sporadic VMLC

```
1  package body <VMLC_Name> is
2     -- for each operation in the PI
3     procedure My_OPCS(Self : in out <OP_Name>_Param_T) is
4     begin
5        Self.OPCS_Instance.<OP_Name>(<Parameters_Signature>);
6        Self.In_Use := False;
7     end My_OPCS;
8
9     protected body OBCS is
10       procedure Update_Barrier is
11       begin
12          -- Additional conditions omitted
13          Pending := Obcs_Queue.Pending > 0;
14       end Update_Barrier;
15
16       entry Get_Request (Req : out Request_Descriptor_T,
17                          Release : out Time) when Pending is
18       begin
19          Release := Clock;
```

```
20          Get (Obcs_Queue, Req);
21          Update_Barrier;
22       end Get_Request;
23
24       -- for each operation in the PI
25       procedure <Op_Name> (<Parameter1>) is
26       begin
27          if <Op_Name>_Params.Buffer (<Op_Name>_Params.Index).In_Use then
28             Increase_Index (<Op_Name>_Params);
29          end if;
30          <Op_Name>_Param_T_Ref (<Op_Name>_Params.Buffer
31                                 (<Op_Name>_Params.Index)).<Parameter1>
32                                 := <Parameter1>;
33          Put (Obcs_Queue,
34               <OP_TYPE>,
35               <Op_Name>_Params.Buffer (<Op_Name>_Params.Index));
36          Increase_Index (<Op_Name>_Params);
37          Update_Barrier;
38       end <Op_Name>;
39    end OBCS;
40
41    package body <VMLC_Name>_Factory is
42       <Op_Name1>_Par_Arr : <OP_Name1>_Param_Arr (1..Param_Queue_Size) :=
43         (others => (false, OPCS_Instance));
44       <Op_Name1>_Ref_Par_Arr : aliased Param_Arr :=
45         (<OP_Name1>_Par_Arr (1)'access,
46          <OP_Name1>_Par_Arr (2)'access,
47          <OP_Name1>_Par_Arr (3)'access);
48       <Op_NameN>_Par_Arr : <OP_NameN>_Param_Arr (1..Param_Queue_Size) :=
49         (others => (false, OPCS_Instance));
50       <Op_NameN>_Ref_Par_Arr : aliased Param_Arr :=
51         (<OP_NameN>_Par_Arr (1)'access,
52          <OP_NameN>_Par_Arr (2)'access,
53          <OP_NameN>_Par_Arr (3)'access);
54       Nominal_Params : aliased Nominal_Param_Type := <Nominal_Params>;
55
56       Protocol : aliased OBCS (Ceiling,
57                                <OP_Name1>_Ref_Par_Arr'access,
58                                <OP_NameN>_Ref_Par_Arr'access,
59                                Nominal_Params'Access);
60
61       procedure Getter (Req : out Request_Descriptor_T,
62                         Release : out Time) is
63       begin
64          Protocol.Get_Request (Req, Release);
65       end Getter;
66
67       package My_Sporadic_Task is new Sporadic_Task (Getter);
68
69       Thread : My_Sporadic_Task.Thread_T (Thread_Priority, MIAT);
```

```
70
71          -- for each operation in PI
72          procedure <OP_Name> (<Parameter_Signature>) is
73          begin
74              Protocol.<OP_Name> (<Parameter_Signature>);
75          end <OP_Name>;
76
77      end <VMLC_Name>_Factory;
78  end <VMLC_Name>;
```

The VMLC uses the sporadic thread archetype discussed in section 2.2.3). The thread is instantiated inside the generic package, using the instantiation parameters of the generic. The thread accesses the queue of the OBCS through the *Getter* operation specified at lines 61-65 of listing 2.15)

The code generation strategy uses the *reification* of the execution requests directed to *deferred* PI operations. The invocation (type and actual parameters) is recorded in a language-level structure and stored in the OBCS. The client of the sporadic VMLC invokes PI operations, which are encoded as at lines 72-75 of listing 2.15). Each such operations is a simple redirection to an operation with the same name and parameters of the OBCS (cf. lines 25-38 of listing 2.15). Each parameter of the invocation is copied into a parameter buffer (cf. lines 30-32 of listing 2.15), which is subsequently posted to the OBCS (lines 33-35) with tag *OP_TYPE* set to either *START_REQ* or *ATC_REQ* according to whether the requested operation is the nominal sporadic operation or else a modifier. The code for procedure *Put* of the Sporadic OBCS is shown in listing 2.9.

The OBCS is thus the repository of the history of the pending requests and it is thus a crucial element of the implementation of the asynchronous communication paradigm prescribed by the RCM.

Entry *Get_Request* at lines 16-22 of listing 2.15). The barrier to the entry is composed of the *single* Boolean variable *Pending*, again in compliance with the restrictions of the RCM. Procedure *Update Barrier* is provided to allow multiple-variable conditions to be composed into a legal RCM guard.

Still at line 19 of entry *Get_Reuqest*, return parameter *Release* is set, so that it can be used in the sporadic thread archetype code to enforce the MIAT (cf. listing 2.7).

Cyclic VM-level Container

The mapping of a cyclic VMLC is very similar to that of the sporadic VMLC. The cyclic VMLC is composed of the same entities (OBCS, thread, OPCS). The sole differences to its sibling are as follows:

- The instantiation parameters of the generic unit include the *Period* instead of the MIAT.
- *OBCS_Queue* is an instance of *Cyclic_OBCS*, thus following the functional behavior specified in section 2.2.4.
- An additional parameterless operation *Cyclic_Operation* is defined in the package (which redirects to the correct operation of *OPCS_Instance* in analogy with procedure *My_OPCS* of listing 2.15 at lines 3-7. When the thread executes procedure *Get_Request* and is redirected to the *Get_Request* entry, it executes the next request for a pending modifier operation (corresponding to an *ATC_REQ*), if any (cf. listing 2.6 at line 23). If the queue of pending requests is empty, the thread executes *Cyclic_Operation* (cf. listing 2.6 at line 20). In contrast to the Sporadic VMLC therefore, for Cyclic VMLC the number of pending requests does not constitute a condition that influences the value of the barrier of the OBCS entry.

Chapter 3

Mapping of AP-level Containers

In this section we discuss the mapping to code of Application-level Containers (APLC). Though APLC are non-executable entities, their mapping is important, since it is instrumental to ensuring complete traceability between the user model and the source code. APLC types and instances must therefore be present, as non-executable architectural artefacts, in the source code. In order to achieve this goal, we must be prepared to pay some (marginal) performance penalty in the executable. To begin our discussion, let us examine a reduced version of the specification of an APLC.

3.1 APLC types

Listing 3.1: Generic specification of an APLC (with omitted fragments)

```
1  package <APLC_Name> is
2     generic
3        -- list of instantiation parameters omitted
4     package <APLC_Name>_Factory is
5        -- code omitted
6     end <APLC_Name>_Factory
7  private
8     -- for each functional state in the APLC
9     type <State_Name>_T is new <OPCS_Type> with record
10       -- for each PI in the <OPCS_Type> superclass
11       <Operation_Name>_0_Ref : access procedure/function (<Operation_Parameters>);
12    end record;
13    -- for each PI operation in functional state <OPCS_Type>
14    overriding
15    procedure/function <Operation_Name> (This : in out <State_Name>_T;
16                                         <Operation_Parameters>);
17    end <APLC_Name>;
```

Similarly to VMLC, APLC are mapped using the *generic* construct of Ada. We defer the discussion of the instantiation parameters themselves. For now, suffice it to note that they are used to instantiate the inner package (the Factory). The instantiation process is discussed in section 3.2.

Listing 3.1 omits some fragments of code to better highlight the definition of functional states. APLC embeds one or more functional states, which are comprised of the cohesive set of static variables on which the PI of the APLC operate (whether individually or as a group thereof). Functional states are typed to *non-abstract classes* defined in the Functional View (cf. section 2.1). In order to implement functional states, the APLC defines a new type (lines 9-12) of each of them, which extends the base class specified in the Functional View: the new type contains an access procedure/function for each operation of the original class; the access procedure has the same signature as the original operation on the corresponding functional state (and as published in the relevant PI of the APLC). As shown at lines 15-16 of listing 3.1, the PI of the APLC publishes a procedure or function for each public service that operates on the functional states embedded in the APLC.

Listing 3.2: Generic implementation of an APLC (with omitted code)

```
1  package body <APLC_Name> is
2    -- for each PI operation
3    procedure/function <Operation_Name>(This : in out <State_Name>_T;
4                                        <Operation_Parameters>) is
5    begin
6       This.<Operation_Name>_0_Ref.all (<Operation_Parameters>);
7    end <Operation_Name>;
8
9    package body <APLC_Name>_Factory is
10     -- for each functional state
11     <State_Name>_Instance : aliased <State_Name>_T;
12     -- for each VMLC involved in the implementation of the APLC
13     My_<State_Name>_<Operation_Name>_<VMLC_kind> is new
14        <State_Name>_<Operation_Name>_<VMLC_kind>.
15           <VM_Container>_Factory (<Generic_Instantiation_Parameters>);
16     -- code fragments omitted
17   begin
18     -- for each PI operation
19     -- <State_Name>_Instance.<Operation_Name>_0_Ref :=
20     --    <VMLC_Instance>.<Operation_Name>'access;
21   end <APLC_Name>_Factory;
22 end <APLC_Name>;
```

As shown at lines 13-15 of listing 3.2, all of the VMLC that are needed to provide the required concurrency semantics for the PI operations of the APLC are instantiated in the body of the APLC. Subsequently, at lines 19-20, the access procedures defined in each functional state of the APLC are redirect to the corresponding operations in the VMLC instances that implement them. This is the first step taken in the code generation strategy to ensure the correct delegation chain of invocations from the PI of an APLC to the PI of the VMLC that implements it.

3.2 APLC instances

The instances of APLC are declared in the specification of their partition of residence. To understand how APLC are instantiated we must briefly return to their definition and their istantiation parameters. A set of real-time attributes are used as parameters of the generics, in particular:

- for each cyclic operation: *period*, *priority* and *ceiling* to set on the OBCS of the realising VMLC;
- for each sporadic operation: *MIAT*, *priority* and *ceiling* to set on the OBCS of the realising VMLC;
- for each functional state decorated with a non-void synchronization protocol: *ceiling* to set on the OBCS of the realising VMLC.

The value of most of those attributes is determined by way of model transformation in order that the user is not required to provide information that does not really belong in the PIM space.

Listing 3.3: Generic specification of an APLC (with omitted code)

```
1  package <APLC_Name> is generic
2     My_ID : Deployment.Map.ProducerAP_Instances;
3     -- for each cyclic operation
4     <OP_Name>_Priority : Priority;
5     <OP_Name>_Period : Integer;
6     <OP_Name>_Ceiling : Any_Priority;
7     -- for each sporadic operation
8     <OP_Name>_Priority : Priority;
9     <OP_Name>_MIAT : Integer;
10    <OP_Name>_Ceiling : Any_Priority;
11    -- for each protected state accessed by at least one operation
12    <State_Name>_Ceiling : Any_Priority;
13    package <APLC_Name>_Factory is
14       -- code fragments omitted
15    end <APLC_Name>_Factory;
16    -- code fragments omitted
17 end <APLC_Name>;
```

Listing 3.4: Instantiation of an APLC

```
1  package <Node_Name>.<Partition_Name> is
2    procedure Initialize;
3    -- for each APLC instance deployed on the partition
4    <APLC_Instance_Name> is new
5       <APLC_Name>.<APLC_Name>_Factory
6          (My_ID => <ID>,
7           Instantiation_Parameter_1 => <Value>,
8           Instantiation_Parameter_N => <Value>);
9  end <Node_Name>.<Partition_Name>;
```

Finally, before commenting on procedure *Initialize* shown in listing 3.4 we must return one last time to the generic definition of the APLC type in listing 3.3, to look at the implementation of the operations defined in generic package *<APLC_Name>_Factory*.

Listing 3.5: Getters and Fulfill_RI

```
1  package body <APLC_Name>_Factory
2    --  for each functional state
3    function Get_<State_Name>_As_<State_Type> return <State_Type>_Ref is
4    begin
5       return <State_Name>_Instance'access;
6    end Get_<State_Name>_As_<State_Type>;
7
8    function Get_<State_Name>_As_<State_SuperClass_Type>
9       return <State_SuperClass_Type>_Ref is
10   begin
11      return <State_Name>_Instance'access;
12   end Get_<State_Name>_As_<State_SuperClass_Type>;
13   --  end for each
14
15   procedure Fulfill_RI
16      (<State_Name1>_<State_Attribute1> : <OPCS_Type_1>_Ref;
17       --  ...
18       <State_Name1>_<State_AttributeM> : <OPCS_Type_M>_Ref;
19       --  ...
20       <State_NameN>_<State_Attribute1> : <OPCS_Type_S>_Ref;
21       --  ...
22       <State_NameN>_<State_AttributeK> : <OPCS_Type_T>_Ref) is
23   begin
24      <State_Name1>_Instance.Set_<State_Attribute1> (<Attribute_Value>);
25      --  ...
26      <State_Name1>_Instance.Set_<State_AttributeM> (<Attribute_Value>);
27      --  ...
28      <State_NameN>_Instance.Set_<State_Attribute1> (<Attribute_Value>);
29      <State_NameN>_Instance.Set_<State_AttributeK> (<Attribute_Value>);
30   end Fulfill_RI;
31 end <APLC_Name>_Factory
```

For each functional state we define a set of getters which return the embedded states of the APLC cast to the applicable superclass as defined in the Functional View (whether a class or an interface).

Procedure *Fulfill_RI* instead is used to set the class attributes of the functional states toward which the RI of the APLC are directed. Procedure *Set_State_Attribute* is directly inherited from the OPCS superclass of the functional state of interest (cf. lines 21-22 of listing 2.2). As a result of the execution of procedure *Fulfill_RI* the APLC for which it was invoked will be bound, correctly and exclusively, to the desired RI.

Listing 3.6: Initialization of an APLC instance

```
1  package body <Node_Name>.<Partition_Name> is
2     procedure Initialize is
3     begin
4        -- For each APLC with non-void RI
5        <APLC_Instance_Name>.Fulfill_RI
6           (<State_Name>_<State_Attribute> =>
7              <OPCS_Type>_Ref
8                 (<APLC_Istance_Name>.Get_<State_Name>_As_<OPCS_Type>));
9     end Initialize;
10 end <Node_Name>.<Partition_Name>;
```

In the initialization code of the partition, procedure *Fulfill_RI* assigns the functional state attributes of each resident APLC a reference to the APLC instance that satisfies the RI. The reference is cast to the *<OPCS_Type>* specified by the signature of the PI that has been bound to the RI of interest, in order that all functional calls be eventually treated as they were meant in the Functional View.

Chapter 4

An Illustrative Example

In order to shed more light on the operational aspects of the code generation strategy we shall use a simple example as a case study. We will model a tiny system composed of a producer and a consumer.

The Functional View of the system, which specifies the sequential behaviour of the system will be as follows:

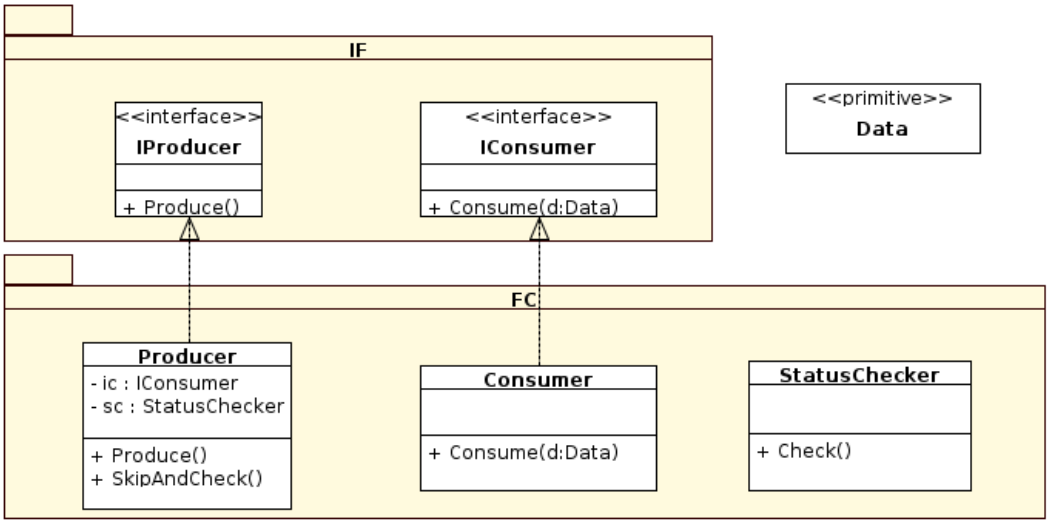

Figure 4.1: Case study: Functional View

Operation *Produce* requires to invoke operation *Consume*. The required call is performed on a reference to interface *IConsumer*. Operation *SkipAndCheck* requires to invoke operation *Check*. The required call is performed on a reference to class *StatusChecker*.

Let us now briefly examine some fragments of code generated for the Functional View of the example:

Listing 4.1: Definition of type Producer

```
1 type Producer is new Controlled and IProducers.IProducer with record
2    ic : IConsumers.IConsumer_Ref;
3    sc : StatusCheckers.StatusChecker_Ref;
4 end record;
```

Class *Producer* implements interface *IProducer* and contains references to an object that implements the *IConsumer* interface and the *StatusChecker* class respectively.

Listing 4.2: Class members of Producer

```
1  procedure SkipAndCheck (This : in out Producer) is
2     -- Invoked RI
3     --+ sc.Check : 1 invocation
4  begin
5     -- User-defined code here
6     Print("Producer#SkipAndCheck");
7     This.sc.Check;
8  end SkipAndCheck;
9
10 procedure Produce (This : in out Producer) is
11    use Datas;
12    -- Invoked RI
13    --+ ic.Consume : 1 invocation
14 begin
15    -- User-defined code here--
16    Print ("Producer#Produce");
17    This.ic.Consume (Data_Default_Value);
18 end Produce;
```

Looking at the code generated for methods *Produce* and *SkipAndCheck* of class *Producer*, we observe that the invocation on the respective RI simply consists of a call to the applicable methods of the relevant appropriate class member of *IConsumer* and *Producer* respectively.

Now let us design the APLC for the example system.

We want APLC *ProducerAP* to embed a functional state typed to class *Producer* and a functional state typed to class *StatusChecker*. The PI and RI of that APLC are automatically derived from the applicable specifications in the Functional View. APLC *ConsumerAP* instead embeds a functional state typed to class *Consumer*. We complete the modeling of the example system by specifying the desired concurrent behaviour of the PI (and RI) of each APLC.

We stipulate that operation *Produce* of APLC *ProducerAP* must be performed cyclically. We also allow that clients of APLC *ProducerAP* can issue deferred requests for the execution of operation *SkipAndCheck*. Now, since PI operation *SkipAndCheck* invokes operation *Check*, which also is published as a PI of the same APLC, we must protect the execution of operation *Check* against concurrent invocations. synchronization protocol. We complete the design of the Interface View by requiring operation *Consume* published in the PI of APLC *ConsumerAP* to have a sporadic behavior.

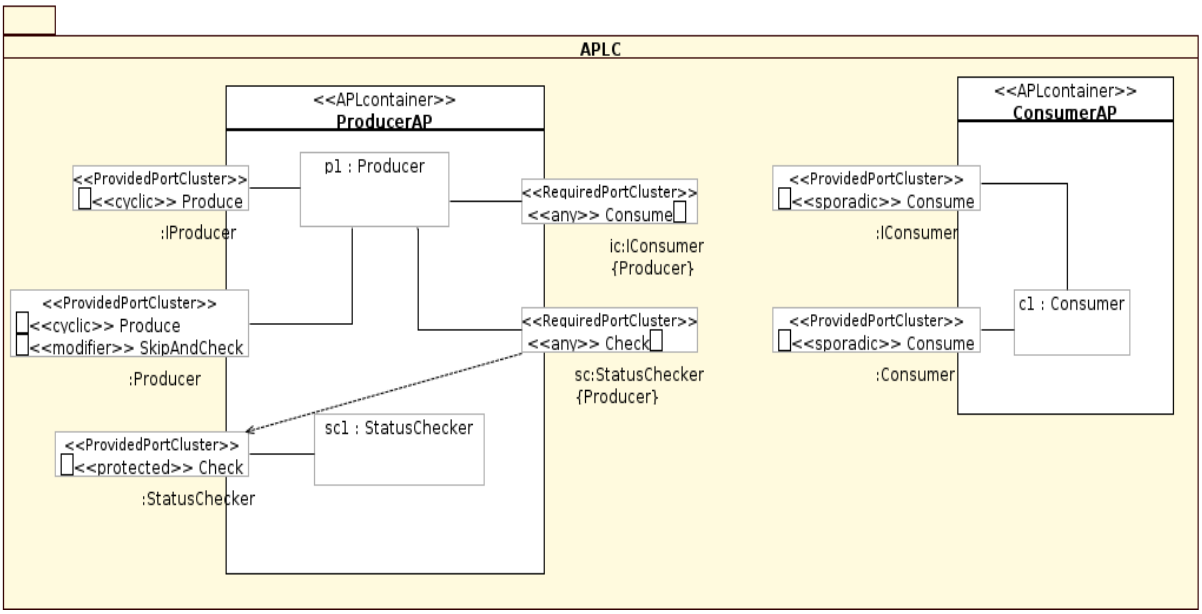

Figure 4.2: Case study: APLC types

In figure 4.3 we create a single instance of each APLC defined in the system. We then interconnect RI *Consume* of APLC instance *ProducerAP_Inst* to PI *Consume* of APLC instance *ConsumerAP_Inst.* The interconnection is permissible because the profile of the RI and the corresponding PI are compatible. Thanks to this interconnection, the RI of *ProducerAP_Inst* is satisfied and thus that APLC instance can fully and correctly discharge its functional obligations toward the system. The only other interconnection in the system is traced between RI *Check* in *ProducerAP_Inst* (required by operation *SkipAndCheck*) published in the PI of the same APLC instance) and PI *Check* subsumed by the incorporation of class *StatusChecker* in *ProducerAP_Inst.* We had seen that already when specifying APLC type *ProducerAP.*

After transformation, the following set of VMLC is generated:

- a cyclic VMLC with a modifiers for *ProducerAP_Inst*, with *Produce* as the nominal operation and *SkipAndCheck* as the modifier;
- a protected VMLC for *ProducerAP_Inst*, with *Check* in its PI;
- a sporadic VMLC for *ConsumerAP_Inst*, with *Consume* as the nominal operation.

We can now follow the delegation chain from the PI of *ProducerAP_Inst* down to the VMLC embedded in its implementation. The specification of *ProducerAP* two new types are defined:

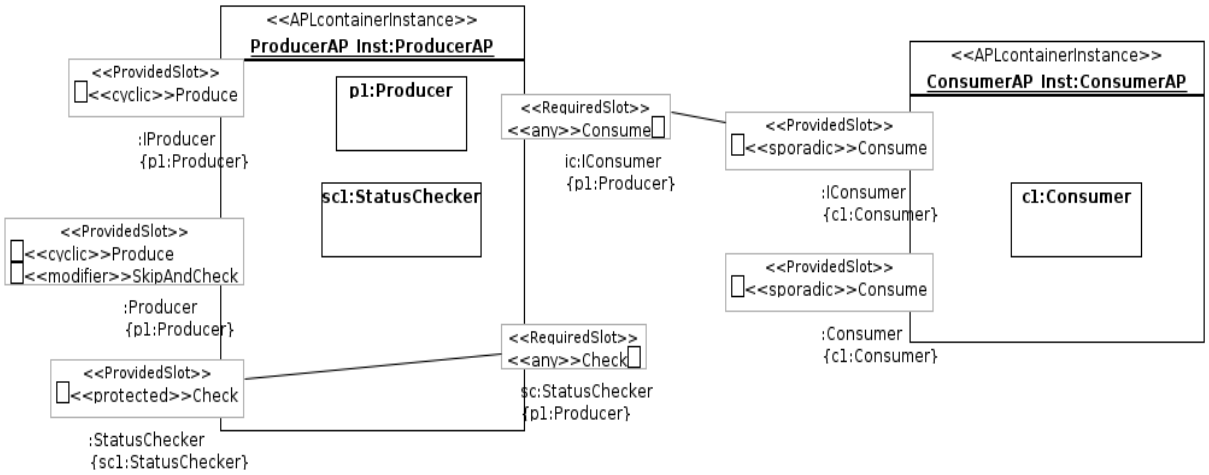

Figure 4.3: Case study: APLC instances

Listing 4.3: Definition of Functional States

```
1  -- in the specification of ProducerAP
2  type p1_T is new Producers.Producer with record
3     SkipAndCheck_0_Ref : access procedure;
4  end record;
5
6  overriding
7  procedure SkipAndCheck (This : in out p1_T);
8
9  type sc1_T is new StatusCheckers.StatusChecker with record
10    Check_0_Ref : access procedure;
11 end record;
12
13 overriding
14 procedure Check (This : in out sc1_T);
```

The PI of *ProducerAP* can then expose procedures that are an indirection to the access procedure types just defined. For example:

Listing 4.4: Implementation of an operation in the PI of ProducerAP

```
1  -- in the body of ProducerAP
2  procedure SkipAndCheck (This : in out p1_T) is
3  begin
4     This.SkipAndCheck_0_Ref.all;
5  end SkipAndCheck;
```

Let us now focus on the definition of the cyclic VMLC embedded in the implementation of *ProducerAP_Inst*.

Listing 4.5: Definition of cyclic VMLC embedded in ProducerAP_Inst

```
1  -- in the specification of Cyclic VMLC p1_Produce_cyclic
2  procedure SkipAndCheck;
3
4  overriding
5  procedure My_OPCS (Self : in out SkipAndCheck_Param_T);
6
7  -- in the body of p1_Produce_cyclic
8  procedure My_OPCS (Self : in out SkipAndCheck_Param_T) is
9  begin
10    Self.OPCS_Instance.SkipAndCheck;
11    Self.In_Use := False;
12 end My_OPCS;
13
14 procedure Getter (Req : out Request_Descriptor_T) is
15 begin
16    Protocol.Get_Request (Req);
17 end Getter;
18
19 procedure Cyclic_Operation is
20 begin
21    OPCS_Instance.Produce;
22 end Cyclic_Operation;
23
24 procedure SkipAndCheck is
25 begin
26    Protocol.SkipAndCheck;
27 end SkipAndCheck;
28
29 package My_Cyclic_Task is new Cyclic_Task_ATC (Cyclic_Operation, Getter);
```

When procedure *SkipAndCheck* in the PI of the VMLC is called, it performs a call indirection to the OBCS (Protocol) where the posting of the request takes place.

Procedures *Getter* and *Cyclic_Operation* are first defined and then passed as instantiation parameters to the Cyclic Thread with modifier that we have seen included in the shared library of archetypes (cf. section 2.2.2). If *Getter* fetches from the OBCS (Protocol) and returns a request descriptor with *ATC_REQ* tag the Thread executes procedure *My_OPCS*, which resolves in *OPCS_Instance* to the modifier operation *SkipAndCheck* that was passed as an actual parameter to the instantiation of the VMLC. If *Getter* returns a *NO_REQ* descriptor instead, the Thread executes procedure *Cyclic_Operation*.

In order to have a complete overview of the full call chain of the PI of *ProducerAP* we must understand how the link between the PI of the APLC and the PI of the VMLC embedded in it are set up. In other words, we have to understand the principles of the delegation chain as realized by the code generation rules.

The delegation chain in question is established in the package body that implements *ProducerAP*.

Listing 4.6: APLC PI delegation to VMLC PI

```
1  package ProducerAP
2     -- code omitted
3     package body ProducerAP_Factory is
4        -- code omitted
5     begin
6        p1_Instance.SkipAndCheck_0_Ref :=
7           My_p1_Produce_Cyclic_Container.SkipAndCheck'access;
8        sc1_Instance.Check_0_Ref :=
9           My_sc1_Protected_Container.Check'access;
10       -- code omitted
11    end ProducerAP_Factory;
12 end ProducerAP;
```

The reader can now appreciate that when calling operation *ProducerAP.SkipAndCheck* an indirection to the procedure pointed by access pointer *p1_Instance.SkipAndCheck_0_Ref* occurs. That pointer is assigned to procedure *SkipAndCheck* of the Cyclic VMLC with modifier, which in turn redirects the call to the OBCS of the VMLC, where the invocation request is reified and stored.

Next we want to show how an APLC instance can issue calls to its own RI. In other words, we wan to understand how instances are informed about which APLC instance satisfies their RI. In this way we will appreciate the differences in the handling of the case in which the RI operation is satisfied by a PI operation published by the issuing or a distinct APLC.

Let us commence by examining the declaration of APLC instances as it is realized in the initialization code generated for the partition where they are deployed on:

Listing 4.7: Initialization code (with example values)

```
1  -- In the initialization code of the partition of residence (spec)
2  package ProducerAP_Inst is new
3     ProducerAP.ProducerAP_Factory
4       (My_ID => Deployment.Map.ProducerAP_Inst,
5        p1_Produce_Cyclic_Thread_Priority => 1,
6        p1_Produce_Cyclic_Obcs_Ceiling => 1,
7        p1_Produce_Cyclic_Period => 4_000,
8        sc1_Protected_Ceiling => 1);
9
10 package ConsumerAP_Inst is new
11    ConsumerAP.ConsumerAP_Factory
12      (y_ID => Deployment.Map.ConsumerAP_Inst,
13       c1_Consume_Sporadic_Thread_Priority => 2,
14       c1_Consume_Sporadic_Obcs_Ceiling => 2,
15       c1_Consume_Sporadic_MIAT => 2_000);
16
17 -- In the initialization code of the partition of residence (body)
18 procedure Initialize is
19 begin
20    ProducerAP_Inst.Fulfill_RI
```

```
21      (p1_ic => IConsumers.IConsumer_Ref
22         (ConsumerAP_Inst.Get_c1_as_IConsumer));
23  end Initialize;
```

The istantiation is straightforward and contains all parameters required by the APLC type. The RI of *ProducerAP_Inst* is satisfied by providing the reference to *ConsumerAP_Inst*.

Procedure *Fulfill_RI* is defined in the body of *ProducerAP*.

Listing 4.8: Procedure Fulfill_RI

```
1  -- in Producer APLC (body)
2  procedure Fulfill_RI (p1_ic : IConsumers.IConsumer_Ref) is
3  begin
4     p1_Instance.Set_ic (p1_ic);
5  end Fulfill_RI;
```

Procedure *Set_ic* simply sets the reference to *IConsumer_Ref* to which invocations of the RI operation have to be directed.

Listing 4.9: The Setter procedure that binds the target of RI invocations

```
1  -- in the body of type Producer (functional specification).
2  procedure Set_ic (This : in out Producers.Producer;
3                    v : IConsumers.IConsumer_Ref) is
4  begin
5     if This.ic = null then
6        This.ic := v;
7     end if;
8  end Set_ic;
```

At this point we have returned to the functional specification from which where our example begun. We should now therefore fully understand that in the implementation of operation *Produce* which we saw in listing 4.2, member *ic* (line 13 in that listing) is a reference to a specific APLC instance, *ConsumerAP_Inst* in this particular case. This implies that *This.ic.Consume* is a call to the PI of *ConsumerAP*, which is in turn resolved to the appropriate delegation chain.

Let us now look at how the RI for operation *Check* is satisfied. A link was set at APLC *type level* which satisfies the RI with an operation of the same APLC. That condition is reflected directly in the definition of the APLC *type*, as follows:

Listing 4.10: Intra-component RI/PI binding

```
1  package ProducerAP is
2    -- code omitted
3    package body ProducerAP_Factory is
4       --code omitted
```

```
5   begin
6      -- code omitted
7      sc1_Instance.Check_0_Ref := My_sc1_Protected_Container.Check'access;
8      p1_Instance.Set_sc (sc1_Instance'access);
9   end ProducerAP_Factory;
10  end ProducerAP;
```

This setting implies that even if operation *Check* is actually satisfied by a PI of the APLC of belonging, it is subject to the appropriate concurrent semantics (protected concurrent kind in the case in question).

We may finally follow the delegation chain of RI invocation *This.ic.Consume* made at line 17 in listing 4.2.

Listing 4.11: Definition of functional state c1_T for the Consumer class

```
1  type c1_T is new Consumers.Consumer with record
2     Consume_1_Ref : access procedure (d : in Datas.Data);
3  end record;
4
5  overriding
6  procedure Consume (This : in out c1_T; d : in Datas.Data);
```

In listing 4.11 we show the definition of the functional state for the *Consumer* class in the specification of APLC *ConsumerAP*.

Listing 4.12: Implementation of the Consumer AP (code omitted)

```
1  package body ConsumerAP is
2     procedure Consume (This : in out c1_T;
3                        d : in Datas.Data) is
4     begin
5        This.Consume_1_Ref.all (d);
6     end Consume;
7
8     package body ConsumerAP_Factory is
9        c1_Instance : aliased c1_T;
10
11       package My_c1_Consume_Sporadic_Container is new
12          c1_Consume_Sporadic.c1_Consume_Sporadic_Factory
13             (My_ID => Deployment.Map.ConsumerAP_APLC_To_VMLC (My_ID)
14                          (Deployment.Map.My_c1_Consume_Sporadic),
15              Thread_Priority => c1_Consume_Sporadic_Thread_Priority,
16              Ceiling => c1_Consume_Sporadic_Obcs_Ceiling,
17              MIAT=> c1_Consume_Sporadic_MIAT,
18              OPCS_Instance => Consumers.Consumer(c1_Instance)'access);
19
20       function Get_c1_As_IConsumer return IConsumers.IConsumer_Ref is
21       begin
22          return c1_Instance'access;
```

```
23        end Get_c1_As_IConsumer;
24
25     begin
26        c1_Instance.Consume_1_Ref :=
27           My_c1_Consume_Sporadic_Container.Consume'access;
28     end ConsumerAP_Factory;
29  end ConsumerAP;
```

The overriden operation *Consume* at lines 2-6 in listing 4.12 uses an access procedure to perform a delegation to the PI of the implementing Sporadic VMLC (*My_c1_Consume_Sporadic_Container*, lines 11-18). The instantiation parameters of the VMLC at lines 13-18 are the instantiation parameters of the instance of *ConsumerAP* as we saw them at lines 12-15 of listing 4.7.

Listing 4.13: Implementation of the Consumer Sporadic VMLC (code omitted)

```
1  package body c1_Consume_Sporadic is
2     procedure My_OPCS (Self : in out consume_Param_T) is
3     begin
4        Self.OPCS_Instance.Consume (Self.d);
5        Self.In_Use := False;
6     end My_OPCS;
7
8     protected body OBCS is
9        procedure Update_Barrier is
10       begin
11          Pending := (Obcs_Queue.Pending > 0);
12       end Update_Barrier;
13
14       entry Get_Request (Req : out Request_Descriptor_T;
15                          Release : out Time) when Pending is
16       begin
17          Release := Clock;
18          Get (Obcs_Queue, Req);
19          Update_Barrier;
20       end Get_Request;
21
22       procedure Consume (d : in Datas.Data) is
23       begin
24          if Consume_Params.Buffer (Consume_Params.Index).In_Use then
25             Increase_Index (Consume_Params);
26          end if;
27          Consume_Param_T_Ref (Consume_Params.Buffer
28             (Consume_Params.Index)).d := d;
29          Put (Obcs_Queue,
30               START_REQ,
31               Consume_Params.Buffer (Consume_Params.Index));
32          Increase_Index (Consume_Params);
33          Update_Barrier;
34       end Consume;
```

```
35    end OBCS;
36
37    package body c1_Consume_Sporadic_Factory is
38       -- code omitted
39       Protocol : aliased OBCS (Ceiling,
40                                Consume_Ref_Par_Arr'access);
41       procedure Getter (Req : out Request_Descriptor_T;
42                         Release : out Time) is
43       begin
44          Protocol.Get_Request (Req, Release);
45       end Getter;
46
47       package My_Sporadic_Task is new Sporadic_Task (Getter);
48       Thread : My_Sporadic_Task.Thread_T (Thread_Priority, MIAT);
49
50       procedure Consume (d : in Datas.Data) is
51       begin
52          Protocol.Consume (d);
53       end Consume;
54    end c1_Consume_Sporadic_Factory;
55 end c1_Consume_Sporadic;
```

Operation *Consume* (lines 50-54 of listing 4.13) in the PI of the Sporadic VMLC is simply an indirection to the operation by the same name in the OBCS at line 22-34. That operation uses reification: parameter*d* of the invocation is stored in the parameter buffer (lines 27-28) at the appropriate index (*Consume_Params.Index*) and it is then put in the OBCS circular buffer (lines 29-31). At its next activation, the sporadic Thread will invoke procedure *Getter* (cf. line 19 of listing 2.7) which is redirected to the entry of the OBCS (lines 14-20 of listing 4.13).

For the sake of simplicity, suppose that there are no further invocations after the one we are analysing. Barrier *Pending* is opened, since the OBCS queue holds one execution request. Procedure *Get* of the OBCS constructs a request descriptor using the stored parameter buffer (lines 46-47 in listing 2.9) and updates the *Self.Pending* variable (line 57 in the same listing) which is used for the barrier of the OBCS.

When returning from procedure *Get* (line 18 of listing 4.13), *out* parameter *Req* contains the parameter buffer in which the parameters of the original invocation are stored. When the sporadic Thread will access the request parameters (line 19 in listing 2.7), it invokes procedure *My_OPCS* of the sporadic VMLC using the parameters buffer of the original invocation (lines 27-28 of listing 4.13). Finally, procedure *My_OPCS* extracts parameter *d* of the original invocation (*Self.d* at line 4 in listing 4.13) and uses it in the invocation of operation *Consume* on *OPCS_Instance* where the sequential behaviour of the system is defined (procedure *Consumer.Consume (d : Data)* in that particular case).

Chapter 5

Open Issues

Functional behaviour of the OBCS. In the current code generation strategy the sporadic OBCS embeds two separate queues for incoming requests: one queue accepts START requests, that is to say, requests of execution for the *nominal* operation; the other accepts ATC requests, that is to say, requests of execution for any *modifier* operation published in the PI.

This choice is made in accordance with the original concept of a sporadic operation as it derives from HRT-HOOD, the modeling language from which RCM strongly inherits. To better understand that concept, we should look back at analogous concept of cyclic operation with modifiers.

A cyclic VMLC exposes a (private) cyclic operation and zero or more *modifier* operations; for our discussion, the case that matters is when the PI includes at least one modifier operation. The behavior of the cyclic VMLC prescribes that the Thread cyclically executes an operation. If a request for a modifier operation has been posted in the OBCS of the cyclic VMLC, then at the subsequent activation the Thread will execute the modifier operation in preference to the nominal one. In this situation it is evident that the modifier operation is considered more important than the nominal operation, and as such it is executed in preference. A consequence of this behaviour is that if there should be a continuous inbound flow of ATC invocation requests, then the VMLC would continue to service them requests and would consequently skip the execution of the nominal operation.

Let us now complete the analogy with the sporadic VMLC. The sporadic VMLC exposes a *nominal* sporadic operation and zero or more modifier operations. As in the previous case, we are interested in a situation with one or more modifier operations published in the PI of the VMLC. The functional behaviour of the sporadic OBCS illustrated in section 2.2.4) shows that requests for START operation can be executed only if there are no pending ATC requests. This behavior mirrors that of the Cyclic VMLC. This situation however is more delicate. If in fact a continuous inbound flow of requests for modifier operations was directed to a Sporadic VMLC, then all pending requests for nominal sporadic operations would run the risk of starvation. As modifier operations are considered more important, they are always executed first: it is then possible that nominal sporadic operations are delayed indefinitely.

Again, this situation derives from the HRT-HOOD inheritance. However, it is possible that for some system this specific behaviour is not appropriate, since the designer may want to design cohesive operations that have the same ”implicit importance” or urgency. The sporadic VMLC instead always induces potential starvation in the nominal operation because it is considered implicitely ”less important” or ”less urgent”.

To remedy this situation and avoid the risk of starvation a more complex synchronization agent is required for Threaded VMLC. The current implementation of the OBCS is rather simple and future work on RCM code generation shall consider suitable protocol extensions.

Code generation of APLC. The current mapping for APLC has a number of advantages. The concept of APLC is easy to recognize, as the designer views it at model level and retrieves in the generated code. This mapping strategy also succesfully renders the dichotomy between APLC types and instances. VMLC instances are instantiated inside APLC instances, together with OPCS (which can be shared and used by multiple VMLC). The use of a package, coupled with an inner generic package is used to declare types in the main package and instantiate them in the generic package, thereby avoiding code duplication.

The current mapping was deemed the most appropriate since the generated code should also comply with the constraints of the Ada Ravenscar Profile (RP) [BDR98]. In particular, the restriction:

```
1  pragma Restrictions (No_Implicit_Heap_Allocation);
```

severely limits the feasibility of other possible approaches (use of record types for example).

Some problems exist with this strategy however, which should be addressed in future work. The rigid enforcement of the separation of concerns principle has been realized with massive use of dynamic binding. In some situations the use of dynamic binding could be avoided. Consider that for each functional class there is always a derived class (the functional state) that is used to enforce the desired concurrent semantics. This derived class is always created and used even when it is not strictly required.

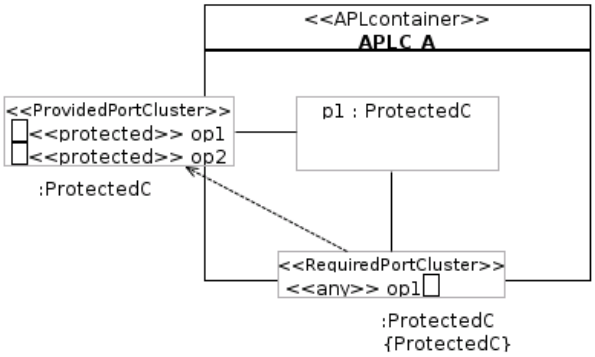

Figure 5.1: Example entailing the deadlock problem

Deadlock in Protected VMLC. Suppose we are in the situation depicted in figure 5.1. The APLC is implemented by a single Protected VMLC that exposes operations *op1* and *op2* in its PI. Now suppose that the execution of *op2* calls *op1* in of the same Protected VMLC.

In the current code generation strategy the call to *op2* is resolved as follows:

- the operation is subject to the Ceiling Locking Policy of the protected object;
- when the flow of control executing *op2* enters the protected object, an indirection to the OPCS is performed;
- a call to *op1* is performed, using dynamic dispatching, inside the OPCS of *op2*;
- the flow of control tries to acquire again the lock to the protected object that it already possess, which incurs a deadlock. The reason why deadlock occurs is that the code generator uses a call indirection mechanism

which causes the call to *op2* to be prefixed as if it were performed from outside of the protected object which was the point of call, and thus outside of the control of the Ceiling Locking Policy.

A revision of the code generation strategy should recognise the case that *op1* is published by the PI of the same protected VMLC as the caller, and so that the call should be readily redirected to the OPCS without passing from the PI of the VMLC (which obviously guarantees the intended concurrent semantics). It is important to notice that this is *not* a limit of RCM, but simply of the current code generation.

It is equally important to also observe that HRT-UML/RCM, in obedience to the general principle of high-integrity systems to ban direct and indirect recursion, prohibits the formation of cycles in call chains in the Interface View; a call chain starts at a deferred operation (i.e., one that is marked either sporadic, cyclic or modifier) and ends at an operation with either a void RI or an RI satisfied by a deferred PI. The check to assure the absence of such call cycles is performed both whenever a new RI-to-PI link is traced and when the user wishes to commit the system model.

Appendix A

Extended Thread Archetypes

In the body of this report we have shown source code in which all constructs devoted to support execution-time monitoring and handling of violation events were omitted. The simplification was meant to let the reader concentrate on the most important concepts of the code generation. Execution-time monitoring and handling of violation events are very important features to the guarantee of property preservation from model to execution.

In this appendix we briefly outline the constructs that are used for execution-time monitoring. The archetypes of the cyclic and sporadic Threads obviousy need to be augmented. We shall only examine the archetype of the sporadic Thread, since the extension of the cyclic Thread follows by analogy.

Listing A.1: Archetype for the Sporadic Thread comprehensive of WCET monitoring (impementation)

```
1  package body Sporadic_Task is
2     task body Thread_T is
3        Req_Desc : Request_Descriptor_T;
4        Next_Time : Time := System_Start_Time + Task_Activation_Delay;
5        Release : Time;
6        Id : aliased Task_Id := Current_Task;
7        WCET_Timer : Timer (Id'Access);
8        Iteration : Integer := 0;
9     begin
10       loop
11          delay until Next_Time;
12          Get_Request (Req_Desc, Release);
13          Set_Handler (WCET_Timer,
14                       Milliseconds (Req_Desc.Params.WCET),
15                       WCET_Violation_Handler);
16          case Req_Desc.Request is
17             when ATC_REQ | START_REQ =>
18                My_OPCS (Req_Desc.Params.all, Release, Iteration);
19             when NO_REQ =>
20                   -- intentional idling
21             when others =>
22                   -- error handling
23          end case;
```

```
24          Iteration := Iteration + 1;
25          Next_Time := Release + Milliseconds (Interval);
26       end loop;
27    end Thread_T;
28 end Sporadic_Task;
```

In order to monitor the execution time, we use *Execution Time Timers*, a novel feature of the Ada 2005 language. In the task body we declare an Execution Time Timer (line 7) that measures the execution time consumed by the thread to which it is attached. Timer managament is done via procedure *Set_Handler* (lines 13-15), which specified the maximum execution time that the thread is allowed to consume and the action to perform in case that limit was exceeded (*WCET_Violation_Handler*). Handler procedure *WCET_Violation_Handler* is to be specified as an instantiation parameter of the generic unit in which the archetype of the Thread resides (see listing A.2). Procedure *Set_Handler* monitors the execution of the Thread operation. (The current placement assumes a single WCET value for all nominal and modifier operations. Should the actual values differ significantly, the code archetype should change accordingly and *Set_Handler* set in each corresponding branch of the case structure.) To this end, the maximum execution time that can be consumed by the current activation (which depends on the operation actually invoked), the request descriptor should be augmented to include the contractual WCET stipulation (line 14).

Listing A.2: Archetype for the Sporadic Thread comprehensive of WCET monitoring (specification)

```
1  generic
2     with procedure Get_Request (Req : out Request_Descriptor_T;
3                                 Release : out Time);
4  package Sporadic_Task is
5     task type Thread_T
6       (Thread_Priority : Any_Priority;
7        Interval : Integer;
8        WCET_Violation_Handler : Timer_Handler) is
9        pragma Priority (Thread_Priority);
10    end Thread_T;
11 end Sporadic_Task;
```

VMLC extensions. The following listing shows an example of sporadic VMLC that includes execution-time monitoring and handling of violations.

Listing A.3: Example of sporadic VMLC with WCET enforcement (with omitted code)

```
1  package body c1_Consume_Sporadic is
2     -- code omitted
3     protected body OBCS is
4        procedure Update_Barrier is
5        begin
6           Pending := (Obcs_Queue.Pending > 0) and My_Mode;
7        end Update_Barrier;
```

```
8
9       entry Get_Request (Req : out Request_Descriptor_T;
10                         Release : out Time) when Pending is
11      begin
12         Release := Clock;
13         Get (Obcs_Queue, Req);
14         Update_Barrier;
15      end Get_Request;
16
17      procedure Switch_To_Safe_Mode is
18      begin
19         My_Mode := True;
20         Update_Barrier;
21      end Switch_To_Safe_Mode;
22
23      -- code omitted
24   end OBCS;
25
26   package body c1_Consume_Sporadic_Factory is
27      -- code omitted
28      Thread : My_Sporadic_Task.Thread_T
29         (Thread_Priority,
30          MIAT,
31          Instances.WCET_Handlers (Deployment.VM_Table (My_ID)));
32
33      procedure Switch_To_Safe_Mode is
34      begin
35         Protocol.Switch_To_Safe_Mode;
36      end Switch_To_Safe_Mode;
37
38      -- code omitted
39   end c1_Consume_Sporadic_Factory;
40 end c1_Consume_Sporadic;
```

Line 29-31 instantiate the Thread augmented with WCET enforcement. The specific *WCET_Violation_handler* procedure passed as the generic instantiation parameter is specified in the deployment information of the system.

For the time being, the policy to handling WCET violations is to set in ”safe mode” the partition of residence of the offending task. In ”safe mode” all threaded VMLC of the partition are set to the ”safe mode”: the WCET handler calls procedure *Switch_To_Safe_Mode* (line 33), which in turn calls the *Switch_To_Safe_Mode* procedure of the OBCS (line 17).

In the current code generation, the strategy for handling of WCET overruns is intentionally kept rather simple. Future work shall devise more powerful policies, most probably taking advantage of extensions of the OBCS protocol, that was briefly mentioned in section 5.

Bibliography

[Bak91] T. P. Baker. Stack-based Scheduling for Realtime Processes. *Real-Time Systems*, 3(1):67–99, 1991.

[BDR98] Alan Burns, Brian Dobbing, and G. Romanski. The Ravenscar Tasking Profile for High Integrity Real-Time Programs. In *Reliable Software Technologies - Ada Europe*, 1998.

[BDV03] Alan Burns, Brian Dobbing, and Tullio Vardanega. Guide for the Use of the Ada Ravenscar Profile in High Integrity Systems. *Technical Report YCS-2003-348, University of York*, 2003.

[BV07] Matteo Bordin and Tullio Vardanega. Real-Time Java from an Automated Code Generation Perspective. In *The 5th International Workshop on Java Technologies for Real-time and Embedded Systems*, 2007.

[ISO05] ISO SC22/WG9. Ada Reference Manual. Language and Standard Libraries. Consolidated Standard ISO/IEC 8652:1995(E) with Technical Corrigendum 1 and Amendment 1, 2005.

[SLR86] Lui Sha, John P. Lehoczky, and Ragunathan Rajkumar. Solutions for some Practical Problems in Prioritized Preemptive Scheduling. In *Proc. of the 7th IEEE Real-Time Systems Symposium*, pages 181–191, 1986.