Project IST-2004 004033

ASSERT

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

Instrument:	**IST [FP6-2004-IST-2 4.3.2.5]**
Thematic priority:	**Embedded Systems**
Deliverable:	**004033.DDHRT_UPM.DVRB.09.I1R2 D7.3-2 Consolidation of requirements and MW definition and coverage**

Work package:	7	
Due date of deliverable:	M33	
Submission date:	14/01/2008	
Start date of project:	5 September 2004	Duration: 3 years
Organization name of lead contractor for this deliverable:	UPM	
Issue Revision:	I1R2	

Project co-funded by the European Commission within the Sixth Framework Programme (2002-2007)		
Dissemination Level		
PU	Public	
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	**X**
CO	Confidential, only for members of the consortium (including the Commission Services))	

D7.3-2 Consolidation of requirements and MW definition and coverage

Distribution: All project
Prepared by: Juan A. de la Puente (UPM)
Jérôme Hugues, Laurent Pautet (ENST)
Marek Prochazka (SciSys)
Checked by: Tullio Vardanega (UPD)
Release type: Internal project release: 14/01/2008
Status: Final

Document Change Record

Issue Revision	Date	Affected Section/Paragraph/Page	Reason for Change/Brief Description of Change
I1R0	07/01/2008	All	First version.
I1R1	08/01/2008	All	Some typos fixed.
I1R2	14/01/2008	All	Final version after IPR.

Contents

Abstract

This document is focused on the basic aspects of dependability coverage of the ASSERT Virtual Machine. It contains an overview of the main features and architecture of the Virtual Machine, and an assessment of the failure coverage of its main components. For the real-time kernel component, a refined task model is defined, and an analysis of temporal faults is provided. Basic ideas on temporal and spatial partitioning are also exposed. Temporal faults and partitioning are also discussed from the point of view of the communications services. Basic strategies to limit the effect of faults in the middleware are also considered.

The main conclusion is that, although there is significant work to be done in the area of fault coverage in the ASSERT VM, the current design and implementation already offers significant support for detecting and correcting temporal faults, and for isolating real-time applications from temporal faults occurring in other applications.

Abbreviations

AADL	Architecture Analysis and Design Language
APLC	Application-level container
API	Application Programming Interface
ASS	Application Support Services (SOIS)
ASSERT	Automated proof based System and Software Engineering for Real-Time Applications
CCSDS	Consultative Committee for Space Data Systems
CMS	Communications Service (SOIS)
DDHRT	Dependability, Distribution and Hard Real-Time
GNARL	GNU Ada Run-time Library
GNULL	GNU Lower-level Library
GNULLI	GNU Lower-level Library Interface
GNU	GNU's Not UNIX
GPL	GNU General Public License
GPS	GNAT Programming Studio
HI	High-Integrity
HRI	Highly Reliable Infrastructure
HRT	Hard Real-Time
MAT	Minimum inter-Arrival Time (of a sporadic event)
MDA	Model-Driven Architecture
MDD	Model-Driven Development
MPC	Multiple Platforms Cooperation
MTS	Message Transfer Service (SOIS)
MW	Middleware
ORK	Open Ravenscar real-time Kernel
PIM	Platform-Independent Model
PSM	Platform-Specific Model
RCM	Ravenscar Computational Model
RTOS	Real-Time Operating System
SEMV	System Engineering Modelling and Verification
SOIS	Spacecraft Onboard Interface Services
TCOAS	Time-Critical On-board Application Services (SOIS)
UML	Unified Modelling Language
VM	Virtual Machine
VMLC	Virtual-Machine-Level Container
WCET	Worst-case execution time

Chapter 1

Introduction

1.1 Purpose

This document is aimed at discussing some basic aspects of dependability coverage of the ASSERT Virtual Machine. It reflects the work of the DDHRT cluster members in WP7, and its main contribution is to make a critical assessment of the VM distribution model with respect to dependability issues.

1.2 Inputs for the study

This document is built upon previous work already undertaken by the DDHRT and SEMV clusters on the definition of the ASSERT virtual machine and on modelling of safety properties. The following documents contain relevant material on this subject:

[R1] D3.3.2-2. *Virtual Machine Architecture Definition.* ASSERT document 004033.DDHRT ENST.DVRB.05.I1R1

[R2] D3.3.2-3. *Virtual Machine Components Specification.* ASSERT document 004033.DDHRT_ENST.DVRB.06.I1R1

[R3] D3.3.3-1. *Refined Virtual Machine.* ASSERT document 004033.DDHRT_UPM.DVRB.08.I1R0

[R4] D3.3.3-2. *Refined VM Demonstrator Experience Report.* ASSERT document 004033.DDHRT_UPM.DVRB.07.I1R0

[R5] D7.2-1. *System Architecture Safety Modelling and Verification Report: Functional and Architecture Levels.* ASSERT document 004033.SEMV_ONERA.DVRB.01.I1R1.

[R6] D7.2-2. *System Architecture Safety Modelling and Verification Report: Functional, Architecture, and Service Levels.* ASSERT document 004033.SEMV_ONERA.DVRB.02.I1R1.

[R7] M. Turin, *Dependability requirements and verification.* Technical Note, GTI6-1030-2. ESA, I2R1.

Other documents of interest are listed in the *references* section at the end of the document.

1.3 The ASSERT Virtual Machine

The ASSERT Virtual Machine (VM) is the execution platform for ASSERT applications. It is intended to enforce the run-time behaviour that the lower-level code entities (called *VM-level containers*) require in order to guarantee their specified properties. It supports the Ravenscar Computational Model (RCM), which in turn is based on the Ada Ravenscar Profile [ISO05].

The concept of the ASSERT VM entails the following characteristics:

1. It only accepts and supports "legal" entities, i.e. VM-level containers.
2. It provides run-time services that support concurrent execution of real-time threads.
3. It is bound to a compilation system that only produces "legal" entities, i.e. code generated from instances of VM-level containers.
4. It realizes a concurrent computational model which is amenable to static analysis.

The document *D3.3.2-2 — Virtual machine architecture definition* [R1] describes in detail the architecture of the VM and its constituent components. The document *D3.3.2-3 — Virtual machine components specification* [R2] contains a detailed description of the computational model and the services provided by the VM.

1.4 Structure of the document

The rest of this document is organized as follows: chapter 2 contains an overview of the VM architecture. Chapter 3 discusses the VM architecture with respects to dependability issues. Finally, chapter 4 contains some final conclusions and remarks.

Chapter 2

Overview of the ASSERT Virtual Machine

2.1 Introduction

The ASSERT Virtual Machine (VM) is the execution platform on which ASSERT applications run. It is based on the Ravenscar Computational Model (RCM) [ISO05], and only accepts software entities which are valid under this model. To this purpose, the ASSERT VM provides run-time services that support the basic elements of the model:

- A static set of threads, with a single activation point, scheduled according to a fixed-priority pre-emptive scheme. Periodic and sporadic activation patterns are admissible.
- A static set of shared data objects, protected by mutual exclusion synchronization. Protected objects may have at most a synchronized operation with a boolean guard, which is executed only when the guard is true. No more than one thread can be suspended waiting for a guard to become true. Access to protected objects is performed according to an immediate priority ceiling protocol [SRL90].

The ASSERT VM also supports predictable communication and distribution on a network of computer nodes. In order to properly implement these functions, it has been designed with a layered architecture including the following subsystems (figure 2.1):

- A real-time kernel providing concurrent execution, scheduling, synchronization, and timing services. The kernel directly supports the Ada Ravenscar profile [ISO07, D.13.1], which is the standard implementation of the RCM in the Ada 2005 programming language.
- A network access layer containing communication drivers.
- A communications subsystem providing a SOIS MTS transport service.
- A middleware layer providing distribution transparency and higher-level services for distributed applications.

The rest of this chapter contains a description of the main components of the ASSERT Virtual Machine.

2.2 Real-Time Kernel

The VM real-time kernel is an evolved version of ORK[dlPRZ00], a small, high performance real-time kernel for LEON processors [Gai05] that provides restricted tasking support as defined by the Ada Ravenscar profile. The

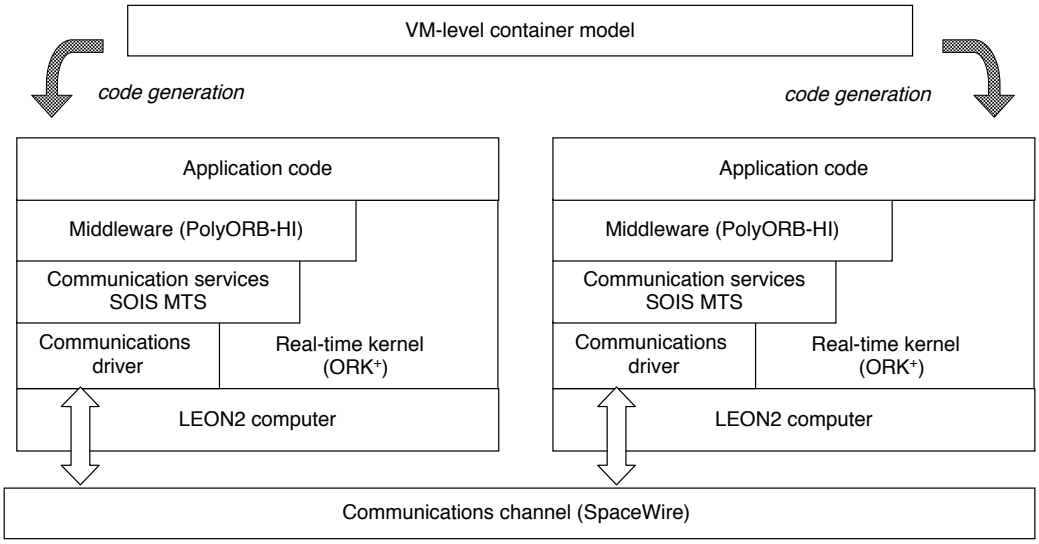

Figure 2.1: ASSERT Virtual Machine Architecture.

current version, ORK+[UPRZ07], includes support for the new Ada 2005 timing features, and is integrated with the GNAT GPL 2007 compiler. The full package including the compiler and the kernel is called GNAT for LEON. The integrated kernel is also known as the GNAT for LEON Bare Board Kernel.

The kernel provides the following functionality:

Thread management: Only static Ada tasks declared at the library level are accepted. Consequently, the associated data structures can use only statically allocated memory.

Thread scheduling: The scheduling of threads is performed according to the `FIFO within priorities` and the `ceiling locking` methods [ISO07, D2.3].

Thread synchronization: Only Ravenscar-compliant synchronization is supported. The only synchronization operations provided by the kernel are unconditional suspend and resume.

Time management: Time is represented as a 64-bit integer number of ticks. A tick is a real-time interval during which the clock value remains constant which is four times the period of the LEON2 input clock. The tick is 97.56 ns for a GR-XC3S-1500 LEON development board with a 41 MHz clock.

The LEON2 *timer 2* is used in periodical mode to maintain the least significant part of the clock whereas the most significant part is updated by the timer interrupt service routine. As a result, the interrupt handling overload is kept low, as the clock interrupt period is only 1.6368 s for a GR-XC3S-1500 LEON development board with a 41 MHz clock.

Execution time monitoring: Every thread has an execution-time clock that computes the amount of CPU time it has consumed. Execution-time is also represented as a 64-bit integer number of ticks. The execution-time clock of the running thread takes into account the elapsed time from the last context switch.

A thread can have at most one execution-time timer associated to it. The timer can be armed to expire at an absolute value of execution-time clock or after some execution-time interval. In this way, an execution-time budget can be set for the execution-time of code segments. When the timer expires, an user-defined protected procedure handler is executed which can take corrective actions.

Group budgets: Groups of threads can have one group execution-time budget associated to them. This feature is supported on top of execution time monitoring of single threads.

Absolute alarms: Absolute alarms can be set with a precision of one tick by absolute delays, execution-time timer of the running thread or timing events. Absolute alarms are implemented in a way that provides a high precision with a low overload.

Interrupt handling: Interrupt handlers are called directly from the hardware, and are executed as if they were directly invoked by the interrupted thread. However, a preamble is executed before the protected procedure handler and an epilogue after it. It must be noticed that the Ravenscar profile makes possible this simple and efficient implementation.

GNATforLEON also supports nesting of interrupt routines in order to minimize the latency of high priority interrupts. An important implication of this interrupt model is that if the active priority of a running thread is equal to or greater than that of an interrupt, the interrupt will not be immediately acknowledged by the processor. In such cases the interrupt remains pending until the active priority of the running task becomes lower than the priority of the interrupt.

Configuration: The kernel has configuration parameters that enable it to be tailored to different applications. The configurable parameters are: size of the interrupt stack, frequency of the processor clock, available memory space, and range of priorities. The VM V3 distribution is configured for a GR-XC3S-1500 LEON development board.

2.3 Communication services

The key part of the VM communications layer is the Message Transfer Service (MTS). MTS is a process-to-process transfer service which is one of the CCSDS Spacecraft On-board Interface Services (SOIS) [CCS07a]. The SOIS Communications Architecture is split into a bus-specific Subnetworking Layer and a higher Application Support Layer, formerly called the Time-Critical On-board Application Services (TCOAS) layer. MTS is one of the services of the SOIS Application Support layer. MTS supports peer-to-peer and publish-subscribe of discrete messages between on-board applications and/or higher-layer services. Messages have priorities assigned to them and are delivered with the priority-driven ordering with FIFO ordering within a single priority level. In terms of functionality, MTS provides the following features:

Peer-to-peer communication: MTS provides API for peer-to-peer communication between application tasks. Any task can open a channel to another task and communicate directly with that task. Once a channel is established between two tasks, they both can play the role sender or receiver. In other words, channels are bi-directional and could be established between any two tasks in the system.

Local message delivery: If both the sender and receiver tasks are located on the same network node then an RTOS-specific message queues (e.g. POSIX queues) are used to deliver MTS messages locally. For RTOS with single address space which does not provide inter-process communication using priority message queues, our MTS implementation provides its own message queue library.

Remote message delivery: If the sender and receiver tasks are located on different network nodes then MTS uses underlying network stack to deliver message over network. So called MTS Daemon is used as a proxy which receives all messages directed to a node and uses local priority message queues to deliver messages to ultimate destination tasks (see figure 2.2). It is transparent from the API point of view whether a message is sent locally or via network.

Different programming models: MTS support both synchronous request/response call or request call with a response callback, blocking and/or non-blocking communication primitives.

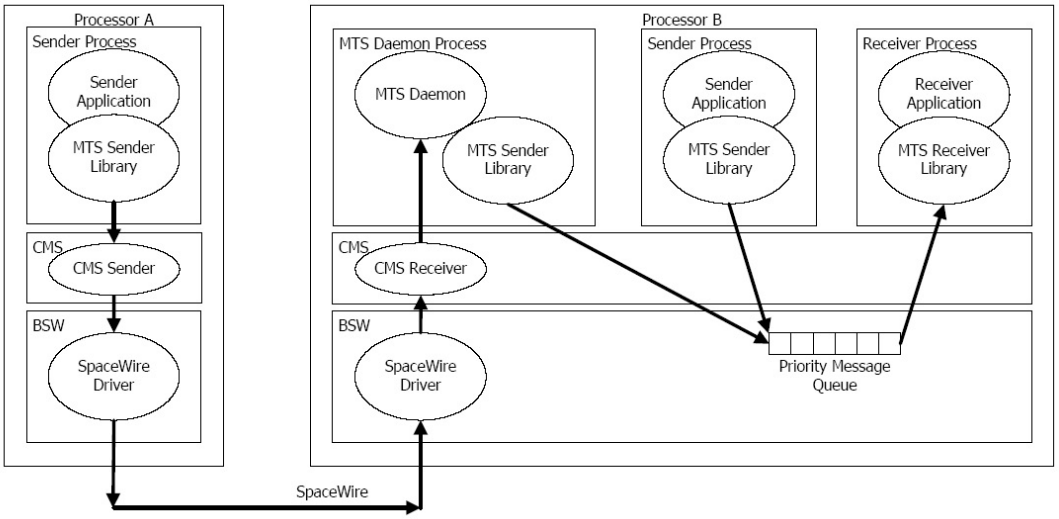

Figure 2.2: MTS architecture with CMS, SpaceWire driver and BSW

The Communications Service (CMS) is an implementation of a proposed SpaceWire mapping of the SOIS Subnetworking Layer's Packet Service [CCS07b]. It implements the Packet Service's Best Effort and Assured Classes of Service. It supports the delivery of packets between SpaceWire hosts in priority and with priority FIFO order (acknowledged or unacknowledged). It provides segmentation of user data where the exceeded a configured Maximum Transfer Unit size (currently not used within ASSERT) and enforces maximum communication bandwidth for communicating applications by limiting the amount of sent data per time window.

The communications layer also includes a SpaceWire driver interfacing to a simulator of the SMCS332 SpaceWire driver chip, re-used from the Software Validation facility for the SMOS mission instrument application software. The simulator is implemented as an I/O module of the TSIM Professional for Leon2 simulator and uses TCP/IP socket communication between TSIM instances to simulate SpaceWire links.

2.4 Distribution middleware

Distribution features are provided through the use of the PolyORB-HI middleware [VHPK04]. This light framework provides a reduced set of primitives to support transparent distribution.

PolyORB-HI is fully compliant with both the Ravenscar profile and the AADL runtime semantics. In addition, it compiles under stringent restrictions mandatory for building High-Integrity systems, as detailed in the document *D3.3.2-2. Virtual Machine Architecture Definition* [R1]:

Distribution model. The model used is based on asynchronous oneway requests. This model is compatible with the programming model enforced by the APLC modelling framework.

Distribution transparency. Transparency is achieved through the encapsulation mechanism provided by the AP-level container. The user-code interact through ports to send or receive data. From the model, code generators deduce the required mechanisms to route date from one APLC to another. This mechanism efficiently provides for distribution transparency.

Supported protocol. Distribution mechanisms can support multiple protocols, depending on the deployment information provided. In the context of ASSERT, PolyORB-HI can use MTS. In addition, for prototyping purpose,

it can support also Ethernet. The addition of new protocols is supported in the framework.

Restrictions: PolyORB-HI limits the semantics of the interactions to match requirements from High-Integrity systems. To match requirements for bounded memory, deterministic communication and concurrency patterns:

1. only bounded data types are supported,
2. all interfaces shall be known and deduced from the model (no dynamic invocation),
3. all communication channels shall be described in the deployment view of the model.

Chapter 3

Dependability aspects of the ASSERT Virtual Machine

3.1 Introduction

As explained in chapter 2, the ASSERT Virtual Machine is the execution platform on which ASSERT applications run. The functional code is embedded in so-called *application-level containers* (APLC), which are used to build a platform-independent model (PIM). This model is transformed, using deployment information, into a platform-specific model (PSM), made up of *virtual machine-level containers* (VMLC). VMLC rely on virtual machine services to implement non-functional properties such as timing, scheduling, synchronization, communication, and distribution. Therefore, the VM subsystems play an important role in the dependability of the whole system.

The current implementation of the VM [R3,R4] includes some dependability-related mechanisms, such as execution-time monitoring, and opens the way to further developments, such as spatial isolation and distributed consensus mechanisms. Previous work on dependability analysis within the ASSERT project [R6,R7], based on a preliminary specification of the VM, set the basis for analysing some dependability aspects, especially those related to temporal faults, and suggests some improvements, some of which are already implemented in the refined VM. The rest of this chapter is intended to contribute to this work by providing updated information and a preliminary assessment about dependability aspects of the ASSERT VM.

3.2 Real-time kernel

3.2.1 Task model

The main function of the real-time kernel is the dispatching of real-time tasks. Tasks are dispatched using a fixed-priority pre-emptive scheduling method, as specified by the Ravenscar Computation Model, which is described in detail in the document D3.3.2-3 [R2]. Some important characteristics of the dispatching model are:

- Tasks have a single activation point and cannot be suspended until the end of their execution cycle.
- Tasks may be pre-empted by higher priority tasks
- Tasks may have to wait for exclusive access to shared resources (protected objects).
- While accessing a shared resource (protected object), a task inherits its ceiling priority (i.e., the highest priority of the tasks which may use the resource).

Since tasks cannot be suspended during their execution, input and output must be performed without waiting. This can be accomplished by reading protected data using non-suspending operations, with a flag indicating successful operation. Alternatively, a task can be activated sporadically upon the reception of some input data.

The automaton in figure 3.1 summarizes the scheduling model. Notice that when running on a single processor (as it is assumed in the RCM), ceiling priority inheritance by itself guarantees mutual exclusion and there is no need for a separate waiting state when a task has to wait for another task to exit on a protected object operation.

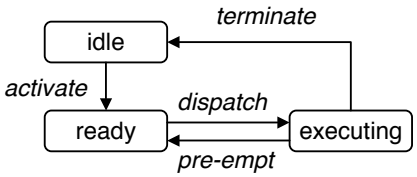

Figure 3.1: Task states.

This task model has some differences with the one that was used in R6. The most important one is that once activated a task can only be executing or pre-empted by a higher priority task.

3.2.2 Time management

The kernel provides timing services which can be used to enforce timing constraints on tasks:

- A *real-time clock* can be used to read the current time.
- An *absolute delay* (delay_until) can be used to enforce periodic execution or minimum inter-arrival time in sporadic activation.
- *Timing events* provide alternative ways to enforce periodic execution or minimum inter-arrival time. They can also be used to detect deadline violations.
- *Execution-time timers* can be used to detect WCET budget violations in a single task.
- *Group budgets* can be used to detect violations of global WCET budgets for a group of tasks.

References [PUZ+06], [PUZd07], [ZdlPHV07], and [PdlPB+07] contain more details on how the above mechanisms can be effectively used to enforce timing constraints.

3.2.3 Coverage of temporal faults

The main temporal requirements for real-time tasks are:

- Periodic activation, for periodic tasks;
- minimum inter-arrival time (MAT), for sporadic tasks;
- deadline, relative to the activation time.

Therefore, the following kinds of faults can be considered:

- **Periodic activation failure**: if a periodic task is activated more or less often than specified, the task may fail to accomplish its expected behaviour. For example, a control algorithm parameters may rely on a variable being sampled at a specified rate, and the algorithm may yield a wrong control action if the sampling period is not the same as required.

  Periodic activation may fail for several reasons:

  – Failure of the timer hardware used to implement delays;

  – incorrect operation of the kernel software.

  These faults are not detected or handled by the current implementation of the VM kernel. The real-time clock could be used to detect incorrect periodic activation, assuming that the clock hardware failures are independent of the timer hardware failures, which is likely not to be the case. For the moment we shall ignore these kind of fault as it is highly improbable, provided that the kernel software and the timer and clock hardware are reliable enough.

- **Sporadic activation failure**: A sporadic task being activated more often than specified may result in processor overloads, possibly leading to other tasks missing their deadlines. Such a kind of failure usually comes from an external event occurring more often than expected.

  The VM kernel provides two mechanisms for dealing with this kind of fault:

  – An absolute delay can be used to enforce minimal separation between task activations at run time;

  – a timing event can be used to detect inter-arrival time violations.

- **Deadline miss**: A task not fulfilling its deadline may result in incorrect operation of the tasks, or even of other tasks as it may also involve processor overloads or higher interference on lower-priority tasks.

  The task model implemented by the kernel implies that deadline misses can be caused by:

  – Execution time of the faulty task exceeding its budgeted WCET (WCET overrun);

  – higher interference from a higher-priority task using more processor time than expected.

  The second situation may in turn be caused by some higher priority tasks incurring WCET overloads or activation failures. The latter kind of faults has already been dealt with, and therefore the only remaining faults to be considered are WCET faults.

  The VM kernel provides some mechanisms that can be used to detect deadline misses:

  – Timing events can be used to directly detect deadline violations;

  – Execution-time timer can be used to monitor the use of processor time by tasks and therefore detect WCET overruns.

In summary, all the mentioned kinds of temporal faults can be detected using kernel mechanisms, except for periodic activation failures which are directly originated by real-time clock failures. See reference [PUZd07] for details and examples on how to use the above mentioned kernel mechanisms and possible fault handling patterns.

3.2.4 Temporal partitioning

Temporal partitioning is a technique for preventing temporal failures to be propagated between tasks with different criticality levels. Basically, it consists in allocating groups of tasks with the same criticality levels to *partitions*, and implementing isolation mechanisms for preventing temporal faults to be propagated from one partition to another. The temporal isolation mechanisms must ensure in particular that low-criticality partitions do not steal processor time from high-criticality ones.

There are different approaches to temporal isolation. The current version of the VM kernel supports temporal isolation by using hierarchical scheduling [PUZ+06]. However, there is still little experience on using this technique in real applications. See UPM's technical notes *004033.DDHRT_UPM.TN.3* and *004033.DDHRT_UPM.TN.7*[1] for more details.

3.2.5 Coverage of spatial faults and spatial isolation

Spatial faults are caused by a task invading the storage space allocated to another partition. Implementing spatial isolation requires some hardware support which is not available on current LEON processors. Consequently, the current version of the ASSERT VM kernel does not include any mechanisms supporting spatial isolation, nor even spatial fault detection mechanisms. Technical note *004033.DDHRT_UPM.TN.8* describes some alternative techniques that can be used to provide spatial isolation in future developments.

3.3 Communications services

The main function of the communications stack is message exchange between tasks. The message exchange is peer-to-peer and tasks could either execute on the same processor or different processors.

3.3.1 Message Transfer Service

The MTS ensures priority ordering of messages with FIFO ordering within a single priority level. More precisely, if in the destination message queue is a message with a lower priority, then newly incoming message with higher priority is inserted into the message queue in front of the other message. If two messages have the same priority then newly incoming message is inserted into the message queue after the other message.

3.3.2 Coverage of communication faults

Document D7.2-2 [R6] defines four respective channel types based on *send* and *receive* operation semantics:

- Both *send* and *receive* operations blocking;
- Blocking *send* with non-blocking *receive*;
- Non-blocking *send* with non-blocking *receive*;
- Both *send* and *receive* non-blocking.

The current MTS implementation only supports non-blocking *send* operation and blocking *receive* operation (it gets blocked in case that no message is in the task's receive message queue). Given that we restrict ourselves only to the communication primitives implemented, the identified communication faults are as follows:

- **Sending a message failed**: Sending a message may fail due to a hardware problem with the SpaceWire chipset. Note that the current preferred configuration is Best-Effort Class of Service (i.e. unacknowledged), and therefore the success of the *send* operation does not rely on the ultimate packet delivery, exceeding the link worst case latency, destination task's receive message queue being full etc. MTS is only able to process error codes returned by the lower communication layers (i.e. CMS and SpaceWire) and it returns its own error codes to the application which called the *send* operation.

[1] Available at `http://www.dit.upm.es/rts/projects/assert/downloads/tech-notes/`.

- **Receiving a message failed**: Receiving a message may fail due to a hardware problem with the SpaceWire chipset. Note that the Communication Stack does not provide any detection as to whether the link or sender processor are working. As with the *send* operation, MTS is only able to process error codes returned by the lower communication layers (i.e. CMS and SpaceWire) and it returns its own error codes to the application which called the *receive* operation. In addition, an application-defined timeout could be associated with each *receive* call, which could be used for higher-level fault detection.
- **Corrupted message**: A message may get corrupted due to problems with hardware or network link. Such a corruption may affect both the message header and body, therefore the communication stack may receive a message which cannot be be decoded to be passed on. The CMS uses checksum to detect some corruptions in the packet.
- **Delayed message**: A message may be delayed due to a processing error or delay within the hardware or on the network line. This may cause an application on the receiver side to miss a deadline. In general, there is no notion of timely message delivery in the communication stack itself. MTS provides an API to detect delayed message (a timeout associated with the *send* operation) so that timing information set in the application layer can be enforced.

3.3.3 Communications partitioning

The ASSERT VM provides communication budgets per partition. The idea is to make sure that no application can either by accident or deliberately overload the communication links by sending big amounts of data frequently. The CMS library provides support for setting bandwidth allocation for different applications, while PolyORB-HI generates the code for enforcing the needed network budgets from user-defined application attributes.

3.4 Middleware

The middleware (PolyORB-HI) handles the correct interaction between application entities (user code), using the services of the real-time kernel and the communication stack.

Faults in a middleware can be either

- static, resulting from a misconfiguration of the system, or
- dynamic, resulting from the occurrence of a fault at runtime.

Static faults. To reduce typical faults when using middleware, mainly concentrated in the configuration of the internals, ASSERT retained an innovative approach. The middleware is instantiated by traversing some views of the user's models. This "by-construction" approach ensures that the middleware is configured correctly and automatically. This ensures no static fault can occurred (incomplete naming, or lack of a communication channel). All static faults are detected at compile-time by the deployment tool.

Dynamic faults. This kind of faults can occur in the application, but as such they are usually not implied by a fault in the middleware itself. In the context of ASSERT, and because of the static instantiation of the middleware layer, the application specific instance of the middleware is reduced to a mailbox system, correctly dimensioned from the information in the model.

Instead, faults may come from the environment and propagated through the middleware (e.g. real-time kernel, breach of a communication channels), or the application (excessive resource usage). The latter may come from a faulty model, whereas the sooner is covered by either the real-time kernel or the communication stack.

Understanding the correct propagation of faults through the middleware, and their eventual contention is an open issue. It is likely to become a strong issue with the increasing coupling between applications in complex component-based systems, like the one foreseen in ASSERT.

Chapter 4

Conclusions

The main components of the ASSERT Virtual Machine have been assessed with respect to basic dependability issues, especially temporal faults, as this is the kind of fault which is more likely to arise from a misbehaviour at the VM or the VMLC component level.

The real-time kernel is the main responsible for enforcing temporal constraints on application tasks. It provides services for detecting all kinds of temporal faults at the kernel level:

- Sporadic tasks released more often than specified,
- deadline misses for periodic and sporadic tasks,
- WCET budget violations for periodic and sporadic tasks.

Some references explaining how these mechanisms can be integrated with fault recovery mechanisms at the VMLC level have been given.

Basic temporal isolation can be implemented by using an additional mechanism provided by the real-time kernel, namely WCET budgets for groups of tasks. Achieving spatial isolation requires either additional hardware support, or preferably changes in the compilation toolchain which are not available yet.

The communications services also offer basic mechanisms for temporal fault detection at this level of abstraction. Some kinds of data-related faults are also covered.

The ASSERT middleware has been designed so that no faults are originated at this level. However, faults can be propagated from other components. The whole issue of analysing fault propagation and contention through the middleware is still open, although a solid basis is already available.

In general, the whole subject of partitioning and fault isolation requires a significant amount of additional effort to be carried out, although the results obtained within the project are encouraging.

References

[CCS07a] CCSDS. *CCSDS 850.0-G-1 Spacecraft Onboard Interface Services, Green Book. Issue 1*, June 2007.

[CCS07b] CCSDS. *CCSDS 851.0-R-1 Spacecraft Onboard Interface Services — Subnetwork Packet Service*, 2007.

[dlPRZ00] Juan A. de la Puente, José F. Ruiz, and Juan Zamorano. An open Ravenscar real-time kernel for GNAT. In Hubert B. Keller and Erhard Plödereder, editors, *Reliable Software Technologies — Ada-Europe 2000*, number 1845 in LNCS, pages 5–15. Springer-Verlag, 2000.

[Gai05] Gaisler Research. *LEON2 Processor User's Manual*, 2005.

[ISO05] ISO/IEC. *TR 24718:2005 — Guide for the use of the Ada Ravenscar Profile in high integrity systems*, 2005. Based on the University of York Technical Report YCS-2003-348 (2003).

[ISO07] ISO/IEC. *Ada 2005 Reference Manual. Language and Standard Libraries. International Standard ISO/IEC 8652/1995(E) with Technical Corrigendum 1 (ISO/IEC 8652:1995:TC1:2000) and Amendment 1 ISO/IEC 8526:AMD1:2007*, 2007. Published by Springer-Verlag, ISBN 978-3-540-69335-2.

[PdlPB+07] José Pulido, Juan A. de la Puente, Matteo Bordin, Tullio Vardanega, and Jérôme Hugues. Ada 2005 code patterns for metamodel-based code generation. *Ada Letters*, XXVII(2):53–58, August 2007. Proceedings of the 13th International Ada Real-Time Workshop (IRTAW13).

[PUZ+06] José A. Pulido, Santiago Urueña, Juan Zamorano, Tullio Vardanega, and Juan A. de la Puente. Hierarchical scheduling with Ada 2005. In Luís Miguel Pinho and Michael González Harbour, editors, *Reliable Software Technologies — Ada-Europe 2006*, volume 4006 of *LNCS*. Springer Berlin / Heidelberg, 2006.

[PUZd07] José A. Pulido, Santiago Urueña, Juan Zamorano, and Juan A. de la Puente. Handling temporal faults in Ada 2005. In Nabil Abdennadher and Fabrice Kordon, editors, *Reliable Software Technologies — Ada-Europe 2007*, number 4498 in LNCS, pages 15–28. Springer-Verlag, 2007.

[SRL90] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9), 1990.

[UPRZ07] Santiago Urueña, José Antonio Pulido, José Redondo, and Juan Zamorano. Implementing the new Ada 2005 real-time features on a bare board kernel. *Ada Letters*, XXVII(2):61–66, August 2007. Proceedings of the 13th International Real-Time Ada Workshop (IRTAW 2007).

[VHPK04] Thomas Vergnaud, Jérôme Hugues, Laurent Pautet, and Fabrice Kordon. PolyORB: a schizophrenic middleware to build versatile reliable distributed applications. In *Proceedings of the 9th International Conference on Reliable Software Techologies Ada-Europe 2004 (RST'04)*, volume LNCS 3063, pages 106–119, Palma de Mallorca, Spain, June 2004. Springer Verlag.

[ZdlPHV07] Juan Zamorano, Juan A. de la Puente, Jérôme Hugues, and Tullio Vardanega. Run-time mechanisms for property preservation in high-integrity real-time systems. In *OSPERT 2007 — Workshop on Operating System Platforms for Embedded Real-Time Applications*, Pisa. Italy, July 2007.