Project IST-004033

ASSERT

Automated proof based System and Software Engineering for Real-Time Applications

Instrument: IST [FP6-2003-IST-2 4.3.2.5]
Thematic Priority: Embedded Systems

Deliverables D3.1.4-1 Guide for Using AP-level Modelling Containers (Part 1)

Work Package: WP3.1
Due date of deliverable: M33
Actual submission date: 04/November/2007
Start date of Project: September 5th 2004 **Duration:** 3 years

Organization name of lead UPD
Contractor for this deliverable

Issue – Revision I1R0

Project co-funded by the European Commission within the Sixth Framework Programme (2002-2006)		
Dissemination Level		
PU	Public	X
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

(Guide for Using AP-level Modelling Containers) A Gentle Introduction to the HRT-UML/RCM Methodology

Distribution: TSC, DDHRT, DVT, PP, All Project

Prepared by: Daniela Cancila, Marco Trevisan, Tullio Vardanega (UPD)

Checked by: Tullio Vardanega (UPD), Silvia Mazzini (Intecs)

Release type: Internal Public release : 04/Nov/07

Approved by: Passed QPR/IPR

Document Change Record

Issue Revision	Date	Affected Section/Paragraph/Page	Reason for Change/Brief Description of Change
I0R1	13/Sep/07	All	Initial release
I0R2	15/Oct/07	A1, Goal, 1-2, 4, 9	Fixed minor textual errors caught in internal reviews at UPD and Intecs
I0R3	04/Nov/07	Goal, 1-6, 9-11	Amended to address comments from the QPR/IPR
I1R0	14/Nov/07	Front matter, Abstract	Final formatting for formal release

Abstract

The goal that was set for this document in Annex 1 was to provide a "guide to using AP-level modelling containers". In order that this goal can be accomplished more comprehensively, this document contains an informal introduction to HRT-UML/RCM, the methodology based on the Ravenscar Computational Model (RCM) defined in the context of the ASSERT project, where the notion of AP-level (modelling) container was first defined, promoted and pursued. The document is edited in the form of a tutorial addressed to ASSERT system designers, with the objective to ease their familiarization with the HRT-UML/RCM toolset across all modelling views which are traversed as part of the proposed development process. The view in which AP-level (modelling) containers are used is in fact a key pivot element to the entire modelling approach. The reader of this document is expected to be familiar with the basic notions of UML modelling as well as with the foundations of Model-Driven Engineering. The document does intentionally refrains from duplicating information which may be found in the User Guide to the HRT-UML/RCM toolset: this document is therefore best understood by relating the provided information to the actual features, capabilities and actions of the HRT-UML/RCM toolset.

This document is tagged Part 1 because, on account of the existence of a parallel development of the concepts and vision pursued by HRT-UML/RCM, centred on the use of AADL. Accordingly, a Part 2 to contractual deliverable report D3.1.4-1 has been produced separately by the DDHRT team that follows the AADL line of work.

Acknowledgements

Special gratitude goes to Silvia Mazzini (Intecs) for her thorough reading multiple drafts of this tutorial. We wish to thank Stefano Puri (intecs) for his technical insight in the HRT-UML/RCM toolset prototype. We would also like to thank the members of our research group, in particular Marco Panunzio and Matteo Bordin for their help in the sections related to Analysis and Code. Special thanks to Enrico Bini of the Scuola Superiore di Studi Universitari e di Perfezionamento Sant'Anna di Pisa (Italy) for his suggestions of definitions of static, feasibility and sensitivity analysis.

Abbreviations

- APLC Application-level container
- CbyC Correctness by construction
- GNC Guidance Navigation and Control
- MDA Model-driven architecture
- MDE Model-driven engineering
- NA Network access
- OMG Object Management Group
- PI Provided Interface
- PIM Platform-independent model
- POS Position (store)
- PRO Propulsion (manager)
- PSM Platform-specific model
- RCM Ravenscar Computational Model
- RI Required Interface
- TC Telecommand
- TM Telemetry
- TMTC Telemetry and telecommand (handler)
- UML Unified modeling language
- VM Virtual machine
- VMLC Virtual machine level container
- WCET Worst-case execution time

Table of Contents

1 Overview of the Methodology

Over the last few years OMG has actively promoted the use of models to design software through the Model Driven Architecture (MDA) initiative [OMG03] more recently evolved into Model-Driven Engineering (MDE). MDA models are characterized by both different semantic specialization and different levels of abstraction, from a higher to a lower level of abstraction closer to implementation and execution. In the MDA approach, first a *platform-independent model* (PIM) is developed, which describes the business and application logic, and then a series of transformations take place to map elements in the PIM to elements in a *platform-specific model* (PSM) which contains details that are specific to the target platform. Model transformation may be defined by rules and may thus be automated [OMG03]. (Cf. Figure 1.1.): in ASSERT and in the HRT-UML/RCM both features apply.

Figure 1.1: The MDA approach

ASSERT follows the MDA approach and so does the HRT-UML/RCM methodology. MDA models describe the system from different points of views, which are partial representations of the unique underlying model, from the perspective of a related set of user concerns, which are of consequence to the system development. HRT-UML/RCM decomposes the model into *six* views: three of them characterize the PIM and the other ones specify the PSM. The designer only specifies the three views which reside in the PIM space, whereas the PSM views are all automatically generated.

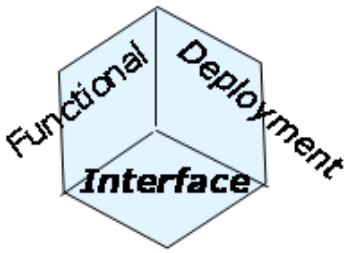

Figure 1.2: The PIM views

The PIM views are:

- The Functional View, which specifies the functional services provided by system components and expresses their sequential behaviour in terms of state machines, classes and interfaces.
- The Interface View, which characterizes the provided and required services of components and declares their intended concurrent behaviour. In this view, a provided service can be specified to execute, for example, as a cyclic operation.

- The Deployment View, which specifies the physical architecture of the system and the way in which software application(s) are to be deployed on it.

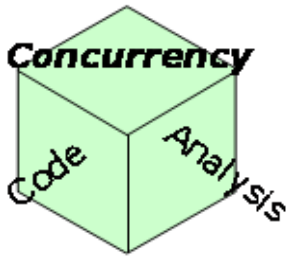

Figure 1.3: The PSM views

The PSM views are:

- The Concurrency View, which specifies (in actual fact, calculates) the concurrent architecture of the system needed to implement the PIM specification of it; this view is designed to be compliant with the Ravenscar Computational Model by construction.
- The Analysis View, which statically determines whether the current implementation of the system is able, under worst-case conditions, to meet all commitments as specified in the interfaces of the systems components.

- The Code, which currently is realized as the Ada 2005 source code representation of the system implementation, destined for execution on an RCM Virtual Machine.

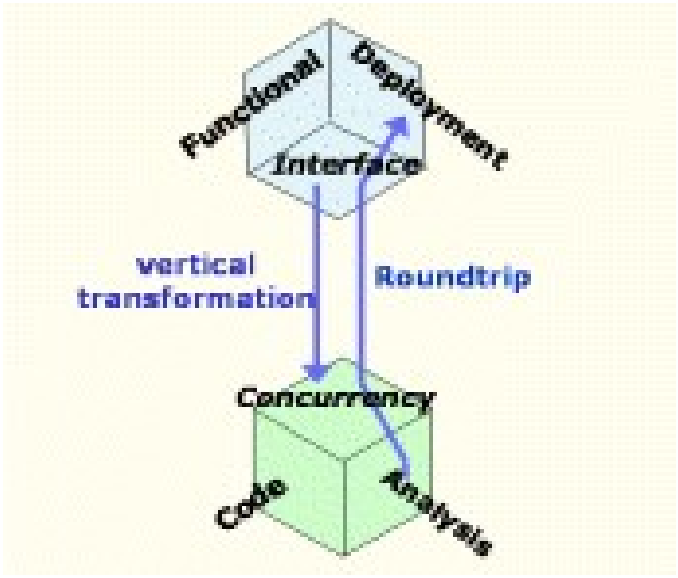

Figure 1.4: Methodology overview

Once the designer has specified the PIM views, fully automatic model transformations allow the user to move from PIM to PSM, and backwards. Within HRT-UML/RCM, the transformation from the Interface View (PIM) to the Concurrency View (PSM) is also known as *vertical transformation* to denote its taking the system down from PIM , under user control, to PSM, supported by automated

rules. (A notable feature of HRT-UML/RCM is that the PSM view is placed *outside* the direct control of the user.) Off-line schedulability analysis may be performed on the system model (as well as any other form of static analysis of interest), whose results are propagated from the Analysis View (PSM) back to the Interface and Deployment View (PIM) through the *round-trip engineering model-transformation*. Finally, the Code View is generated from the Concurrency View.

The HRT-UML/RCM methodological approach exhibits the following properties:

- Preservation of semantic consistency across views.
- Fully automatic PIM to PSM model transformation, with no need for the designer to directly operate on the PSM.
- Correctness by construction (CbyC) of the whole process.
- Round-trip engineering, whereby values measured in the PSM space are promoted to match/update the corresponding values in the PIM space in order that the designer may determine whether the system specifications are met satisfactorily or changes are required.

2 RCM Virtual Machine

The *RCM Virtual Machine (VM)* is an abstraction of the execution platform on which ASSERT applications run and it is a central concept to the entire HRT-UML/RCM methodology. Figure 2.1 illustrates the layered architecture of an ASSERT software system.

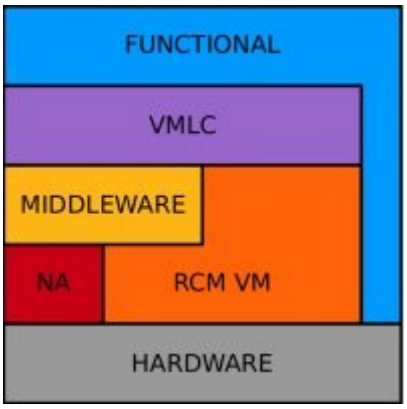

Figure 2.1: Overview of an ASSERT software system

(Some explanatory words are in order to discuss the intended meaning of the illustration in Figure 2.1. The attentive reader may notice that the functional component of the software system depicted in Figure 2.1 reaches down to the *hardware* directly. The intent of this representation is very innocent and simply wants to reflect the fact that some elements of the hardware platform may be directly visible to functional models for reasons that trade transparency and portability for speed of access. The general philosophy used in ASSERT does indeed discourage, but not prohibits, the recourse to direct access of functional models to hardware. A further observation is in order with regard to the Middleware box in Figure 2.1, which is shown to be external to the RCM VM, which in fact it is, and below the VMLC layer. In actual fact, the Middleware part of the ASSERT software system is realized in terms of "primitive" VMLC components which run like any other VMLC on top the RCM VM and use the support of Network Access, NA, components to cater for distribution-transparent communications to the application components included in the Functional component of the system. VMLC, otherwise known as Virtual Machine level containers, are the sole run-time entities allowed to operate on the RCM Virtual Machine. As a result of a complex transformation process VMLC)

A system modelled with HRT-UML/RCM encompasses multiple layers: the whole stack of layers is described here for the reader to get some insight on the mechanism of the run-time structures on which the methodology rests. In fact, the designer need not to be aware of all the layers and all the related run-time mechanisms to benefit from the methodology, since all those details are (intentionally) hidden away from her/him at design time.

The bottom layer is the physical layer, comprised of a network of interconnected computational nodes. On top of the physical layer lies the RCM Virtual Machine, which provides run-time mechanisms required to implement the Ravenscar Computational Model (for instance, run-time enforcement and consistency checks, resource locking, synchronization and scheduling). The physical layer is accessed by the RCM Virtual Machine (in so far as local physical resources are concerned) and by the Network Access module, termed "NA" in the figure. The role of the NA module is to implement the protocol

stack which allows applications to transparently communicate across the network. Although the NA module provides the basic communication services, in HRT-UML/RCM the application software is not allowed to access the network directly: in order to cater for distribution transparency, communication across remote nodes is is always mediated by a "Middleware" layer. The discussion of the operation of the Middleware presently falls outside the scope of this tutorial. Suffice it to say for the moment that all of the internal components of that layer are designed and implemented in full compliance with the RCM; add no semantics to the node-bound computation and handle remote communication in a form which can be regarded as a "distributed RCM". The RCM Virtual Machine is inflexible (as opposed to permissive) in hosting, executing and actively policing the run-time behaviour of the entities that may legally exist and operate in it.

HRT-UML/RCM guarantees valuable software properties by controlling all distributed and concurrent behaviour through the RCM Virtual Machine and the Middleware. Application-level software is build in compliance with specific rules; in particular, all functional code and data must be encapsulated into VMLC. We shall see below that VMLC are automatically built out from transformations of the PIM.

The RCM Virtual Machine concept entails the following features:

1. it is a run-time environment that only accepts and supports "legal" entities; the sole legal entities that may run on it are VMLC, described in section 8; no other run-time entity is permitted to exist and no other can thus be assumed in the model;
2. it provides run-time services that assist VMLC in actively preserving their stipulated properties; mechanisms and services of interest may for instance enable one to:
3. accurately measure the actual execution time consumed by individual threads of control over a given span of activity
4. attach and replenish a monitored execution-time budget to a thread, and then raise an exception when a budget violation should occur;
5. segregate threads into distinct groups, attaching a monitored execution-time budget to individual groups, to be handled in the same way as for threads;
6. enforce the minimum inter-arrival time stipulated for sporadic threads;
7. build fault containment regions around individual threads and groups thereof;
8. attain distribution and replication transparency in inter-thread communication;
9. it is bound to a compilation system that only produces executable code for "legal" entities and rejects the non-conforming ones; run-time checks provided by the Virtual Machine shall cover the extent of enforcement that cannot be exhaustively achieved at compile and link time; the details of the checks to be performed to warrant preservation of the computational model constraints, whether statically or at run time, are given in [BDV03];
10. the number of threads within the system is fixed at design time, and thus no thread may be created at run-time;
11. dynamic allocation of memory is not permitted;
12. it realizes a concurrent computational model provably amenable to static analysis; the model must permit threads to interact with one another (by some form of synchronization mediated by

intermediary non-threaded entities) in ways that do not incur non determinism.

2.1 Computational Model

The computational model entailed by the Virtual Machine directly stems from the Ravenscar Profile [BDV03]. It assumes concurrent threads of execution, scheduled by pre-emptive priority-based dispatching policy, which can intercommunicate - for data-oriented synchronization purposes - by means of monitor-like structures with access protocols based on priority ceiling emulation [GS88] and with support for both exclusion and avoidance synchronization[1]. The system is made of a finite, statically-defined number of non-terminating threads, which infinitely repeat the following execution behaviour:

THREAD execution behaviour

repeat forever {

wait for activation event, which may carry input data; (2.1.1)

execute sequence of non-blocking operations; (2.1.2)

}

Threads do *not* have internal (data) state of their own. Instead, they access data that may be embedded in the same container as the thread's or else global to the system. At step (2.1.1) of execution a thread (logically) draws input from the system state while in step (2.1.2) it may read from and / or write to any particular (data) state, whether local or global. Owing to the effect of priority ceiling emulation any write access performed by a thread to a shared resource fully takes effect (i.e. commits) before any other thread may get to it. The execution stage at step 2.1.2 must only include non-blocking operations (i.e. operations that do not perform self-suspension and / or invocations that may involve conditional wait in access to resource protected by avoidance synchronization). As a direct consequence of this restriction, step 2.1.1 represents the single blocking invocation that may be made by threads.

2.2 Execution Timing

Threads issue jobs (i.e., instances of execution) at either a fixed rate (in which case the issuing thread is called "Periodic" or "Cyclic") or sporadically, with a stipulated minimum time separation between subsequent activations (in which case the issuing thread is termed "Sporadic"). Therefore, threads are either Cyclic or Sporadic in accordance with the nature of the source of the activation events. The source may be attached to the system clock for a periodic event or else to some other system activity for a sporadic event. Cyclic threads inherit a "Period" attribute from the rate attribute specified for the relevant source. Sporadic threads inherit a "Minimum Separation" attribute from the corresponding attribute specified for the designated source of it. During execution the Virtual Machine polices that the timing behaviour of all threads proceeds in keeping with their respective specification. Consequently no threads may issue jobs more frequently than specified and no Cyclic thread may fail to issue a job at the next period short of a failure of the system clock.

1 Exclusion synchronization is the basic run-time mechanism that warrants mutual exclusion (in either read-lock or write-lock mode) in the face of concurrent access. Avoidance synchronization is the complementary run-time mechanism that withholds granting mutually-exclusive access until specific logical conditions which typically depend on the functional state of the shared resource are satisfied.

A Deadline attribute may be attached to execution step 2.1.2 of a thread's specification. This attribute requires that every single activation of that thread must complete within the specified time interval. A thread is fully characterized by the functional contents of execution step 2.1.2, by the nature and arrival rate of its activation event and by the deadline that applies to the completion of every job of it.

3 Toy Examples

In this tutorial, we discuss a modelling example that addresses issues of distribution added on top of the original requirements specified in a well-known case study which has come to be known within the project as the "Toy Example" [Les06], to cover as broad a spectrum as possible of the use of the HRT-UML/RCM methodology.

3.1 Statement of the Problem

Following the definition given in [Les06], the **POS** (Position store) is a data resource shared by two processes: **GNC** (Guidance, Navigation and Control) and **TMTC** (Telecommand/Telemetry):

- TMTC either writes to POS the data values uploaded from ground by a dedicated telecommand (TC in the sequel) or sends a "BOOST_ORDER" command to the **PRO** (Propulsion Manager) component.
- GNC performs a feedback-control loop on POS, by first reading the current value, computing any necessary adjustments and then updating the initial value accordingly.
- PRO (Propulsion manager) periodically executes its default operation unless it receives the "BOOST_ORDER" command, in which case PRO executes the "BOOST_ORDER" operation once, in place of the default one and then resumes nominal execution.

In the regard of the potential interaction between the two distinct update operations on POS, the problem specification stipulates that "If TC [command] occurs when GNC is active, the update of POS has to be delayed [i.e., deferred] until the termination of GNC". To meet this requirement, the system must guarantee *transactional* access (with atomicity, consistency and isolation: "ACI" properties) to POS specifically for use by GNC. The difference between mutually exclusive and transactional access is described in detail in Appendix 3. Transactional access implies mutual exclusion access, but the converse does not hold.

3.2 The Partitioned Toy Example

As shown in Figure 3.1, for the purposes of this discussion, the Toy Example is deployed on one computational node and three logical partitions residing on it. This configuration enables us to illustrate a broader spectrum of issues in the generation of the PSM views of the system.

- Partition P_1: POS and GNC
- Partition P_2: TMTC
- Partition P_3: PRO

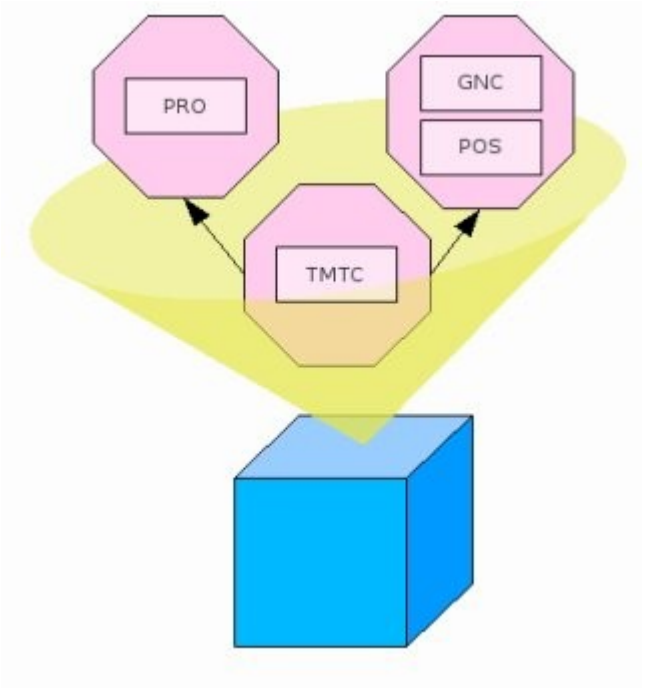

Figure 3.1: The Toy Example deployed on processor and three partitions

3.3 The Distributed Toy Example

As shown in Figure 3.2, the physical graph (that is, the system interconnect) on which the Toy Example is deployed includes three distinct computational nodes. In keeping with the ASSERT assumptions, the network interconnect is point-to-point. We assume that one distinct instance of logical partition P_3 (which includes one instance of PRO process) resides on each of the three physical nodes. This configuration allows us to illustrate issues of interest in the modeling of the PIM views.

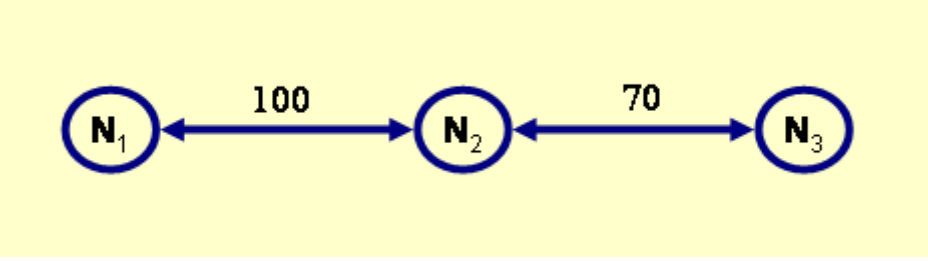

Figure 3.2: Computational nodes and their physical point-to-point interconnect (**Legend**: The numbers that tag the interconnection links between nodes denote the maximum bandwidth capacity of the relevant link expressed in terms of a configurable predefined unit.)

4 The Functional View

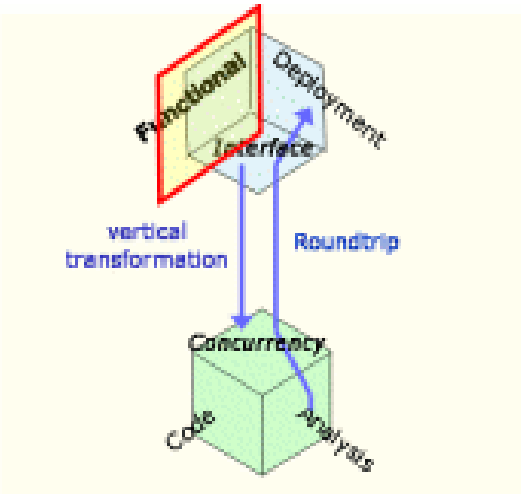

Figure 4.1: The Functional View

The Functional View provides the functional specification of the methods that are to operate within designate containers in the Interface View. The Functional View describes the system using UML notions and notations such as Class Diagrams and State Machine Diagrams. (Cf. figure 4.2) The Functional View allows the designer to model the object-oriented structure of the system and its sequential behaviour only. The concurrent behaviour is specified in the Interface View and realized in the Concurrency View. In order to represent the Functional View, the HRT-UML/RCM tool provides editing capabilities for both Class Diagrams and State Machine Diagrams.

At present, the Functional View can be imported, in XMI format, from a UML model to which the ETH FWProfile [CEPV06] was applied to ensure that the underlying metamodel is compatible with that of HRT-UML/RCM which in turn warrants that the imported model abides by the RCM constraints. In principle any other UML2 Profile which *provably* ensures the required level of compliance with the RCM metamodel could be used to produce a legal Functional Model.

An RCM-compliant functional model can only express and imply (1) time-free and (2) sequential execution semantics and (3) must make no reliance on actions that may incur blocking semantics at run time.

- The prescription that the functional model semantics shall be time free prohibits the use of time-related suspensions (sleep, delay, wait, time-out)
- The prescription that the functional model semantics shall be strictly sequential prohibits the use of constructs which may require the spawning of threads of control, whether directly or indirectly
- The prescription that the functional model shall not include constructs that may incur blocking semantics at run time prohibits the use of invocations which require interventions from outside the application space (dynamic memory allocation is an action that can only be performed outside the application space and, as such, it is forbidden in the RCM functional model).

The Functional View supports the modelling of classes which define operations and attributes with the proper visibility and also interfaces and realization relations between classes and provided interfaces (PI).

An attribute for a given class can also be typed with an interface or with another class: in HRT-UML/RCM this is currently the way to model required services (i.e. the required interfaces, RI) for a given functional class. The PI specifies what the component may do for the environment. The RI specifies what the component may need from the environment to carry out its own tasks.

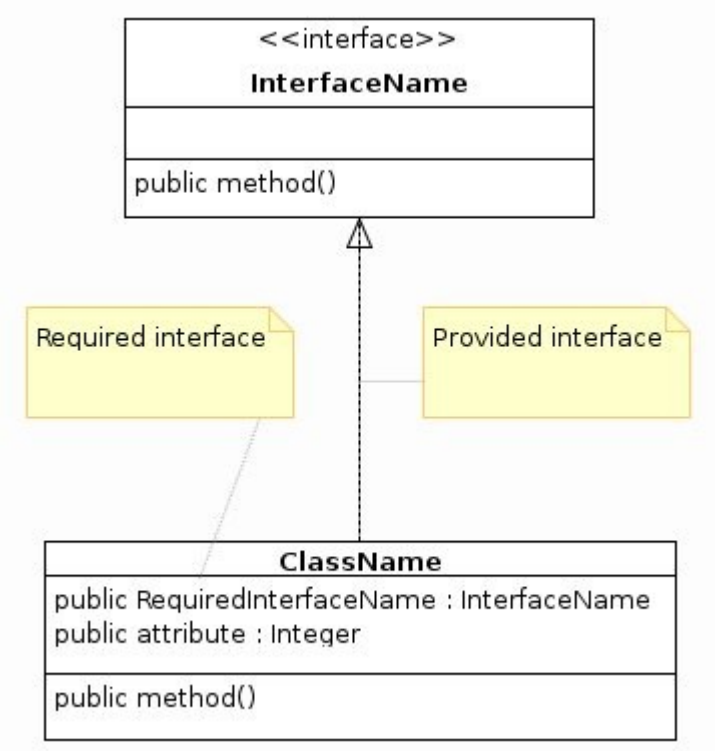

Figure 4.2: Modelling Provided and Required Interfaces using a Class Diagram

Figure 4.2 is a schema of a class used in the Functional View. As part of functional modeling, the designer also needs to specify how many times a provided method uses a required method (a worst-case bound is given when multiple execution paths are possible). This information is crucial for system analysis.

4.1 The Distributed Toy Example: Functional View

Figure 4.3 is the representation of the original toy example in the Functional View; Figure 4.4 shows the Dispatcher component we use instead of the original one.

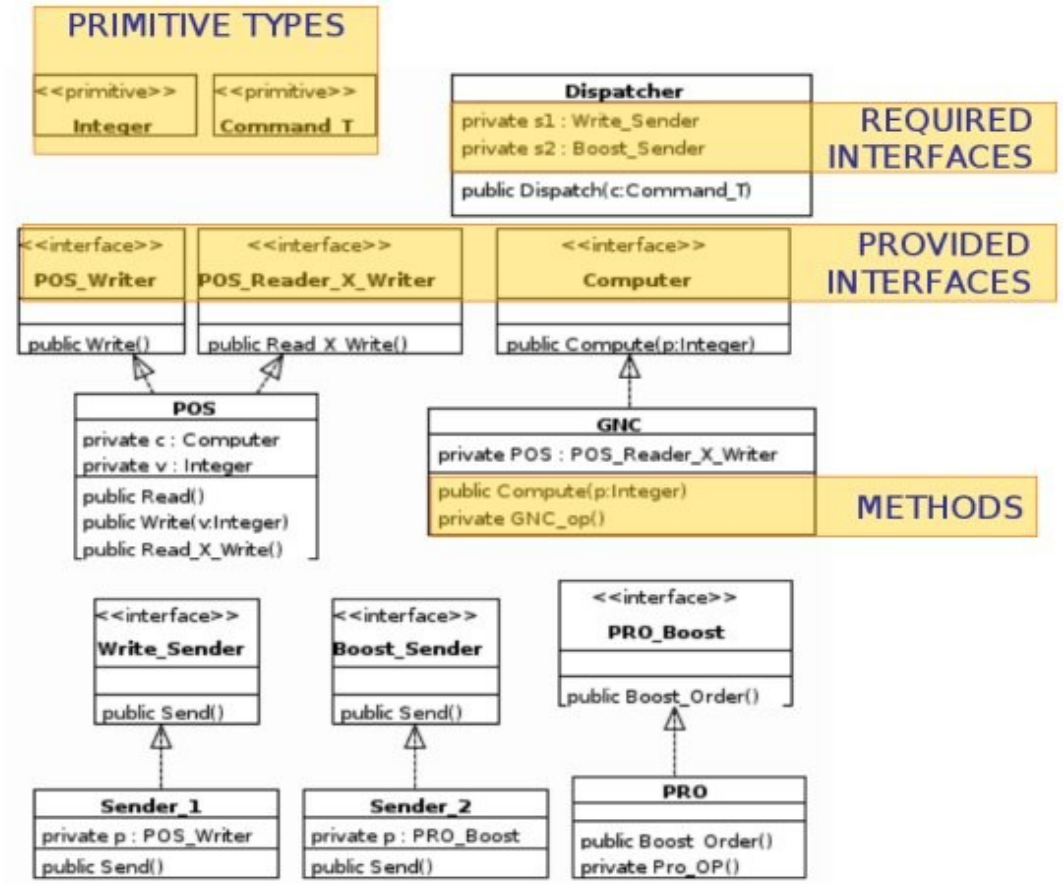

Figure 4.3: Modelling the original Toy Example in the Functional View

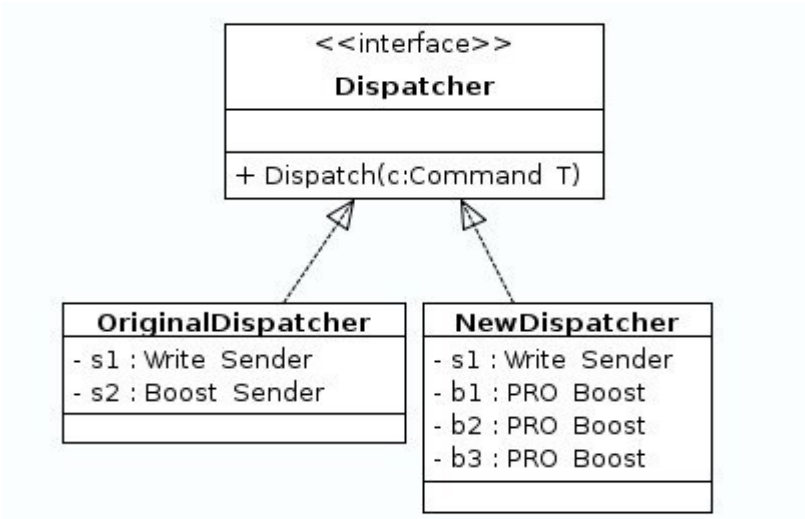

Figure 4.4: The Dispatcher for the distributed Toy Example

POS is a data resource shared by two processes: GNC and TMTC. POS exhibits three service methods, Read, Write and Read_X_Write, each of which must warrant that its caller shall hold

mutually exclusive access on the functional state of the component. Read_X_Write invokes Compute on the RI typed Computer just once per execution. As we shall see, this solution successfully captures the scenario depicted in the original problem specification, but it fails to be a sound object-oriented design in the way it introduces *unneeded* mutual dependency between POS and GNC. A simpler design would *not* have POS depend on Computer, in that POS is a mere data wrapper. The role of GNC, as specified in the original problem, is to read the value, compute some adjustments and eventually write the computed value on POS: there is no obvious need for POS to actively take part into this process. This unusual design is part of the solution currently provided in HRT-UML/RCM to obtain transactional access control. We shall return to this issue in section 5, while a more detailed discussion is provided in Appendix 3.

GNC performs a feedback-control loop on POS, by first reading the current value, computing any necessary adjustments and then updating the initial value accordingly. In order to overcome the potential interaction between the two distinct update operations on POS, GNC accesses to POS with transactional access rights. As a result, GNC exhibits two operations: one *default* operation, which is periodically executed, and a *public* Compute operation, which is used to perform the local adjustment computation on the value read from POS. The problem specification requires that the default operation should be granted transactional access to the POS. Consequently, GNC exhibits one RI typed Reader_X_Writer. When default operation GNC_op() is executed, it invokes RI Read_X_Write operation exactly once.

TMTC either writes to POS the data values uploaded from ground by a dedicated TC or sends to PRO the "BOOST_ORDER" command sent from ground by a TC. TMTC is then functionally decomposed into:

- a Dispatcher element, with a single Dispatch operation, which receives a TC, checks its contents and dispatches it to the appropriate destination for servicing via one of the Send operations included in its RI. If the TC is an update on POS, the dispatcher invokes a Send operation typed Write_Sender, else it chooses one of the three Boost_Order operations on the RI typed PRO_Boost, depending on the physical location of the destination PRO component specified in the incoming TC.
- a Write_Sender element which performs the write on POS via a Write operation on the RI typed POS_Writer

PRO is a process that periodically executes its default operation (PRO_op in figure). PRO also provides the Boost_Order service.

4.2 Open Issues

Visibility attributes over elementary interfaces (or groups thereof), which range: *public* (to any user); *private* (to the container which exposes that interface); and *restricted* (to a specific set of users), are currently only specified in the Interface View. The setting of visibility attributes in the Functional View is only limited to the classical UML semantics (for example, the editor forbids that an operation of an object X which is marked as "private" may appear as "required" in the specification of another object distinct from X). We are currently investigating how the expected finer-grained level of support could be provided.

5 The Interface View

5.1 What the Interface View Provides for

The Interface View

- attaches execution-semantics attributes to the containers that populate this model view and to their provided and required interfaces
- embeds the Functional View into the containers that populate this model view
- delivers the designer from the need to specify details of both hard-real time concurrency (addressed by the automatically-generated platform-specific Concurrency View) and execution platform (which is addressed by the Deployment View)
- traces information flows in relation to the Deployment View, so that the interconnection of provided and required interfaces follows permission, capability and access rules defined at system level
- permits consistent and fully automated mapping to the platform-specific model and to source code with binding to the RCM Virtual Machine (the whole of this mechanic being named "vertical transformation")

In the Interface View the designer only needs to specify some details while others are automatically derived from the specification by the model transformation logic.

1. The designer creates the Interface View by specifying AP-level containers (APLC in the sequel). One essential part of this specification is the inclusion of specific functional components from the Functional View into designated APLC. The criteria for inclusion of functional components in APLC may follow those used for designing UML components (in particular via the UML2 Component Diagram). As we noted in chapter 4, however, not all UML models can be imported in the Interface View, but only those which proceed from a UML Profile compliant with the RCM metamodel (hence with the restrictions and constraints that the RCM imposes on Functional models off that profile).
2. After this inclusion, the HRT-UML/RCM tool automatically generates port clusters within APLC, that is, homogeneous groups of either elementary PI or RI. (See for example Figure 5.3: TMTC.)
3. The designer decorates all PI and RI with semantic decorations, which specify the concurrent behaviour to occur upon invocation of the relevant elementary interface and then sets the desired visibility attribute for port clusters (and thus not on elementary interfaces individually).
4. The designer creates the desired instances of the APLC classes just specified and completes the specification of the interface attributes by setting instance-specific values such as criticality (per APLC) and worst-case execution time, WCET (per elementary interface).
5. The designer interconnects APLC instances in order that RI are bound to PI which are semantically and contractually compatible with one another. Semantic compatibility follows from type matching whereas contractual compatibility is based on value matching (for

example, the WCET set on a PI must not exceed the WCET set on the interconnected RI, where the latter is a contractual specification).

Expected values are attribute values set by the designer on APLC types or RI types or instances, such as Memory Size, WCET, etc. The final system may not fully conform to those values: it is for the designer to decide whether compliance should be warranted for all values mandatorily or else some required values could only be taken to steer the design process in a non-prescriptive manner. Every expected value will eventually be matched by a measured value when the PSM corresponding to the designer-specified PIM will be available from model transformation for various forms of static analyses to be performed.

5.2 What We Mean by Interface View

A model in the Interface View is composed by interconnected APLC. The following table summarizes the set of APLC attributes.

APLC = {P-Identifier, AP-Identifier, Provided Interfaces, Required Interfaces, State}

P-Identifier: Any identifier type, whether string or integer, in a system-wide enumeration, which denotes the partition of residence of the APLC. The enumeration of partitions is defined in the Deployment View of the system and it is referred to in the AP-level model.

AP-Identifier: Any identifier type, whether string or integer, in a system-wide enumeration, which warrants that no two APLC may have the same AP-identifier.

Provided Interface: The set of methods provided (i.e., services supported) by the corresponding APLC.

Required Interface: The set of methods required (i.e., services invoked) by the APLC in the execution of a provided service. The set may be empty, in which case the APLC is said to be self-sufficient to the discharge of its provided interface.

State: The set of functional states, local to the container, which are operated upon by the PI of the APLC.

An elementary port wraps a single method and decorates it with execution semantics. An elementary port and an elementary interface thus designate the same concept. In the literature, both names are used interchangeably: which name to use in a given context depends on the chosen point of view. The term "elementary interface" is used when discussing the methodology from a theoretical point of view, as opposed to the term "elementary port". This term belongs to UML2 terminology and has been borrowed together with the concept it describes in order to realize in practice the "elementary interface" concept.

In the Interface View an elementary port is decorated with concurrent semantic. In the Interface View an elementary port is decorated with concurrent semantics. The following table explains some of the allowable attributes.

immediate: an immediate elementary PI provides a service whose execution is to be carried on by the calling thread. It may be either protected or unprotected.

unprotected: an unprotected elementary PI provides an immediate service with no mutual exclusion guarantees.

protected: a protected elementary PI provides an immediate service with mutual exclusion guarantees. Mutual exclusion is guaranteed only on the data accessed by the operation and encapsulated within the APLC providing the port.

deferred: a deferred elementary PI provides a service whose execution is carried on by a dedicated thread. It may be either nominal or modifier.

nominal: a nominal elementary PI is a deferred elementary PI. It may be either cyclic or sporadic.

modifier: a modifier elementary PI is a deferred elementary PI which represents an alternative behaviour of the thread designated to the carry on the execution of a specific nominal PI.

cyclic: a cyclic elementary PI is a nominal PI. The operation associated with the PI is periodically executed by a dedicated thread. Whenever a modifier of the cyclic PI is invoked, the next time the cyclic PI is executed, the thread carries on the operation associated with the modifier PI instead, subsequently the thread resumes its nominal behaviour.

sporadic: a sporadic elementary PI is a nominal PI. The operation associated with the PI is executed by the dedicated thread whenever the PI is invoked. The operation associated with a modifier of the sporadic PI is executed by the same dedicated thread whenever the modifier PI is invoked.

5.3 The Distributed Toy Example: Interface View

Figures 5.2 and 5.3 show the Interface View for the Distributed Toy Example.

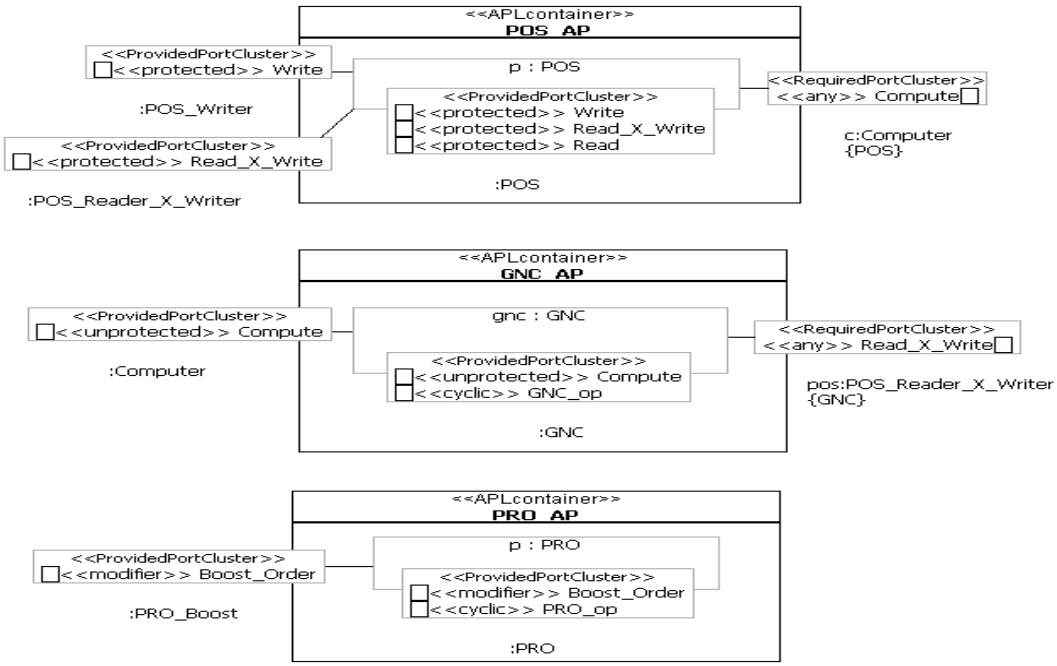

Figure 5.2: The APLC for PRO, POS and GNC

5.3.1 PRO_AP

PRO_AP is an APLC which encapsulates the data and the functional behaviour of a PRO. All the methods of PRO are provided by the PRO_AP APLC through a port cluster which has "private" visibility. Since the visibility of that port cluster is private, the tool editor shows it *inside* the PRO_AP APLC box. The Boost_Order port is also provided within a public port cluster and consequently it is shown *outside* the PRO_AP, too. Private port clusters are shown inside their container APLC, attached to the state that provides data and functionality. Nevertheless, methods within private port clusters are delegated to the APLC just like all other methods of every encapsulated functional entity.

PRO implements the PRO_Boost interface: all the methods declared by this interface are also provided by the PRO_AP through an other port cluster which instead has "public" visibility. The only port accessible outside the APLC is Boost_Order, since it is the only port contained inside a public port cluster.

All the ports of an APLC must be decorated with concurrent semantics. PRO_op is declared as "cyclic", i.e. PRO_AP executes operation PRO_op at a fixed rate specified as an attribute of the corresponding port. The PI of PRO also includes the Boost_Order elementary port, which is declared in PRO_AP as a behaviour modifier: the modifier attribute can be set on a port only in conjunction with another elementary PI in the same APLC which is tagged cyclic or sporadic. The modifier attribute specifies that when the corresponding elementary interface is invoked, it is executed once in place of the nominal operation.

5.3.2 POS_AP

POS_AP encapsulates the data and the functional behaviour of a POS. It provides three port clusters, one of them private. All the ports of POS_AP are "protected", since POS has to be accessed in mutual exclusion. (The HRT-UML/RCM editor does *not* allow yet the designer to distinguish between read-mode and write-mode mutual exclusion attributes.) POS_AP also features a RI which requires a Compute method from a Computer interface. The problem specification requires GNC to have transactional access on POS to perform Read, Compute and Write. HRT-UML/RCM currently addresses this need at the *supplier* side and not (as it should conceptually be) at the *client* side. In the example, POS exhibits a PI (Read_X_Write) which requires Read, Compute and Write: marking Read_X_Write as "protected" grants that every invocation of Read_X_Write (which actually operates as an "envelope" operation) shall execute the Read, Compute and Write operations sequence continually holding mutually exclusive access on POS. More details on this issue appear in Appendix 3.

5.3.3 GNC_AP

GNC_AP encapsulates the data and the functional behaviour of a GNC defined in the Functional View. GNC_AP provides two ports: one for the Compute operation and one for the GNC_op operation, grouped together in a **private** cluster. GNC_AP provides an unprotected Compute port through a port cluster with the Computer interface. In the Functional View, GNC_op requires a Read_X_Write operation on a POS_Reader_X_Writer, which is provided by POS with transactional access rights. An RI has therefore been created to this effect for GNC_AP inside a required port cluster attached to it.

5.3.4 TMTC_AP

TMTC_AP encapsulates the Dispatcher functional state together with a POS_Sender functional state. The only public port cluster of TMTC_AP is the one for Dispatcher, which provides the Dispatch elementary PI. In the example both the Dispatch and the Send PI within Write_Sender have been marked as sporadic. Since the execution of Write on POS is computationally intensive, modelling the Send port as immediate would cause Dispatcher to delay the dispatching of subsequent commands until competition of the whole Read-Compute-Write procedure. Conversely, the execution of Send in POS_Sender is computationally intensive, since it involves the execution of Read, Compute and Write. Dispatching a Boost_Order command received from a TC is far less expensive, since Boost_Order is provided as a deferred service: for this reason Boost_Order is immediately invoked by Dispatch.

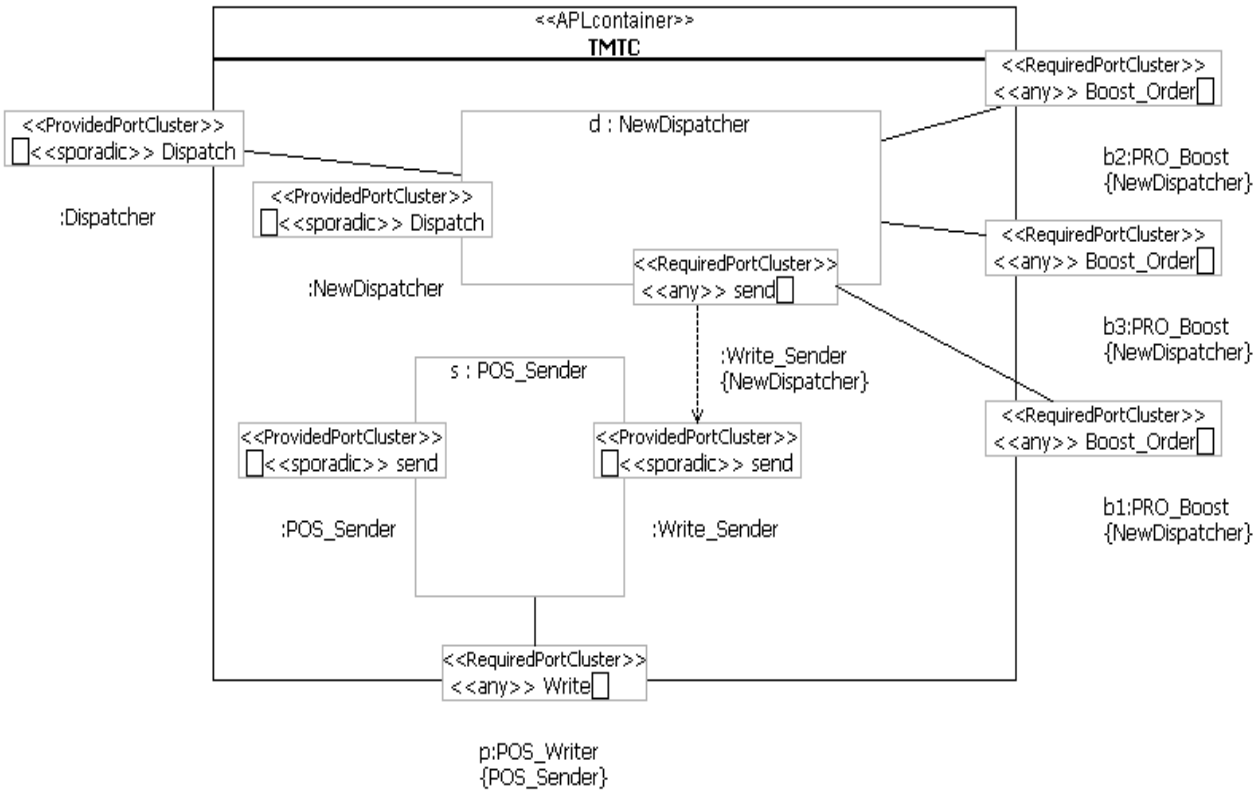

Figure 5.3: The APLC for TMTC

In Figures 5.2 and 5.3 lots of boxes are used which seem to clutter the design space. However, when we consider realistic systems, all these boxes will arguably simplify the system. For example, elementary ports are grouped in port clusters which mirror the structure of the classes defined in the Functional View.

6 The Deployment View

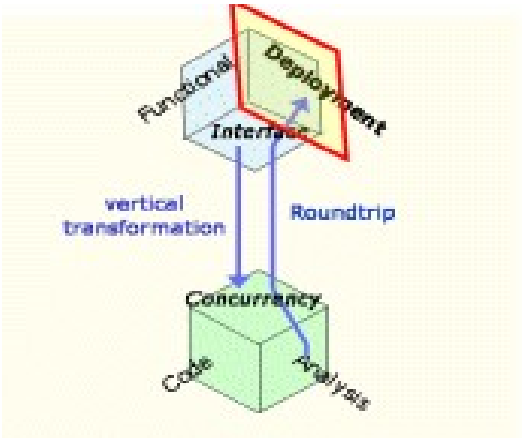

Figure 6.1: Deployment

The designer uses the Deployment View to specify the physical architecture of the system target, to command the mapping between that and the logical architecture of the system and to assess the size, time and communication performance of the assignment.

The unit of allocation on a physical computation node is the *logical partition*, which is a user-defined aggregate of APLC. A single physical node however may host multiple logical partitions. A partition is a fault containment region: the APLC within a partition are isolated in space and time from all the other partitions in the system. (Cf. Figure 6.4.)

In the Deployment View, we consider partitions as "black-boxes" whereas in the weaving process we consider partitions as "white-boxes" and consequently treat their software contents as APLC. In the black-box interpretation of the Deployment View instead we are only interested in a specific attributes of partitions, for example memory size bounds and criticality level. A list of some of the attributes of a partition follows.

Min priority: the minimum priority assignable to a thread within this partition. This is a derived attribute.

Max priority: the maximum priority assignable to a thread within this partition. This is an automatically derived information.

Criticality: a human-readable value which drives the derivation of the min and max priority values. Higher criticality generally translates into higher priority.

Criticality sub-order: whenever more partitions are modelled with the same criticality value this attribute allows to order them nevertheless.

Storage budget: the amount of memory this partition is expected to require.

Time budget: the amount of computational time per time unit this partition is expected to require.

Effective time budget: the amount of computational time per time unit this partition actually requires.

Local scheduling policy: the policy used to schedule threads within this partition. Presently, only

Fixed-Priority Pre-emptive scheduling is available.

Figure 6.2 shows four partitions: P_1, P_2, P_3, P_4 and their assignment to three physical nodes: N_1, N_2, N_3.

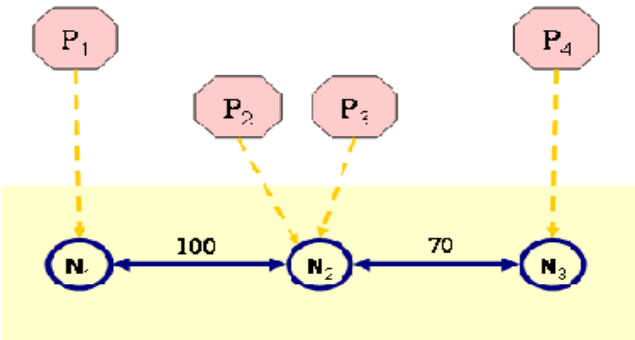

Figure 6.2: Partitions deployed on a physical graph

Table 6.1 below reports the values of some attributes of some logical and physical components of this Deployment View.

Node	Memory Capacity	Partition	Storage Budget	Criticality / Criticality sub-order	Interconnection	Bandwidth
N_1	M_{N1}	P_1	M_{P1}	C_{P1}	N_1 N_2	$B(N_1$ $N_2)$
N_2	M_{N2}	P_2	M_{P2}	C_{P2}	N_2 N_3	$B(N_2$ $N_3)$
N_3	M_{N3}	P_3	M_{P3}	C_{P3}		
		P_4	M_{P4}	C_{P4}		

Table 6.1: Attributes setting on physical nodes, interconnections and logical partitions

Once partitions are assigned to nodes, the HRT-UML/RCM tool automatically checks whether individual nodes have sufficient memory capacity for hosting the partitions to be deployed on them. For example, this check on node N_2 would assess whether the following condition holds and warn the designer in case it did not.

In addition to checking the memory required by a partition against the memory available on the host node, the HRT-UML/RCM tool checks whether the physical communication channels provide enough bandwidth to support the communication between APLC deployed in different nodes. Whenever the amount of data exchanged between two nodes exceeds the available bandwidth, as specified in the Deployment View, a warning is raised to the user.

$$M_{N2} \geq M_{P2} + M_{P3}$$

At present the tool does *not* support the capture of the actual memory size of logical partitions, the relevant values must be therefore input by the designer. The incorporation of this feature is currently in progress.

7 Weaving

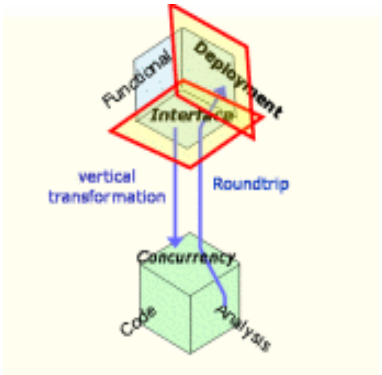

Figure 7.1: Weaving of views

The weaving is the process that merges the specification information provided in the Interface and Deployment Views. The weaving starts by associating each APLC to one partition and the result of it is a merge between logical and physical architecture as illustrated in figure 7.2. The figure has three parts. In the first part, the designer creates APLC in the Interface View and partitions, computational nodes and interconnects in the Deployment View.

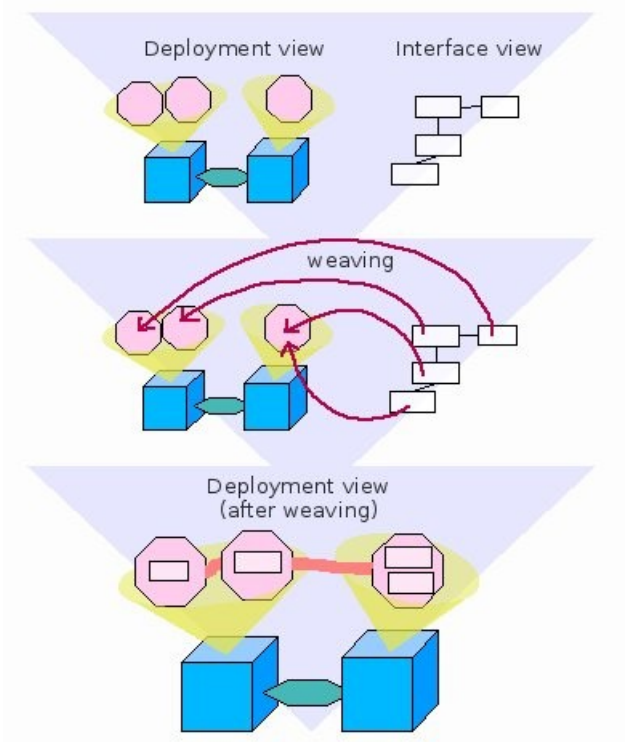

Figure 7.2: An overview of the weaving process

In the second part, APLC are assigned to partitions. The designer has just to drag every instance which composes the system and drop it inside a partition. This assignment is straightforward since the HRT-

UML/RCM tool displays in the Interface View all the partitions defined in the Deployment View. Once the assignment is performed, the model is updated with logical communications between partitions and with memory and bandwidth usage estimates, and the weaving process is completed. Communications among partitions are graphically displayed as assemblies between APLC which belong to them. If a communication between any two partitions result from some APLC assemblies, then the Deployment View shows a logical interconnection between the relevant partitions, directed from the caller to the callee.

A logical interconnection between partitions is considered feasible if it can be mapped (in terms of capacity, capability and rights) on a physical interconnect between the node of residence of the concerned partitions.

Once the Interface and Deployment Views are woven together, the model is further automatically decorated with the information needed to check on the following crucial properties:

- the safety property: a logical communication is allowed if the caller partition has a criticality level greater then or equal to the criticality of the callee partition.
  In case the safety property was violated, that is, the caller partition had a criticality level lower than that of the callee partition, some additional protection mechanisms designated to preserve the system safety have to be provided and/or considered by the analysis; else the weaving process yields a warning which the designer can only remove by changing either the assignment of APLC to partitions or the criticality level attribute of partitions.
- the bandwidth property: a physical interconnection must have enough bandwidth to support all logical communications deployed on it.

$$B_{N1,\,N2} \geq B_{P1,\,P2} + B_{P1,\,P}$$

The current release of the HRT-UML/RCM tool does not support the analysis of the communication bandwidth. Work is in progress to include this feature.

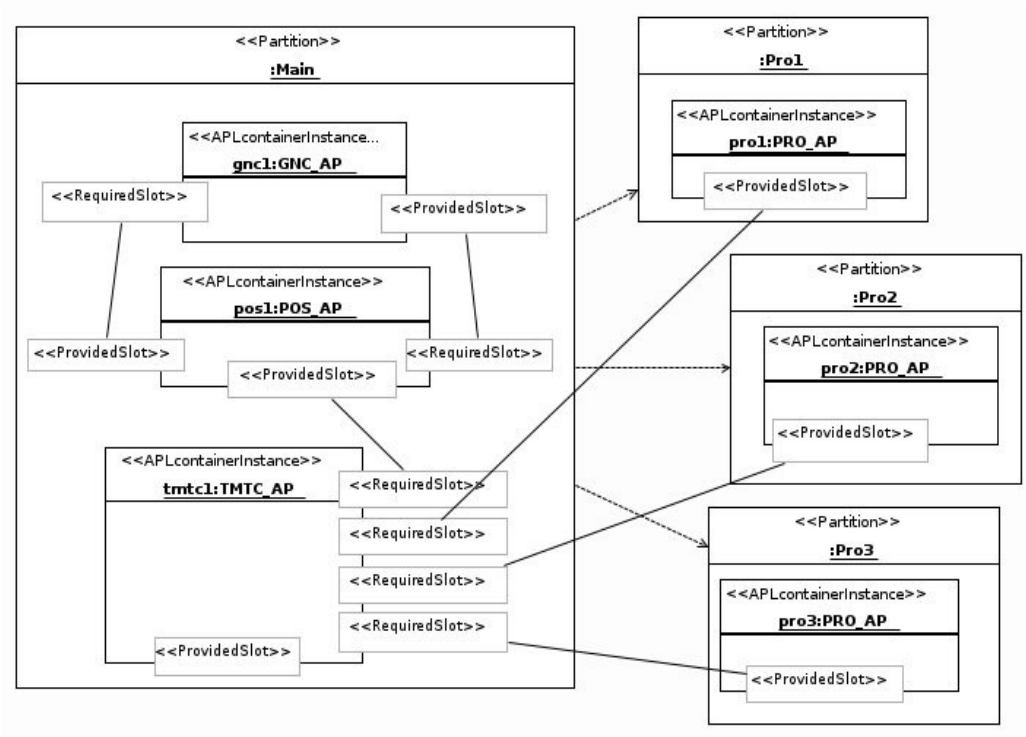

Figure 7.3: Placing instances of APLC into partitions

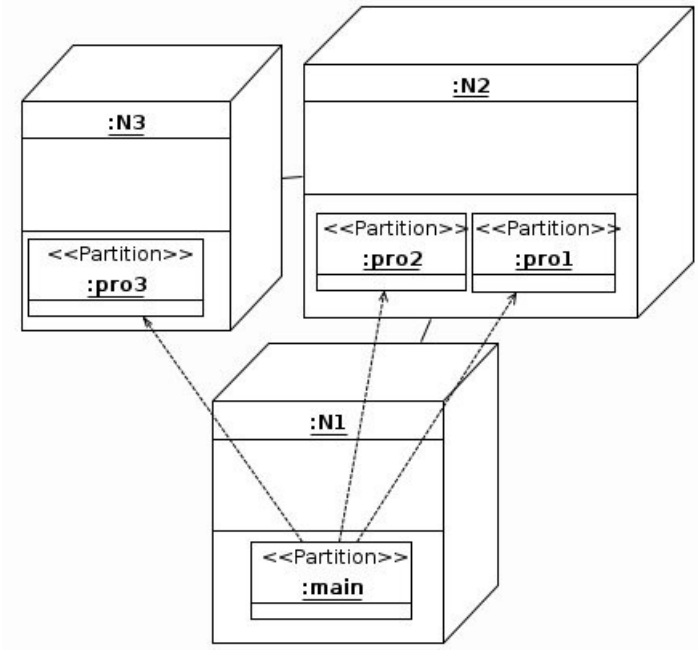

Figure 7.4: Output of the weaving process seen from the Deployment View

8 The Concurrency View

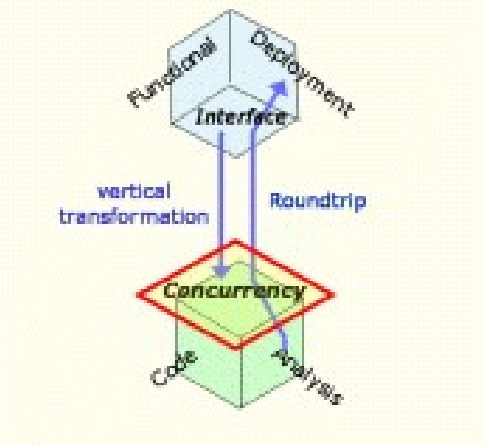

Figure 8.1: The Concurrency View

The Concurrency View describes the concurrent infrastructure of the system realized in compliance with the RCM [BDV03, VZd05]. In ASSERT-related publications the Concurrency View is also referred to as "Timing View" (cf., e.g., [CEPV06]). The Concurrency View describes the components that realize the provided interfaces specified by the designer in the Interface View. Those components are termed VMLC, which we have introduced in section 2. VMLC are the sole run-time entities supported by the RCM Virtual Machine and thus the sole entities that may populate a legal ASSERT system. The VMLC are typed and their legal types are as follows.

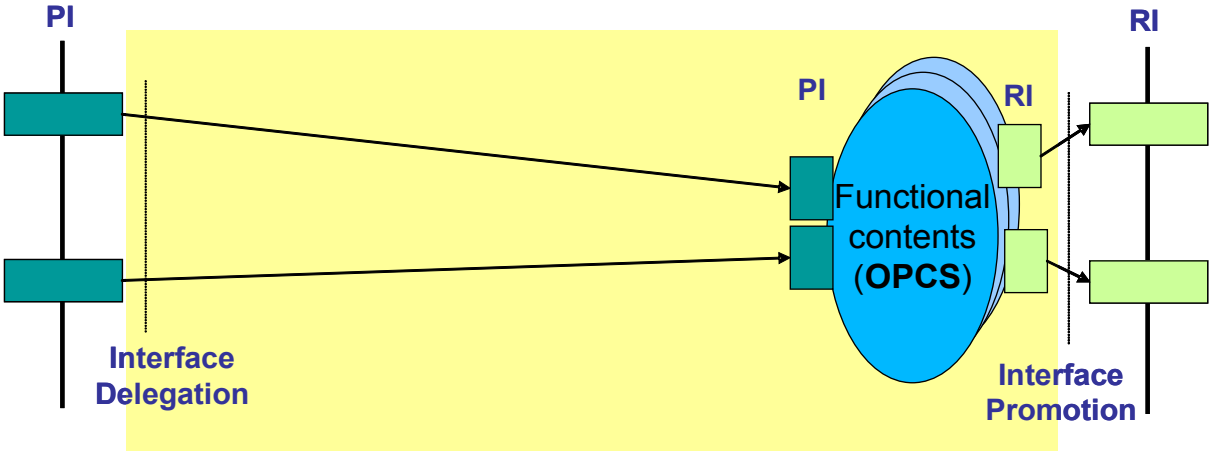

Figure 8.2: Passive VMLC

Passive VMLC (see figure 8.2): It realizes the services specified in its provided interface by warranting no access protection to its local functional state. The execution of the PI services of a passive VMLC is performed by the caller. The passive VMLC is defined by: PI; RI (which may be empty); OPCS.

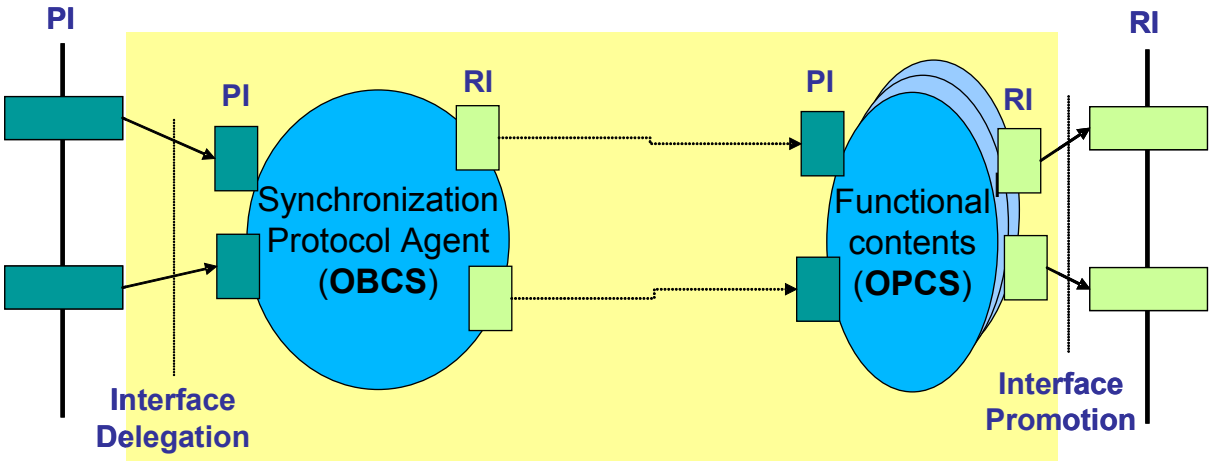

Figure 8.3: Protected VMLC

Protected VMLC (see figure 8.3): It realizes the services specified in its provided interface by warranting access protection (whether mutually exclusive or transactional) to its local functional state. The execution of the PI services of a protected VMLC is performed by the caller. The protected VMLC is defined by: PI; RI (which may be empty); OBCS (which provides access protection); OPCS.

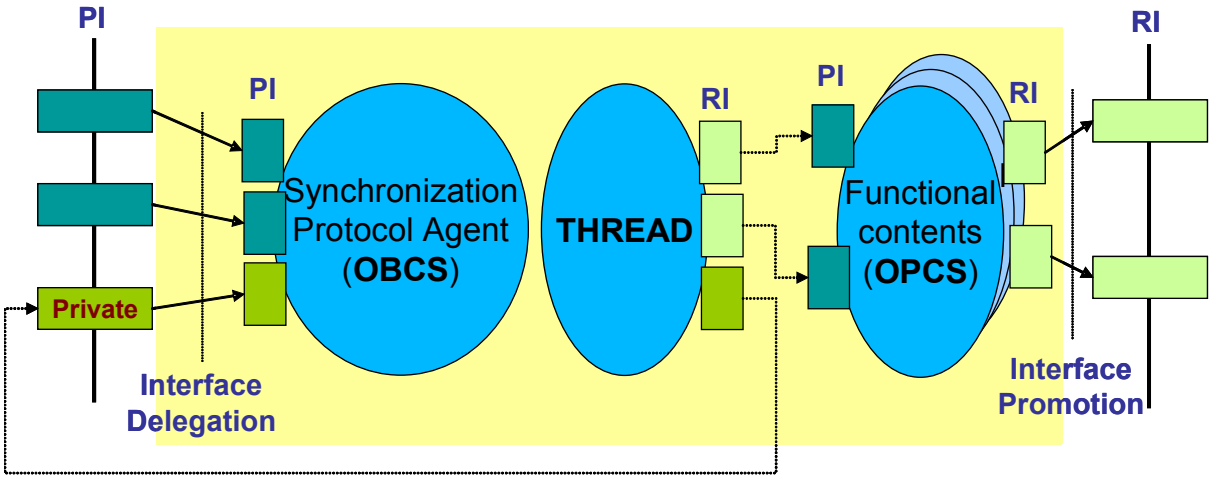

Figure 8.4: Threaded VMLC

Threaded VMLC (see figure 8.4): It realizes the services specified in its provided interface and has an internal thread of control execute them on behalf of the caller. The thread of control is released by either the arrival of an external invocation of a PI service or, with a fixed rate, by the system clock. In the former case the threaded VMLC is called sporadic (and a minimum inter-arrival time attribute is to be specified in order for the RCM VM to enforce it at run time) whereas cyclic in the latter case.

The PI attributed to a VMLC by model transformation from a source APLC determines the RI exhibited by that same VMLC.

8.1 Model Transformation

The model transformation supported by the HRT-UML/RCM tool (which is also known as *vertical*

transformation in the ASSERT-related literature) transforms the PIM in the PSM. It is a fully automated process that takes individual APLC and generates all the VMLC and the interconnections between them which are necessary to realize the source APLC.

Every APLC is transformed independently from other APLC. Elementary interfaces are grouped and every single group is transformed into a single VMLC following a set of production and semantic rules (the operation of which is briefly illustrated in section 3).

The model transformation preserves the interconnection between APLC so that some of the VMLC which result from the transformation of the interconnected APLC are themselves interconnected. Port clusters, which we have discussed in the Functional View where they have been introduced to simplify the design space, do not add any other semantics to the system model than visibility attributes. Since every elementary PI inherits the visibility attribute of the port cluster it belongs to then model transformation only considers the visibility attribute of port clusters. The model transformation engines embedded by HRT-UML/RCM are realized on the basis of a mathematical formalization which provably guarantees that no semantics distortion may occur in the PIM to PSM transformation.

Model transformation starts from the output of the Weaving stage of the HRT-UML/RCM development process. In the present version of the tutorial, we omit the description of the transformation rules and their mechanics; we give instead a single example of application, on the Partitioned Toy Example (see Appendix 1) as well as on the Distributed Toy Example (see Appendix 2).

9 Analysis and Round-Trip Engineering

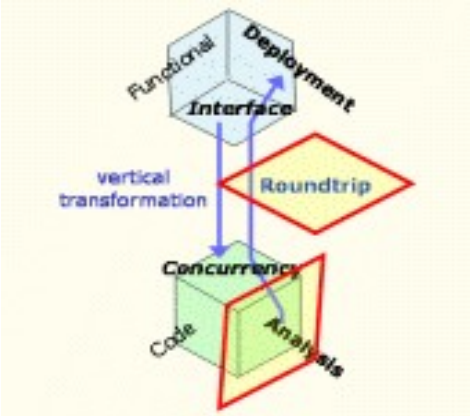

Figure 9.1: Analysis and Round-Trip Engineering

Static analysis: it statically determines whether the current implementation of the system is capable of meeting the space, time and communication requirements (e.g. memory space, bandwidth, deadlines) as set in the model specification when executed on the target platform.

Feasibility analysis: the classical forms of feasibility analysis concern timeliness and check whether the threads comprised in the system can execute within the stipulated deadlines.

Sensitivity analysis: it determines the margins by which the timing parameters of the model can be changed while keeping the system feasible or to make the system feasible.

In the current release of the HRT-UML/RCM tool-set feasibility analysis and sensitivity analysis are performed by MAST+, an enhancement of the MAST analysis utility developed at the University of Cantabria, Spain [UCA04]. In order to integrate MAST+ in the HRT-UML/RCM tool-set, we had to transform the Concurrency View to a model view which could be accepted by MAST+ (in fact by plain MAST since the MAST+ upgrade did not modify the input/output components of the original utility): what happens in practice is that a further instance of model transformation takes place from the RCM metamodel (which underpins HRT-UML/RCM) to the MAST+ metamodel and backwards.

The Analysis View is the place where the system model is described in terms of the MAST+ formalism. Although the model represented in the Analysis View need not be transformed into source code targeted for any particular platform, the information used to build it is platform-specific, and thus the Analysis View is part of PSM. The Analysis view abstracts away the details which are not relevant to its intent and purpose. For example, the system model is reduced to the instance level. Accordingly, the results of the analysis are reported back to the individual instances of the system, in every applicable view.

Once the relevant information is imported from the Concurrency View and the MAST+ model is constructed, the analysis is performed, and the Analysis View is decorated with its results and the new information automatically propagates throughout the entire model, thereby decorating the Concurrency, Interface and Deployment Views. This back propagation is part of the essential mechanisms of round-trip engineering. Some results of the analysis are placed back on individual ports and on the port clusters which are attached to VMLC instances (and then to the originating

APLC); others are attached to partitions and logical connections. In the following we enumerate a few examples of the information reported back from the Analysis View:

- For a cyclic task (as in the PRO threaded VMLC): maximum blocking time, maximum feasible WCET, minimum feasible period, OBCS ceiling, priority, task ceiling, worst-case response time
- For a sporadic task (for the Dispatcher threaded VMLC that realizes the Dispatch operation of the TMTC APLC): maximum blocking time, maximum feasible WCET, minimum feasible inter-arrival time, minimum inter-arrival time, OBCS ceiling priority, thread priority, worst-case response time
- For a protected resource (e.g. POS): ceiling priority
- For a passive resource (e.g. GNC): no return information.

9.1 The Partitioned Toy Example: Round Trip

In its current release MAST+ does not fully support distributed analysis as yet (but work is in progress to incorporate this feature). Accordingly this Tutorial cannot show and discuss the proceedings of the analysis of the Distributed Toy Example. We shall therefore limit ourselves to presenting the analysis of the Partitioned Toy Example. The Tutorial will be completed with the missing information before the end of the project.

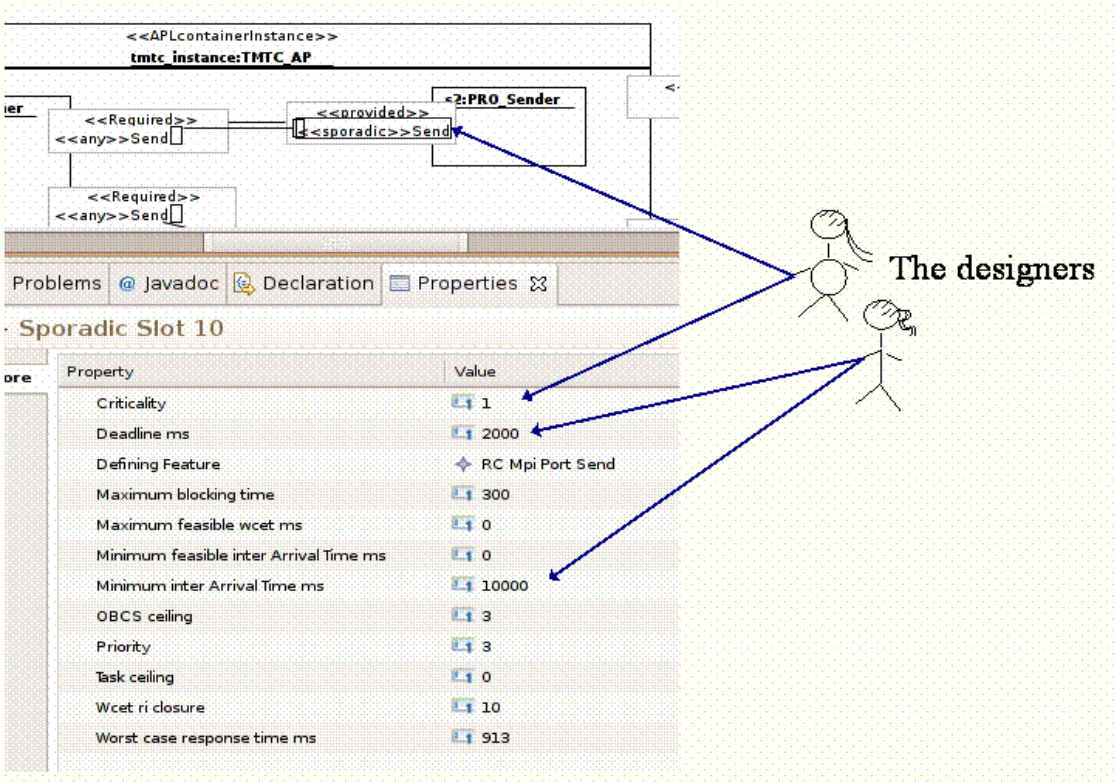

Figure 9.2: Sample of results from the feasibility analysis

Figure 9.2 shows the results of the analysis performed on the TMTC APLC of the Partitioned Toy

Example. The top of the figure shows an instance of TMTC APLC, whereas the bottom part of the figure reports the results of the timing analysis performed on port Send (which is typed Sporadic). The designer has set those values pointed to by the side arrows: the criticality level to 1; the deadline to 2,000; and the minimum inter-arrival time between any two subsequent activations of the deferred operation Send, to 10,000 units of time. The other values represent the results of the analysis. A simplified description of the measured attributes follows:

Maximum blocking time: the maximum time during which a task T_i can be blocked because of the priority inversion incurred on application of the priority ceiling emulation protocol.

Maximum feasible WCET: threshold of the maximum duration that the execution of the provided operation may take including the cost of any *immediate* operations invoked by the RI exposed by that operation. The system is feasible (timely) as long as the time cost of that provided operation does not exceed the reported threshold.

Minimum feasible inter-arrival time: the minimum time span that must occur between any two subsequent activations of the thread that execute the designated set of deferred services.l

OBCS ceiling [priority]: attribute of the synchronization agent used by the RCM Virtual Machine for managing asynchronous communication and synchronization between threads.

Priority: attribute used for FPPS scheduling (Fixed-Priority Pre-emptive Scheduling). The value of this attribute is automatically set by the analysis tool, on account of the criticality attribute set by the designer on the corresponding APLC.

Task ceiling [priority]: attribute that reflects the preemption level assigned to threads of control subject to EDF (Earliest Deadline First) scheduling.

WCET RI closure: maximum duration that the execution of the provided operation may take including the cost of any immediate operations invoked by the RI exposed by that operation.

Worst-case response time: maximum time it may take for a thread to complete a job under worst-case activation conditions.

9.2 Ongoing work

At the time this report is being written some planned features of the MAST+ analysis tool are not completed as yet. In particular:

1. Analysis is not available for distributed systems.
2. Analysis of communication bandwidth and memory usage is not available as it needs information specified in the Data View, which is still under development at present.

10 The Code View

Figure 10.1: The Code View

Although the Concurrency View is a platform-specific model which fully specifies the system in terms of the run-time entities supported by the RCM Virtual Machine, some further transformation steps have to be taken to obtain code ready for deployment: first source code generation and then compilation. All of the source code generated from the Concurrency View is currently in Ada 2005. In this tutorial, we chose to consider the source code as yet another instance of a platform-specific view, at the same level as the Concurrency and the Analysis View. This conceptual decision places emphasis on the following facts:

- the generated code is not intended for the inspection and editing of the designer (only the classes that feed the Functional View are; cf. Section 10.1 Architecture)
- n an MDA approach, the more specialized models ought to *contain all the information of all the "higher level" models, but at a low level of subject-matter abstraction*[Mel05]; the Code View is no exception and thus it should be expected to contain all of the information specified by the designer in the PIM with no omission and/or distortion: if the transformation logic is considered trustworthy, the designer can satisfy themselves with the PIM
- once the code is generated from the Interface View (via the required extent of weaving with the Deployment View) and the system is ready for use, not much else should be required to the designer in the way of system validation. Since the run-time preservation of the temporal, spatial, communication (and, prospectively, dependability) properties stipulated in the model is guaranteed by the HRT-UML/RCM methodology by construction, the final product of the transformation process should not need to be qualified by observing operational behaviour in test runs. This vision however assumes that the code which proceeds from the Functional View has been subject to prior verification so that the designer (and the transformation process) can place justified trust on its ability to compute the correct values.

10.1 Architecture

The code is automatically generated in the Ada 2005 programming language in conformance to the Ravenscar Profile restrictions. The extent of automation is 100% for the Interface View and the Concurrency View, while the code from the Functional Model may be either generated, in part, for models produced with the ETH FWProfile or else incorporated in the form of prefabricated code

components which however need to be asserted to comply with the RCM restrictions and must also have an Interface View description of them.

Figure 10.2 shows a Threaded VMLC, The OPCS represents its functional code. The OBCS and the thread represent its structural, or concurrent, side. The thread and the OBCS can be factored out in artefacts (called "archetypes") reusable across all VMLC instances of that type.

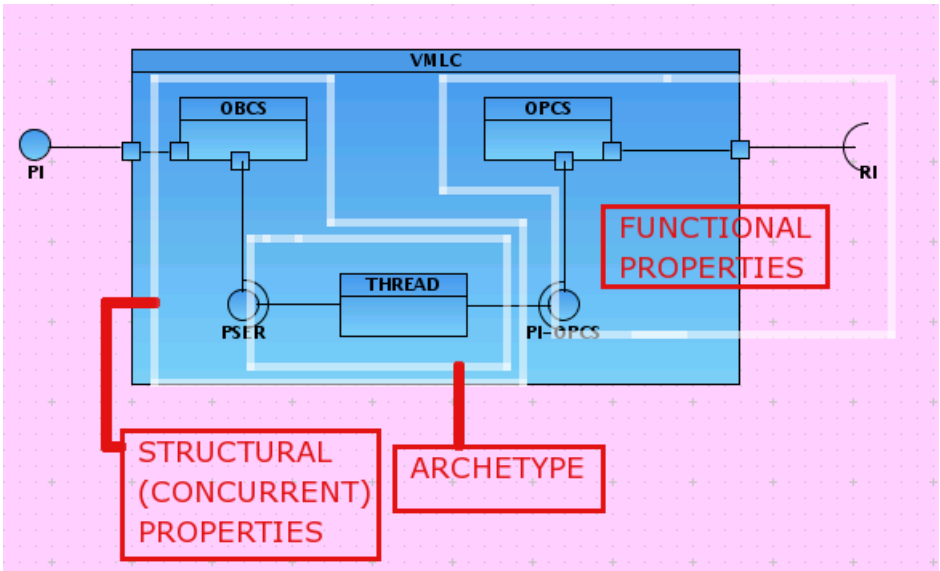

Figure 10.2: Breakdown of a threaded VMLC with archetype and OPCS

Consider the Distributed Toy Example discussed in the Concurrency View. Figure 10.3 below shows the architecture of the generated code for that example.

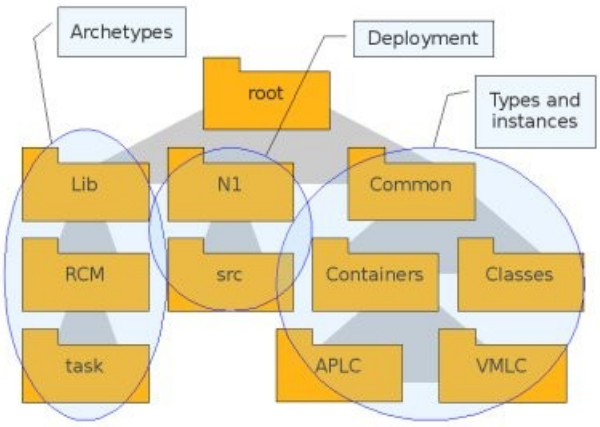

Figure 10.3: Architecture of the generated source code

The source code is (currently) placed under three directories respectively named: Common, Lib, N1 (for the 1st node in the system topology), as follows:

Common: it encloses the code which implements all the types specified in the model and all of their instances:

- the primitive types which have been specified in the Functional View, with the generated source code located in common/basic_types.ads;
- the Classes which have been specified in the Functional View, with the generated source code located in Common/Classes;
- the APLC and their instances as specified in the Interface View, with the generated source code located in Common/Containers/APLC;
- the VMLC and their instances as specified in the Concurrency View (produced off fully automated transformation, with the generated source code located in Common/Containers/VMLC.

In the absence of fully-automated code generation capabilities from the Functional View, the designer needs to implement all the methods specified in the Functional View by directly editing the source code in Common/Classes. The HRT-UML/RCM methodology forbids modifications in any other directories.

Lib: it encloses the prefabricated archetypes which the HRT-UML/RCM transformation and code generation engines need to build up the VMLC which populate the Concurrency View The generated code which is recurrent across implementations is factored into reusable basic entities like Request Descriptor, Thread and OBCS: VMLC are formed from suitable aggregations of those archetypal components.

N1: it represents the product of the automated generation of the application-specific parts of the software system (targeted for node N1 as identified in the Deployment View) and encloses

- identifiers for every software entity composing the system, i.e. for every computational node, for every partition, for every APLC and VMLC instance;
- all the deployment information, e.g., which APLC and VMLC instances are deployed in which partition;
- the initialization procedures, which bind instances to their designated node, following the specification of the Concurrency View.

10.2 The Partitioned Toy Example: Source Code

```
01   procedure Read_And_Write (This : in out POS) is
02      --Invoked required interfaces--
03      --+ This.Read : 1 invocation
04      --+ This.Write : 1 invocation
05      --+ C.Compute : 1 invocation
06      Res : Boolean := False;
07   begin
08      --#BlockStart number=3
09      --id=RCMoperation@eb77b0{file:/path_to_my_model/
10      --my_model.rcm#//@rootPackage/@ownedMember.11/@ownedOperation.2
11      -- User-defined code here --
12      declare
13         R : Integer := This.Read;
14      begin
15         This.C.Compute (R);
16         This.Write (R);
17      end;
18      --#BlockEnd number=3
19   end Read_And_Write;
```

Code fragment 10.1: The code for the Read_And_Write operation in common/classes/poss.adb

Code fragment 10.1 is a sample implementation for the method Read_And_Write found in Common/Classes/poss.adb. The generated code includes comments which help the user provide an implementation conformant with the functional specification. By way of example, the comments on lines 2-5 advise the designer that Read_And_Write was designed to invoke methods Read, Write and Compute exactly once. The other comments are needed by the generation facility to preserve the user-defined implementation when the code is regenerated.

The final source code is generated in such a way that the designer need not concern themselves about the infrastructure which realizes the concurrent behaviour to be obtained at run time. Three abstraction levels are crossed in the generation of the final source code: the Functional View (the most abstract); the Interface and the Concurrency Views (more specialized). Consider method Compute: it is specified by the designer in the Functional View as a provided method. In the Interface View, it is designated as a provided method of the GNC APLC. In the Concurrency View it appears as a provided method of the GNC VMLC. In all those views method Compute always provide one and the same signature, but with different semantic specializations. For this reason, the designer may content themselves with assuming that the command at line 15 in code fragment 10.1 should be a direct invocation of the Compute method specified in the Functional View while in actual fact, before getting there, the execution of it will first travel across the overriding layers added up by the code generation process to realize the concurrency semantics specified in the Interface View for that method.

The final version of this Tutorial (to be released by the end of the project) will contain, in annex, a representative sample of the source code automatically generated from the model for the examples discussed in this document.

11 Bibliography

[AN05] J. Arlow, I. Neustadt. *UML2 and the Unified Process* 2nd Ed., Addison-Wesley, 2005

[BDV03] A. Burns, B. Dobbing, T. Vardanega. Guide to the Use of the Ada Ravenscar Profile in High Integrity Systems. Techinal Report YCS-2003-348. University of York (UK), 2003. available at http://www.cs.york.ac.uk/ftpdr/reports/YCS-2003-348.pdf

[CEPV06] V. Cechticky, M. Egli, A. Pasetti, T. Vardanega. A UML2 Profile for Reusable and Verifiable Software Components for Real-Time Applications. In *Reuse of Off-The-Shelf Components (ICSR)* volume 4039 of LNCS Series, Springer-Verlag, 2006. Related ASSERT deliverable reports: **D4.2.4-1** (I1R2): *Software Framework Concept* (ETH, April 2005); **D4.2.2-1** (I1R0): *Software Building Block Adaptation Techniques* (ETH, September 2005).

[CV06] D. Cancila, T. Vardanega. AP-level Containers: A Survival Kit. ASSERT - Internal Release. 004033.DDHRT3-1.TN.1, December, 2006

[DL06] D. Lesens, P. Disseaux. Converation about the Toy example for transaction. Feb. 2006.

[GS88] J.B. Goodenough, L. Sha. The Priority Ceiling Protocol: A Method for Minimising the Blocking of High-Priority Tasks. *Ada Letters*, ACM, 8(7):35-38, 1988

[Les06] D. Lesens. Toy example for transactions. Feb. 2006

[OMG03] Object Management Group. MDA Guide Version 1.0.1. 2003. available at http://www.omg.org/docs/omg/03-06-01.pdf

[SG03] A. Silberschatz, P.B. Galvin. *Operating Systems*. Addison-Wesley. 2003

[TAN01] A.S. Tanenbaum. *Modern Operating Systems*. Prentice-Hall, 2003

[TAN03] A.S. Tanenbaum. *Computer Networks*. PH-PTR Pearson Education International, 2003

[UCA04] Universidad de Cantabria. Mast: Modeling and Analysis Suite for Real-Time Applications. http://mast.unican.es/

[Var06] T. Vardanega. A Property-Preserving Reuse-Geared Approach to Model-Driven Development. In *The 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications* pages 223-230. IEEE August 2006. Invited paper

[VZd05] T. Vardanega, J. Zamorano, J.A. de la Puente. On the Dynamic Semantics and the Timing Behaviour of Ravenscar Kernels. In *Real-Time Systems* Springer-Science, volume 29, pages 58-89, 2005]

11.1 Further Reading

11.1.1 On static scheduling analysis

E. Bini and G. Lipari. A methodology for designing hierarchical scheduling systems. *Journal of Embedded Computing*, Volume 1, Number 2, 2005, pages 257 - 269

R. I. Davis and A. Burns, Hierarchical Fixed Priority Pre-emptive Scheduling. In *Proc. of the 26th IEEE Real-Time Systems Symposium*, 2005, pages 389--398

E. Bini, M. Di Natale and G. Buttazzo. Sensitivity Analysis for Fixed-Priority Real-Time Systems. In *Proc. of the 18th Euromicro Conference on Real-Time Systems*, 2006.

Jose L. Lorente and J. Carlos Palencia. An EDF Hierarchical Scheduling Model for Bandwidth Servers. In *Proc. of the 12th International Conference on Embedded and Real-Time Computing Systems and Applications*, 2006

J.A. Pulido, S. Uruena, J. Zamorano, T. Vardanega and J.A. de la Puente. Hierarchical scheduling with Ada 2005. In *Reliable Software Technologies - Ada-Europe 2006*. Springer LNCS 4006, pages 1-12.

11.1.2 On modeling for analysis

M. Gonzalez Harbour, J.J. Gutierrez, J.C. Palencia and J.M. Drake. MAST: Modeling and Analysis Suite for Real-Time Applications. In *Proc. of the 13th Euromicro Conference on Real-Time Systems*, 2001

Marco Panunzio and Tullio Vardanega. A Metamodel-driven Process Featuring Advanced Model-based Timing Analysis. In *Reliable Software Technologies Ada-Europe 2007*, Springer LNCS 4498, pages 128-141

M. Bordin, T. Vardanega "Real-Time Java from an Automated Code Generation Perspective". In *Proc. of the 5th International Workshop on Java Technologies for Real-Time and Embedded Systems*, ACM Press, 2007.

12 Appendix 1: The Partitioned Toy Example: Concurrency View

This part of the tutorial describes the application of the model transformation to the Partitioned Toy Example, introduced in section 3 and recalled in Figure A1-1. Note that this example is different from the Distributed Toy Example which has been used in the previous sections. The main difference is that the system modelled in the Partitioned Toy Example is not distributed. The application of model transformation to the Distributed Toy Example is slightly more complex, and it is discussed separately in Appendix 2.

- Partition P_1: POS and GNC;
- Partition P_2: TMTC;
- Partition P_3: PRO.

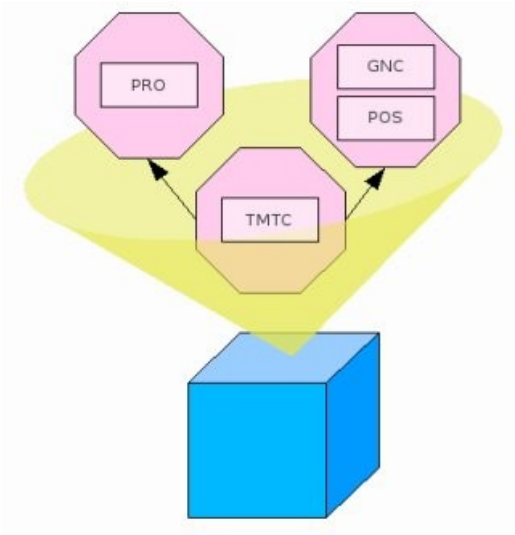

Figure A1-1: The Partitioned Toy Example

12.1 Partition P_1: POS, GNC

POS APLC: As shown in Figure 5.2, the POS APLC has three elementary PI which are all of type *immediate*:

- **Read** is *private* (in that in the Functional View it is not included in any public port cluster) and provides read-protected access rights to the functional state of POS.
- **Write** provides *mutual-exclusive access* to the functional state of POS.
- **Read_X_Write** is a composite operation (that is, one whose execution entails the invocation of at least one RI) with *transactional access* and thus holds mutually-exclusive access rights over

the functional state of POS during its entire execution.

POS also has an elementary RI typed to the Compute operation of the GNC APLC. That elementary RI is attached to the Read_X_Write operation and it is thus attributed to the VMLC product of the transformation that includes Read_X_Write in its PI. The model transformation of POS places all elementary PI in one and the same group since all of them use the same set of variables. This group generates a single *protected* VMLC (shown in Figure A1-2) which thus realizes the whole POS APLC. The type of the resulting VMLC is determined by the grammar rules presented in [Var06, CV06].

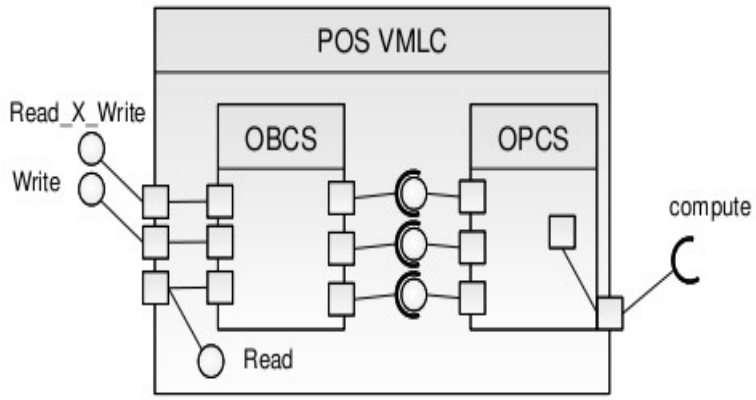

Figure A1-2: The POS VMLC

GNC APLC: The GNC APLC has two elementary PI as follows:

- GNC_op, which is *private* and *cyclic*
- Compute, which is *public* and *unprotected*

and one elementary RI which requires the Read_X_Write operation. Model transformation places those two elementary PI in two groups since their respective attributes make them incompatible.

Model transformation places those two elementary PI of in two distinct groups since their respective attributes make them incompatible. Each group (which in this case includes a single elementary PI) generates one VMLC. A cyclic threaded VMLC realizes the GNC op operation, whereas a passive VMLC realizes the Compute operation. Since the GNC op operation exhibits a requirement for the Read_X_Write RI, that elementary RI is attributed to the cyclic threaded VMLC which provides the GNC_op operation. The result of the transformation is shown in Figure A1-3.

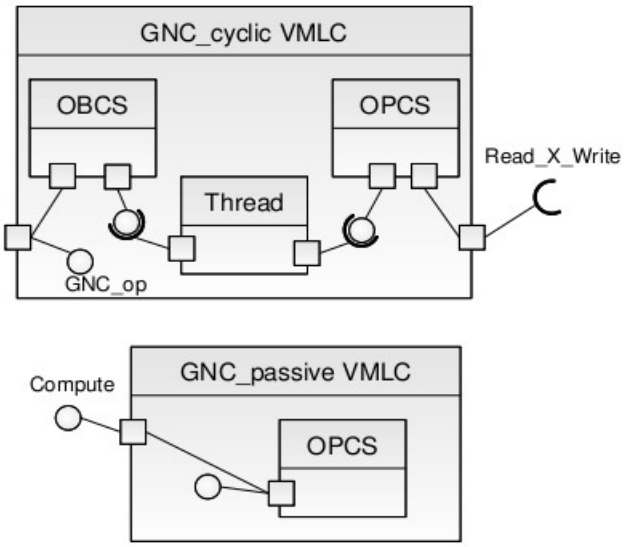

Figure A1-3: The GNC VMLC

12.2 Partition P_2: TMTC

The TMTC APLC has two elementary PI as follows:

- **Dispatch**, which is *public* and *sporadic*
- **Send**, which is *public* and *sporadic*, too

and four elementary RI, typed POS_Writer and PRO_Boost, which emanate from the implementation of the Send and Dispatch operations, respectively. The RCM grammar states that an interface group can contain only one deferred elementary PI which is set to denote the nominal operation. Model transformation of the TMTC APLC places each individual elementary PI in a single interface group, which maps to a sporadic threaded VMLC where they denote the nominal operation of the thread embedded in the VMLC. As we have seen earlier, the PI of every VMLC that results from the transformation determines the RI of that VMLC in accord with what specified in the Functional View (and reflected in the Interface View). Figure A1-4 shows the sporadic threaded VMLC that results from the model transformation of the TMTC APLC for the part which realizes the Dispatch operation.

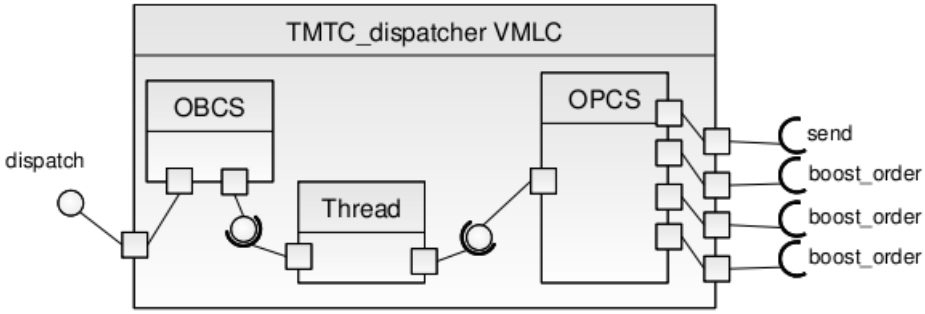

Figure A1-4: The threaded VMLC that realizes the Dispatch operation of the TMTC APLC

Figure A1-5 zooms in the sporadic threaded VMLC which realizes the Send operation.

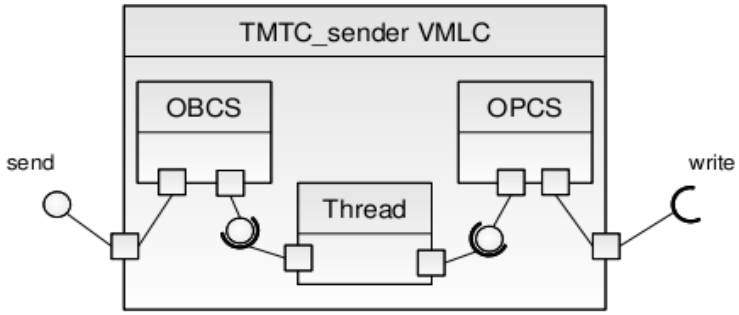

Figure A1-5: The threaded VMLC that realizes the Send operation of the TMTC APLC

12.3Partition P_3: PRO

The PRO APLC exhibits two elementary PI as follows:

- **Pro_OP**, which is *private* and *deferred*, tagged *nominal* and *cyclic*
- **Boost_Order**, which is *public* and *deferred*, tagged *modifier* to the nominal deferred behaviour of the component.

The RCM grammar states that an interface group can contain only one nominal elementary PI and that group must also contain all of the modifiers specified to it. Therefore, model transformation of the PRO APLC places all elementary PI in one and the same interface group where Boost_Order is the *modifier* of the nominal cyclic elementary PI PRO_op. The PRO APLC is thus realized by a single cyclic threaded VMLC as shown in Figure A1-6.

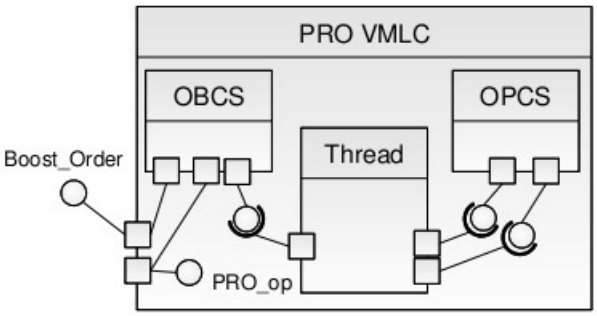

Figure A1-6: The PRO VMLC

13 Appendix 2: The Distributed Toy Example: Concurrency View

In this section, we apply the model transformation to the Distributed Toy Example.

13.1 Node N_1, Partition P_1: POS, GNC, TMTC, Stub

The generation of POS, GNC and TMTC has already been discussed in Appendix 1. Here we turn our attention only to the generation of the *stub* called for in the distributed transformation of the Toy Example Model.

The generation of a stub is required by the calling VMLC involved in a distributed communication and provided by the corresponding remote VMLC. In the case of the Distributed Toy Example the Boost_Order operation become distributed. The stub VMLC is created to be the access point to the network communication middleware: to limit the overhead induced on the timing and sizing behaviour of the system at run time, the PI of the stub VMLC are always *immediate*.

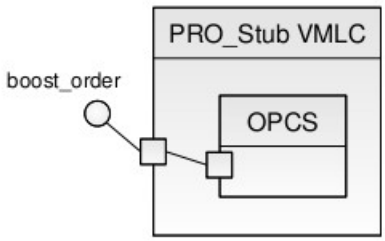

Figure A2-1: The stub VMLC for a PRO VMLC

13.2 Node N_2, Partitions P_2 and P_3: PRO instances, Skeleton

We have already discussed the generation of the PRO APLC. We now show the generation of its *skeleton* components.

Model transformation generates one partition with one VMLC that acts as a skeleton, that is to say, as the local proxy of the remote caller of the designated component. The partition that contains the skeleton VMLC is attributed the same criticality level as the partition of residence of the remote caller. Generating a distinct partition on the destination node eases the representation of the criticality attribute of the calling component on the remote node. Simple optimization permits to merge that partition with any other partition on the target node which has equivalent criticality attributes. In this particular case, model transformation generates only one skeleton partition since one is the remote calling partition. The skeleton VMLC is threaded and typed sporadic; it provides an elementary PI which is invoked by the network communication middleware, and an elementary RI connected to the target elementary PI of the callee.

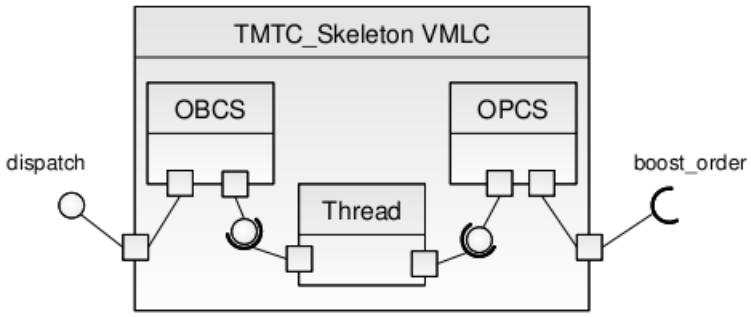

Figure A2-2: The skeleton VMLC for a TMTC VMLC

13.3Node N_3, Partition P_4: PRO instance, Skeleton

The treatment of this case produces a VMLC similar to the one discussed in Appendix 1 for the Partitioned Toy Example.

14 Appendix 3: Mutual Exclusion and Transactional Access

Mutually-exclusive access to a resource ensures that only one process at a time may be granted write-mode access to that resource while any other process requiring it be denied reading or writing access [Tan01]. (Conversely, read-only access permissions can be granted to multiple processes simultaneously.)

Example 1 (Mutually-exclusive access)

Consider a railway system that interconnects three stations: Venice, Vienna, Berlin. (Cf. figure) A passenger wants to buy a ticket from Venice to Vienna. When the counter staff contacted by the passenger commences the ticket issue procedure procedure, no other counter staff anywhere in the railway system may begin to issue tickets on the Venice-Vienna line until the ticket issue procedure in progress on that line completes. In this example the shared resource is the passenger seat on a single railway line, the exclusive right to which is represented by a ticket.

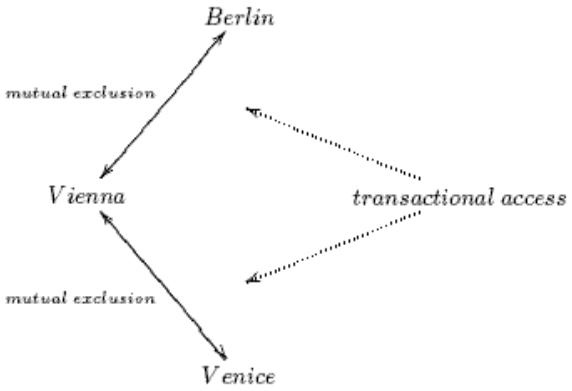

Figure A3-1: Mutual Exclusion and Transactional Access

Notice that if a passenger wants to buy a ticket from Venice to Berlin, the mutual exclusion access guaranteed alone could cause the following situation: the passenger may actually acquire a ticket from Venice to Vienna only to discover that all tickets from Vienna to Berlin (in the time window of interest) have already been sold out.

Transactional access to a resource ensures that only one process at a time may be granted access to that resource until completion of a designated set of operations; not all the operations in the set need to operate on one and the same the shared resource, as for example in set: read from resource; process value outside the resource; write to resource. Any other process that requires access to a resource locked with transactional access will be denied access until the resource has been released (and thus the entire set of operations has completed) [SG03]. It follows from this definition that a single transactional-access operation set may involve multiple mutually-exclusive-access resources and lock them all for the entire duration of the operation set.

Example 2 (Transactional Access)

Consider the system depicted in Figure A3-1. A passenger wants to buy a ticket from Venice to Berlin. When the counter staff contacted by the passenger commences the ticket issue procedure, no other counter staff in the system may issue tickets until the current procedure completes. In this example the transactional-access operation involves two mutually-exclusive-access resources and locking both of them in effect locks the entire system.

14.1Transactional Access in HRT-UML/RCM

At present the HRT-UML/RCM methodology provides partial support for node-local transactions.

In the Toy Example, the original problem specification requires the read-compute-update sequence to be performed while holding exclusive access rights on the POS resource. In HRT-UML/RCM, one way to obtain this behaviour is to define an additional operation, which we named Read_X_Write, whose only purpose is to execute the sequence of operations requiring transactional access on POS. Marking Read_X_Write as protected caters for access to POS to be locked during the execution of the three required operations. In fact, the ceiling assigned to the protected resource associated with Read_X_Write will reflect the ceiling of POS to ensure that, when the execution of Read_X_Write commences, no local competition could ever arise on access to POS. For the same reason, should POS require additional protected data resources for the execution of Read or Write, mutual exclusion access to those resources would automatically be acquired at the beginning of the execution of the transactional operation.

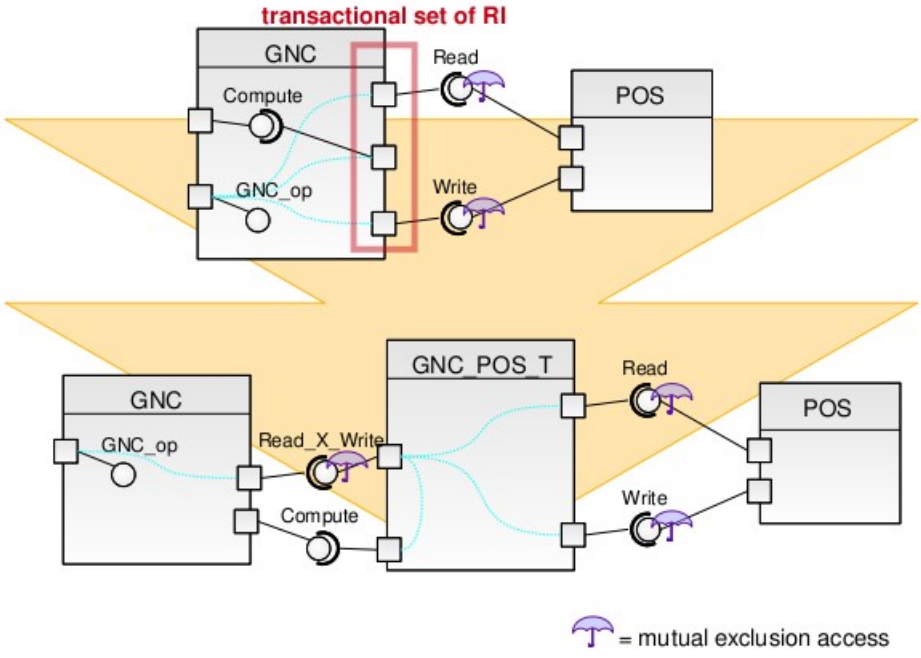

Figure A3-2: Providing transactional access using an additional protected PI

Figure A3-2 shows how the mechanism could be automated: in an ideal scenario, whenever a designer

wished to perform a transaction on a single resource, s/he would simply mark the operations required within that transaction as *transactional* RI, Read, Compute and Write in the example. The tool would then automatically and transparently copy the transactional operation GNC_op to a separate resource, marking it "protected". In Figure A3-2, this copy is named Read_X_Write, and the resource is named GNC_POS_T. The original APLC (GNC) is transformed to substitute the original RI of GNC_op with a single RI connected to the new PI (Read_X_Write). RI of GNC_POS_T are eventually connected to the PI the designer originally intended to use.

To optimize the solution, the new PI (Read_X_Write) could be placed into the called APLC (POS), in order to bypass access control structures (OBCS) of the called APLC. Otherwise, the copy could be placed into the caller APLC (GNC) in order to avoid additional dependencies between the callee (POS) and the caller (GNC).