Use of SIMA in the DIANA Project White Paper

GMV-SKYSOFT
Torre Fernão de Magalhães
Av. D. João II Lote 1.17.02, 7º Andar
1998 - 025 Lisboa Portugal

TABLE OF CONTENTS

LIST OF FIGURES

1 INTRODUCTION

1.1 PURPOSE

DIANA was a research and technology development project funded through the European Commission in the scope of the Sixth Framework Programme (FP6). It aimed at the implementation of a new avionics platform based on the concepts of Integrated Modular Avionics and the ARINC 653 specification. This new platform AIDA (Architecture for Independent Distributed Avionics) proposes a series of novelties that ease on-board software development. These novelties consist in both: new tool concepts and new run-time technologies. In terms of tools, AIDA proposes

- To base development of critical on-board functions on model-based engineering for early validation and verification of architectures as well as for automatic generation of code and configuration artefacts;
- To use formal methods for early validation and verification of algorithms.

In terms of new run-time technologies, AIDA proposes

- A new concept of software components that provides
  - Basic services and composed services (consisting of a collection of components);
  - Partition- and module-local services and platform-wide services;

  Those service components are described by configuration files and provide well-defined interfaces; as such they can be offered by different suppliers and can be plugged into a system at different physical locations without affecting application code.

  The components extend the ARINC 653 specification to the platform level and, at the same time, they use ARINC 653 basic services (defined in part 1 and part 2 of the specification) to implement the advanced features they provide.
- A new communication paradigm based on the Publish and Subscribe architecture; in the scope of DIANA, a Publish and Subscribe library was implemented on top of ARINC 653 that follows the specification of the Object Management Group's (OMG) Data Distribution Services (DDS).
- New Reconfiguration approaches, based on the DDS library for in-flight reconfiguration and, for pre-flight reconfiguration, based on an innovative concept, called Multi-Static Reconfiguration (MSR) that is able to improve availability of aircrafts without increasing the amount of hardware.
- The use of object-oriented programming languages and virtual machines (VM) executing applications. The project implemented a Java virtual machine, based on Atego's PERC Pico Safety Critical Java VM. This VM was completely ported to ARINC 653.

In the scope of the DIANA project two demonstrators were built to validate the concepts and their implementation. Both demonstrators use safety-critical avionics applications that had been developed with real on-board requirements. One of these demonstrators was based on a Flight Warning System (FWS), developed by THALES, and the other was based on an Environmental Control System (ECS), developed by Dutch National Aerospace Laboratory (NLR). Both demonstrators run on a heterogeneous system consisting of real target Power PC (PPC) boards and Intel-based desktops. On the PPC boards, the demonstrators used Windriver's VxWorks 653 Safety Critical Platform; on the desktops, the FWS used VxSim, a host-based simulator part of the VxWorks 653 tool chain, and the ECS used SIMA, GMV's ARINC 653 simulator.

This document describes the use of SIMA in the ECS demonstrator. In **section 2**, a short overview on the SIMA execution environment is given. In **section 3**, which is based on contributions by NLR, the ECS Application is presented. In **section 4**, the integration of the application with the AIDA components is described. In **section 5**, the extended set-up of the demonstrator at the exhibition of the Avionics Event in Amsterdam 2010 is shown. **Section 6**, finally, presents some conclusions.

1.2 ACRONYMS

ADIRU	Air Data Inertial Reference Unit
AIDA	Architecture for Independent Distributed Avionics
API	Application Programming Interface
APEX	Application Executive
ARD	Application Requirements Descriptor
ARINC	Avionics Radio Inc.
BITE	Built-In Test Equipment
CDS	Cockpit Display System
COTS	Commercial Off-The-Shelf
CPIOM	Core Processing Input/Output Module
CPM	Core Processing Module
CPU	Central Processing Unit
DAL	Development Assurance Level
DDS	Data Distribution Services
DIANA	Distributed Equipment Independent environment for Advanced avionics Applications
EC	European Commission
ECS	Environmental Control System
ESA	European Space Agency
EU	European Union
FAA	Federal Aviation Administration
FIFO	First-In/First-Out
FWS	Flight Warming System
GCC	GNU Compiler Collection
GPL	General Public License
HMI	Human Machine Interface
HW	Hardware

IMA	Integrated Modular Avionics
IDE	Integrated Development Environment
LIFO	Last-In/First-Out
MDA	Model-Driven Architecture
MDB	Model Based Development
MOS	Module Operating System
MSR	Multi-Static Reconfiguration
NPTL	Native POSIX Thread Library
OMG	Object Management Group
OS	Operating Systems
PC	Personal Computer
PBIT	Power-up Built-In Test
PDD	Platform Definition Descriptor
PIM	Platform Independent Model
POS	Partition Operating System
PowerPC	Performance Optimisation With Enhanced RISC Performance Computing
PPC	PowerPC
PSM	Platform-Specific Model
POSIX	Portable Operating System Interface
RTOS	Real Time Operating System
RTSJ	Real-Time Specification for Java
SCJT	Safety-Critical Java Technology
SIMA	Simulated Integrated Modular Avionics
SDD	Service Definition Descriptor
SW	Software
VM	Virtual Machine

1.3 REFERENCE DOCUMENTS

Ref.	Title
[AD.1]	Airlines Electronic Engineering Committee (AEEC). Avionics Applications Software Standard Interface (ARINC Specification 653 Part 1 – Required Services). ARINC Inc., 2006.
[AD.2]	Airlines Electronic Engineering Committee (AEEC). Avionics Applications Software Standard Interface (ARINC Specification 653 Part 2 – Extended Services). ARINC Inc., 2008.
[AD.3]	Airlines Electronic Engineering Committee (AEEC). Avionics Applications Software Standard Interface (ARINC Specification 653 Part 3 – Conformity Test Specification). ARINC Inc., 2006.
[AD.4]	Christian Engel, Eric Jenn, Peter H. Schmitt, Rodrigo Coutinho, Tobias Schoofs: Enhanced Dispatchability of Aircrafts using Multi-Static Configuraltion, ERTS2, 2010.
[AD.5]	Tobias Schoofs, Eric Jenn, Stéphan Leriche, Kelvin Nilson, Ludovic Gauthier, Marc Richard-Foy: Use of PERC Pico in the AIDA Avionics Platform, JTRES, September, 2009.
[AD.6]	GMV: SIMA Overview, January, 2010.
[AD.7]	GMV: SIMA Command Line Tools – Application Development and Configuration Guide, January 2010.
[AD.8]	GMV: The AIDA System, DIANA White Paper, January 2008.
[AD.9]	EuroSim: Pushing Real-time Simulation to the Limit, http://www.nlr.nl/?id=12264&l=en

1.4 INDEX OF COTS

Name.	Description	Vendor
APERO	Avionics Prototyping Environment for Research and Operations	NLR
AVT	ARINC 653 Validation Test Suite; implementation of the ARINC 653 Part 3 – Conformity Test Specification	GMV
EuroSim	Real-Time Simulator with person or hardware in the loop	NLR, DutchSpace, TASK24
GNU	Gnu is Not Unix. Free Operating System Software.	Free Software Foundation (FSF)
PERC Pico	Safety Critical Java Virtual Machine.	Atego
PikeOS	ARINC 653 compliant RTOS	SYSGO
SIMA	ARINC 653 RTOS simulator for Linux	GMV
Simulink	Environment for multi-domain simulation and model-based design form dynamic and embedded systems	MathWorks
Vincent	Prototyping tool for glass cockpit displays	NLR
VxWorks 653	ARINC 653 compliant and DO-178B-certifiable RTOS	Wind River

2 SIMA OVERVIEW

Simulated Integrated Modular Avionics (SIMA) is an execution environment, providing the ARINC 653 Application Programming Interface (API) and robust partitioning to operating systems that do not support these features by themselves. SIMA is designed to run on all POSIX-compliant OSes; it is tested and optimised for the Native POSIX Thread Library (NPTL), available on OSes like GNU/Linux, kernel version 2.6 or higher, and for RTEMS, version 4.6 or higher.

The ARINC 653 standard ([AD.1], [AD.2]) specifies a programming interface for a Real-Time Operating System (RTOS), and, in addition, establishes a particular method for partitioning resources over time and memory. Today, this standard has been established as an important foundation for the development of safety-critical systems in the avionics industry.

ARINC 653 defines support for robust partitioning in on-board systems, such that one processing unit, usually called a module, is able to host one or more avionics applications and to execute these applications independently. This can be achieved if the underlying system, often called the Module Operating System (MOS), provides separation of the avionics applications, such that

- Each partitioned function has guaranteed access to the processor. The guarantees shall reflect the frequency as well as the execution time of the specific application;
- A failure in one partitioned function cannot cause a failure in another partitioned function.

In consequence, the partitioning approach allows reducing on-board hardware and, at the same time, eases verification, validation and certification.

The unit of partitioning is called a partition. In a given sense, a partition is equivalent to a program in a single application environment: it comprises data, code and its own context configuration attributes (see Figure 1).

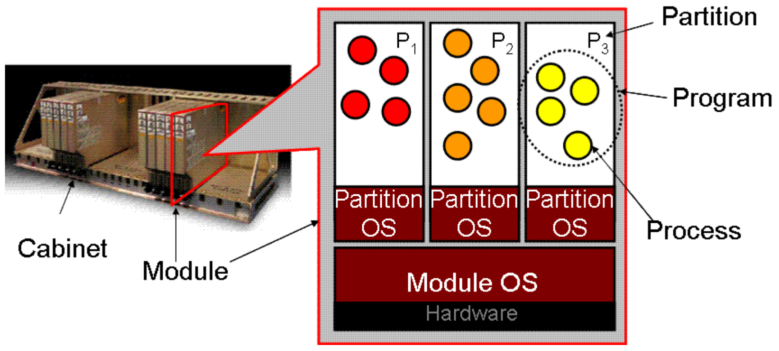


Figure 1: Partitioning

Partitioning separates applications in two dimensions: space and time. Spatial separation means that the memory of a partition is protected. No application can access memory out of the scope of its own partition. Temporal separation means that only one application at a time has access to system resources, including the processor; therefore only one application is executing at one point in time – there is no competition for system resources between partitioned applications.

ARINC 653 defines a static configuration where each partition is assigned a set of execution windows. The program in the partition associated with the current execution window gains access to the processor. When the execution window terminates, the program is preempted; when the next execution window starts, the program continues execution from the point it was previously preempted.

Processes within the scope of a partition are scheduled by a priority-based preemptive scheduler with first-in-first-out (FIFO) order for processes with the same priority.

Processes in ARINC 653 must not be confused with processes in POSIX: In ARINC 653, processes within the same partition share the same address space. There is no memory separation between processes. However, since partitions are separated, processes in different partitions cannot access each other's memory. Communication between processes in different partitions is achieved by ports and channels. Ports are communication end points either for reading or writing that are identified by a name that is unique in the scope of the partition. Channels connect these ports transparently to application code.

In SIMA, ARINC 653 partitions are mapped to POSIX processes and ARINC 653 processes are mapped to POSIX threads. Each SIMA application is, hence, linked to a single POSIX program, containing user code and data, the APEX code and data and, finally, the platform execution environment, i.e. the NPTL for Linux.

The Module Operating System (MOS), controlling the different POSIX processes, belonging to the same simulated module, is likewise linked to one POSIX process. The following picture illustrates this design:

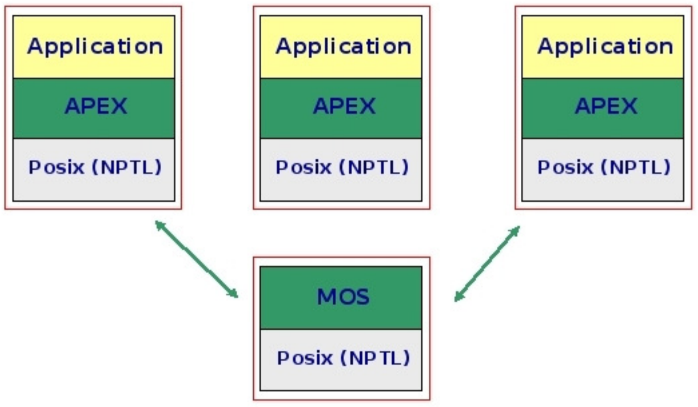


Figure 2: SIMA Architecture

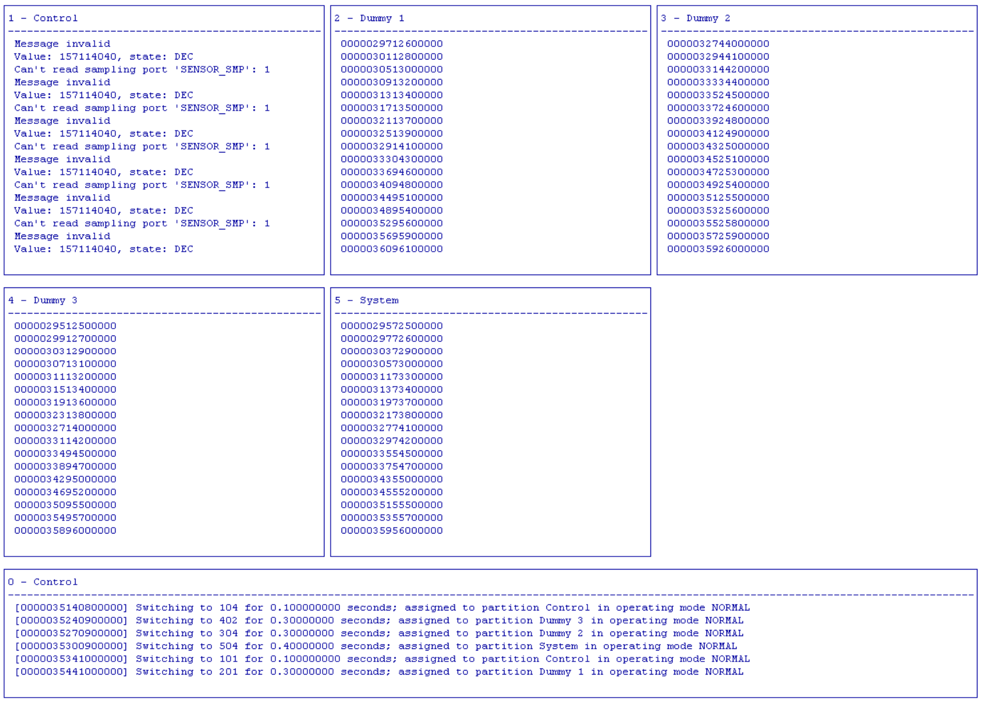
The APEX services are implemented by a static library, called POS. The POS implements the APEX process scheduler on top of the POSIX FIFO scheduler (*sched_fifo*). POSIX features are encapsulated within a core layer; this way main parts of the APEX code do not rely directly on POSIX, but on scheduling policies implemented by the POS itself. The advantage of this approach is enhanced portability - there is even an implementation of the SIMA POS, running on bare hardware - and the fact that scheduler features that introduce subtle differences between different POSIX implementations are handled in the core layer and hidden from the APEX implementation.

The MOS implements the APEX partition scheduler. To be able to suspend and resume partitions, commands are exchanged with the POS layer in the partitions using signals and shared memory segments. Obviously, this approach does not answer safety and security threats, caused by random errors in the partitioned code. The POS has to respond correctly to the given commands which it may fail to do in the case where faulty or malicious application code has corrupted the state of the POS. In fact, the POS was designed and developed, following safety critical software guidelines; its purpose is to support embedded applications. The MOS, however, was not; the MOS does only simulate the behaviour of an ARINC 653 compliant OS on top of non-safety aware systems like standard Linux.

POS and MOS are designed to support real-time applications. They use the real-time programming interfaces of the POSIX thread library, like FIFO scheduling and thread priorities. Additionally, all memory used during execution is created during initialisation and locked in RAM, avoiding paging and the latency penalties caused by swapping pages in and out. However, hard real-time guarantees cannot be met without a fully preemptive operating system kernel. If hard real-time is wanted, the PREEMPT-RT patch by Ingo Mólnar and Thomas Gleixner can be used to achieve guarantees for very short deadlines.

ARINC 653 queuing and sampling ports are mapped to UDP ports. This way, ports can be flexibly linked to other ports on the same virtual module or to external resources represented by a pseudo partition. Alternative mappings, e.g. to TCP/IP or even to I/O devices and protocols, can be defined by means of user callbacks.

For visual control over the execution of ARINC 653 applications on SIMA, the *simout* tool is provided. The *simout* program shows the output of the MOS and up to six partitions in a graphical environment based on the *ncurses* library. The following figure gives an impression of a module with five partitions running in the *simout* environment:



```
1 - Control
---------------------------------------------------
 Message invalid
 Value: 157114040, state: DEC
 Can't read sampling port 'SENSOR_SMP': 1
 Message invalid
 Value: 157114040, state: DEC
 Can't read sampling port 'SENSOR_SMP': 1
 Message invalid
 Value: 157114040, state: DEC
 Can't read sampling port 'SENSOR_SMP': 1
 Message invalid
 Value: 157114040, state: DEC
 Can't read sampling port 'SENSOR_SMP': 1
 Message invalid
 Value: 157114040, state: DEC
 Can't read sampling port 'SENSOR_SMP': 1
 Message invalid
 Value: 157114040, state: DEC
```

```
2 - Dummy 1
---------------------------------------------------
 0000029712600000
 0000030112800000
 0000030513000000
 0000030913200000
 0000031313400000
 0000031713500000
 0000032113700000
 0000032513900000
 0000032914100000
 0000033304300000
 0000033694600000
 0000034094800000
 0000034495100000
 0000034895400000
 0000035295600000
 0000035695900000
 0000036096100000
```

```
3 - Dummy 2
---------------------------------------------------
 0000032744000000
 0000032944100000
 0000033144200000
 0000033334400000
 0000033524500000
 0000033724600000
 0000033924800000
 0000034124900000
 0000034325000000
 0000034525100000
 0000034725300000
 0000034925400000
 0000035125500000
 0000035325600000
 0000035525800000
 0000035725900000
 0000035926000000
```

```
4 - Dummy 3
---------------------------------------------------
 0000029512500000
 0000029912700000
 0000030312900000
 0000030713100000
 0000031113200000
 0000031513400000
 0000031913600000
 0000032313800000
 0000032714000000
 0000033114200000
 0000033494500000
 0000033894700000
 0000034295000000
 0000034695200000
 0000035095500000
 0000035495700000
 0000035896000000
```

```
5 - System
---------------------------------------------------
 0000029572500000
 0000029772600000
 0000030372900000
 0000030573000000
 0000031173300000
 0000031373400000
 0000031973700000
 0000032173800000
 0000032774100000
 0000032974200000
 0000033554500000
 0000033754700000
 0000034355000000
 0000034555200000
 0000035155500000
 0000035355700000
 0000035956000000
```

```
0 - Control
------------------------------------------------------------------------------------------------------------------------------------------------------------
 [0000035140800000] Switching to 104 for 0.100000000 seconds; assigned to partition Control in operating mode NORMAL
 [0000035240900000] Switching to 402 for 0.30000000 seconds; assigned to partition Dummy 3 in operating mode NORMAL
 [0000035270900000] Switching to 304 for 0.30000000 seconds; assigned to partition Dummy 2 in operating mode NORMAL
 [0000035300900000] Switching to 504 for 0.40000000 seconds; assigned to partition System in operating mode NORMAL
 [0000035341000000] Switching to 101 for 0.100000000 seconds; assigned to partition Control in operating mode NORMAL
 [0000035441000000] Switching to 201 for 0.30000000 seconds; assigned to partition Dummy 1 in operating mode NORMAL
```

Figure 3: Partition Visualisation

The ARINC 653 services that are currently implemented by SIMA are:

- All required services defined in Part 1 of the specification:
  - Partition Management Services;
  - Process Management Services;
  - Time Management Services;

    - Health Monitoring Services;
    - Intra-Partition Communication Services;
    - Inter-Partition Communication Services;
- Some extended services defined in Part 2 of the specification:
    - Multiple Module Schedules;
    - Logbook System.

More extended services of Part 2 are under development, in particular:

- File System;
- Service Access Points;
- Naming Services;
- Sampling Port Extensions;
- Memory Blocks.

For more information on the SIMA simulation environment, please refer to [AD.6] and [AD.7]

3 THE ECS APPLICATION[1]

In the scope of the DIANA project, a case study was defined that involves a real-life scenario of an environmental control – or air conditioning – system. A scenario was defined that was easy to understand, but sufficiently complex to serve as a representative use case for AIDA. The case study focuses on a generic air conditioning system representative of systems currently in use on regional and long-distance air transport aircrafts.

Note that an actual environment control system involves more subsystems than discussed in this document. For simplicity, the description of the system is limited to the subsystems relevant to the case study.

The ECS involves the following avionics systems, see Figure 4:

- Air conditioning panel;
- Zone controller;
- Pack controller;
- System display.

The air conditioning pack is regulated by the pack controller to supply the mixing unit with a sufficient flow of cool fresh air. This air is supplied to the cockpit and cabin zones. In order to regulate the temperature of this airflow, the zone controller regulates the amount of hot air added to the flow of cool air.

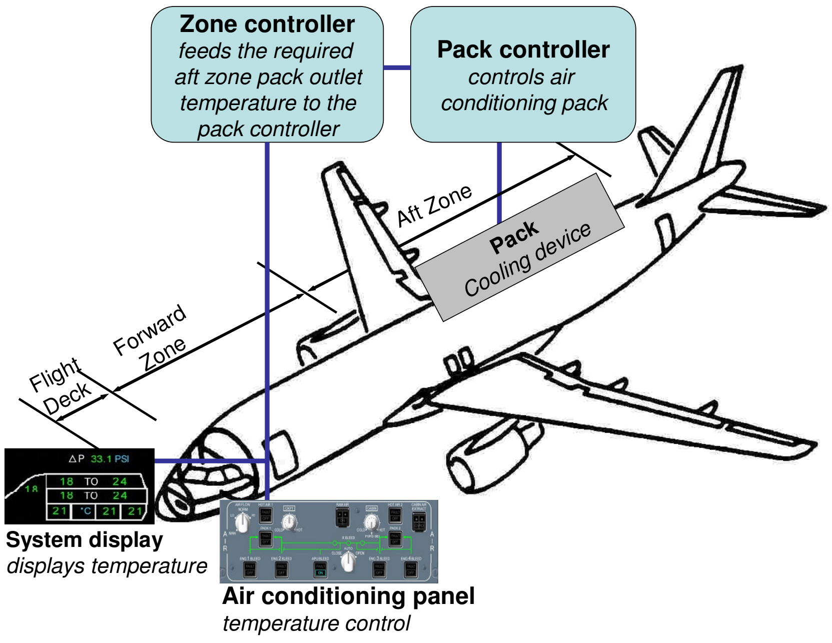


Figure 4: Air conditioning system[2]

[1] This section is based on contriubutions by NLR.

[2] © NLR, 2010, all rights reserved

Air conditioning panel

The desired temperature in the cabin zones and in the cockpit can be manually selected on the overhead panel by the pilot. The pilot may increase or decrease the temperature in order to obtain the desired temperature. This is depicted in Figure 5.

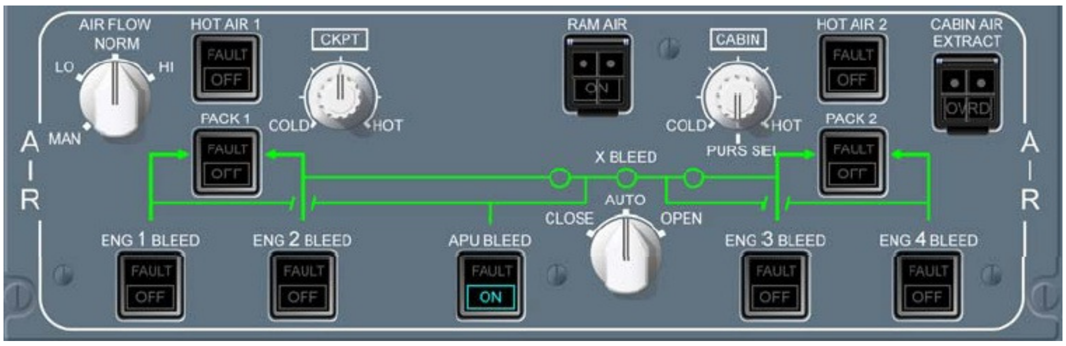


Figure 5: Air conditioning panel[3]

Zone controller

The temperature selections on the air conditioning panel of cockpit, forward and aft cabin are read by the zone controller. The zone controller regulates zone temperatures to match temperature selections on the air conditioning panel for cockpit, forward and aft cabin. It is responsible for setting the pack discharge temperatures thereby maintaining an optimal setting of the cold air mixer unit temperature. By supplying trim air to the zone inlet ducts it keeps the temperature as requested.

Zone control is divided into separate control functions for each zone of the aircraft. Each control function has the following inputs:

- Temperature selection (from air conditioning panel);
- Zone temperature (measurement in each zone);
- Air duct temperature (measurement in each duct for each zone);
- Aircraft altitude (from ADIRS).

Each function calculates the desired temperature of the mixer unit, the lowest temperature demand will determine the actual mixer unit temperature. In addition, each function calculates the desired amount of trim air to be added to the air duct in order to obtain the required duct demand temperature. The outputs of each function therefore are:

- Desired mixer unit temperature;
- Trim air demand.

Pack controller

The zone controller feeds the required pack outlet temperature to the pack controllers.

The pack controller then sets the water extractor outlet temperature in accordance with demands from the zone controller, by modulating the ram-air doors and by-pass valve.

[3] © NLR, 2010, all rights reserved

For safety reasons, the pack controller is redundant. When the primary pack controller fails (as detected by the pack controller Build-In Test Equipment (BITE)), the secondary pack controller takes over (see **Figure 6**).

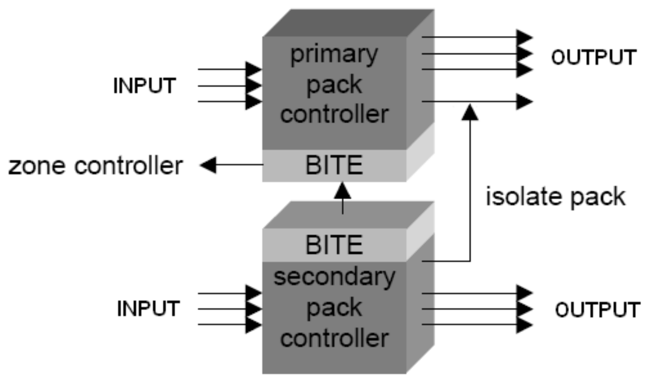


Figure 6: Redundant pack controllers[4]

Pack control is divided in flow control and temperature control functions. The flow control function takes the desired pack flow and regulates the pack flow valve. Its input is:

- Pack flow demand (from air conditioning panel).

Its output is:

- Pack flow valve setting.

The pack temperature control function regulates the pack temperature by modulating the by-pass valve and ram air doors. Its inputs are:

- Pack outlet temperature (measurement);
- Pack temperature demand (from mixer control).

Its outputs are:

- Ram air in demand;
- Ram air out demand;
- By-pass valve setting.

[4] © NLR, 2010, all rights reserved

System Display

On the system display, the cruise page monitors the cabin temperature and pressure and informs the pilot about the actual temperature in the cockpit and cabins (Figure 7). The cruise page provides the pilot some basic information from the air conditioning page.

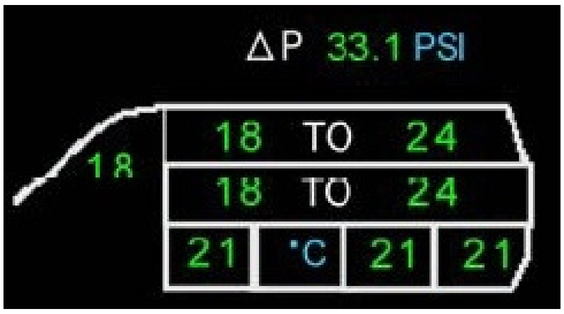


Figure 7: AIR section of Cruise page[5]

[5] © NLR, 2010, all rights reserved

4 THE DEMONSTRATOR INTEGRATION

4.1 AIDA COMPONENTS

AIDA ([AD.8]) is an IMA-based platform, backward compatible with the ARINC 653 standard. This means that AIDA is compatible with ARINC 653 COTS RTOS, certifiable at highest DO-178B/C DAL level and commercially available today. AIDA enhances aspects of ARINC 653 and the current state-of-the-art in IMA, namely it improves the neutrality of the IMA execution environment regarding the underlying hardware and operating system; it enhances the location transparency and it supports a development and integration process based on model-driven engineering and formal methods. The following figure gives an overview on the AIDA architecture:

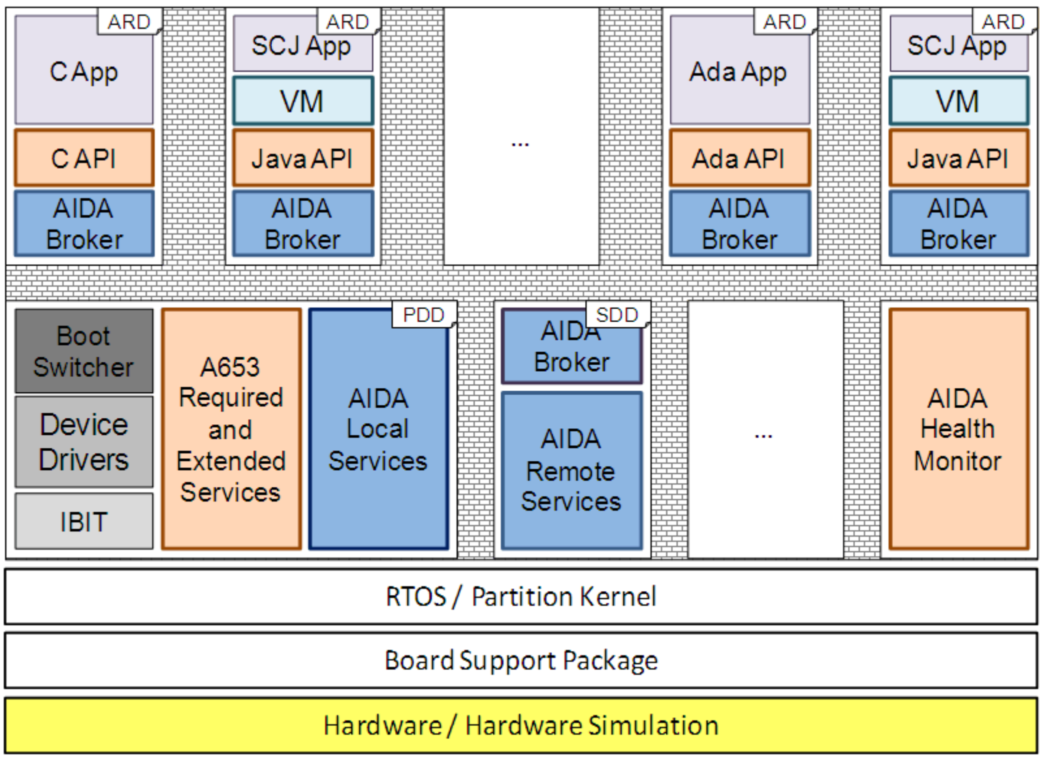


Figure 8: AIDA Architecture

The basic building blocks in the AIDA platform are partitions as defined in the ARINC 651 and 653 standards. Partitions are fault and change containment units and as such relevant for incremental certification of applications and services as well as for application deployment and reuse.

Three kinds of partitions, defined by their language binding, are supported: C, Ada and Java partitions. In general, it is forbidden to mix languages at application level within one partition. Concerning Java, this requirement is relaxed. The compilation model of the Java execution environment foresees an automated conversion to C code; moreover, Java applications interface directly with the C code of those AIDA middleware components that are directly linked into the partition.

Applications rely on the ARINC 653 API. Additionally, they can and shall use services defined by the AIDA middleware to invoke local or remote services and to exchange data, based on the publish/subscribe paradigm.

The API level of the middleware is based on ARINC 653 and – logically - hosted as layer in the partitions. Note that RTOSes may implement this architecture differently; VxWorks 653, for instance, does not instantiate the partition operating system (POS) once per partition; instead the POS is linked to the partition's virtual address space. This is depicted in **Figure 8** by separating A653 services in an own partition.

Other components, namely, the AIDA platform services, the AIDA broker, responsible for remote invocation and data distribution services, the reconfiguration services (Boot Switcher, System Manager) and the System Health Monitor may be placed in separate partitions. In Figure 8, this is depicted by placing a layer of partitioned platform components below the application layer.

The elements of the AIDA architecture, such as services, platform and applications, are controlled by configurations given in descriptors. Applications are defined by their requirements (memory, time resources, services) collected in the Application Requirements Descriptor (ARD). Services are defined by the resources they provide, captured in the Service Definition Descriptor (SDD). The platform as a whole is defined in terms of applications and services on one hand and available hardware resources on the other. This information is collected in the Platform Definition Descriptor (PDD) that is made available through dedicated services.

The descriptors are generated from a model-based tool chain. The model engineer first defines a platform independent model which describes application components, communication channels, provided and requested services, message formats and so on. This high-level architecture is then mapped onto the platform, *i.e.* to hardware nodes, partitions, communication buses, *etc.* In contrast to general purpose modelling approaches that are based on automatic transformations, this mapping is done interactively using the PIM-PSM Mapping editor. During the mapping process, the system-wide architecture is broken down to modules and their partitioning layouts. The output of the process is a description of the mapping of the architecture to the platform, hence, a new model, but also configuration items on module level for different operating system (currently, VxWorks and SIMA) describing memory layout, partition schedules, connection tables and the location of service components. The tool also generates glue code needed to link the components together.

On porting the ECS application to the AIDA platform, a set of components had to be integrated into the development tool chain and into the run-time environment. Figure 9 gives an overview on the components that were used during demonstrator integration:

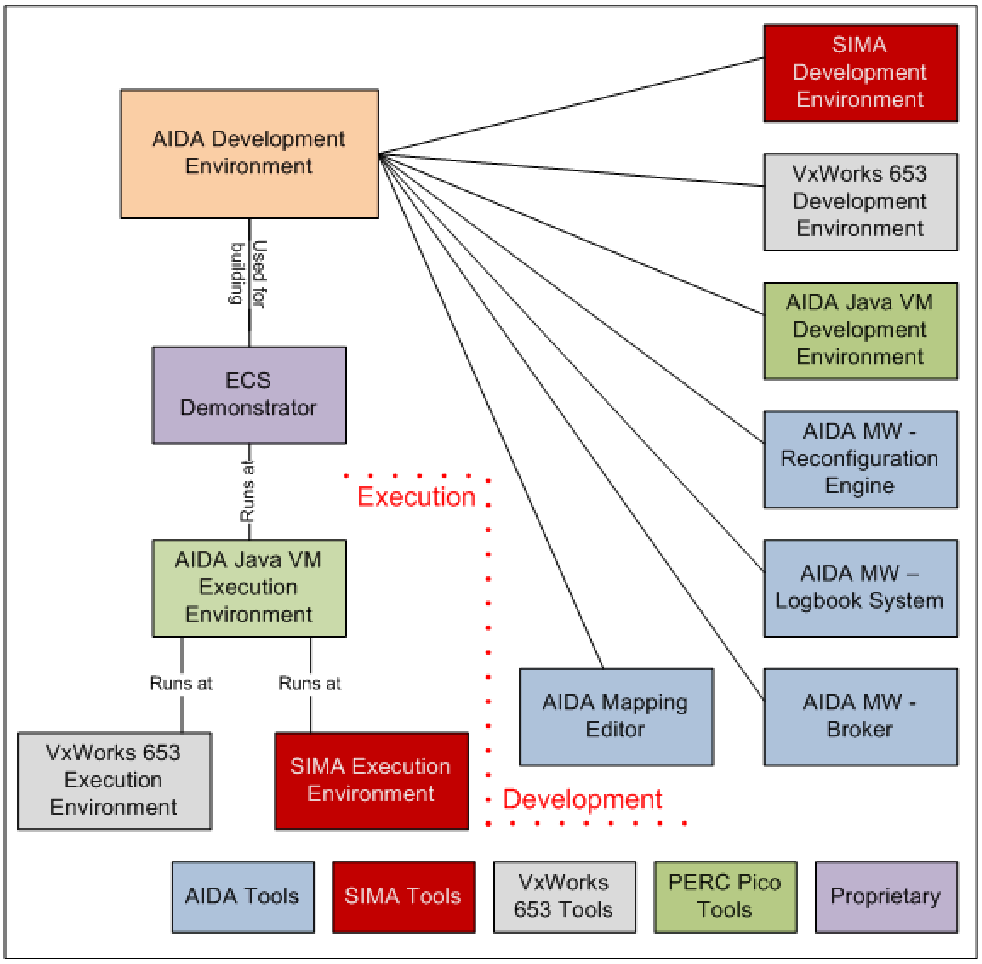

SIMA Development Environment
AIDA Development Environment
VxWorks 653 Development Environment
Used for building
AIDA Java VM Development Environment
ECS Demonstrator
AIDA MW - Reconfiguration Engine
Runs at
Execution
AIDA Java VM Execution Environment
AIDA MW – Logbook System
Runs at
Runs at
AIDA Mapping Editor
AIDA MW - Broker
VxWorks 653 Execution Environment
SIMA Execution Environment
Development
AIDA Tools
SIMA Tools
VxWorks 653 Tools
PERC Pico Tools
Proprietary

Figure 9: AIDA Components

The components are:

- The ECS application;
- The SIMA execution and development environments;
- The VxWorks execution and development environment;
- The AIDA Mapping Editor that defines a mapping of high-level architecture models to the platform;
- The AIDA Java VM that was implemented using Atego’s Safety Critical Java VM PERC Pico;
- The AIDA Logbook System; this is a remote service that provides logbooks similar to ARINC 653 Logbooks (described in part 2 of the standard). The logbooks as well as the client applications that use these logbooks are location transparent. Logbook and application instances can be hosted on different nodes in the system. Schedules, memory resources and communication lines are configuration controlled and the necessary artefacts like configurations and glue code are generated by the model-based tool chain;
- The AIDA Reconfiguration Engine, based on Multi-Static Configurations; this is a platform-wide service that allows the re-hosting of applications in case of hardware failures on start-up.

- The AIDA Broker; this is a set of components, implementing the DDS communication infrastructure; the component is not further described in this document.

4.2 DEMONSTRATOR ARCHITECTURE

The ECS components were implemented on three computers, two Intel-based desktop computers running SIMA on top of Linux, and one PPC on-board computer. Additionally, a Concurrent real-time Linux system was used to host an environment simulator. The simulator was connected to the ECS system by means of a standalone SIMA application that uses the ARINC 653 API, without time partitioning (ECS Plant Bridge). Two Windows-based desktop computers served as display and control station and development host, respectively. Figure 10 depicts this architecture:

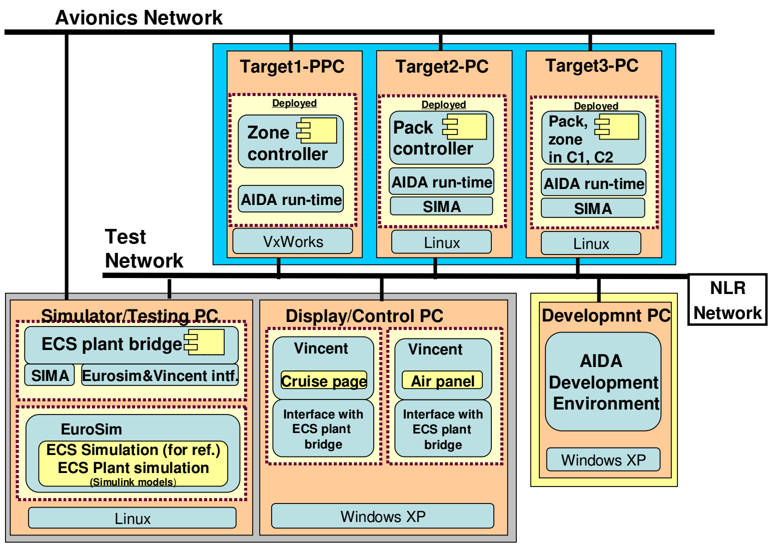


Figure 10: Demonstrator Architecture[6]

The computers were connected by two networks: an avionics network for application interoperability and a test network that was used for measurements, test execution and component deployment.

For the environment simulator the EuroSim [AD.9] simulation framework was used. The display system is based on the glass cockpit emulator Vincent.

Note that Figure 10 does not show the complete mapping of all components (*e.g.* logbooks) to modules and partitions. The reason is that the demonstrator uses multi-static configurations, *i.e.* several pre-defined mappings for the case of hardware failures during start-up. There is, hence, not one mapping of components to hardware nodes and partitions, but a set of such mappings. Complete descriptions are given below in section 4.6.

The following photo gives an impression of the ECS test bench:

[6] © NLR, 2010, all rights reserved

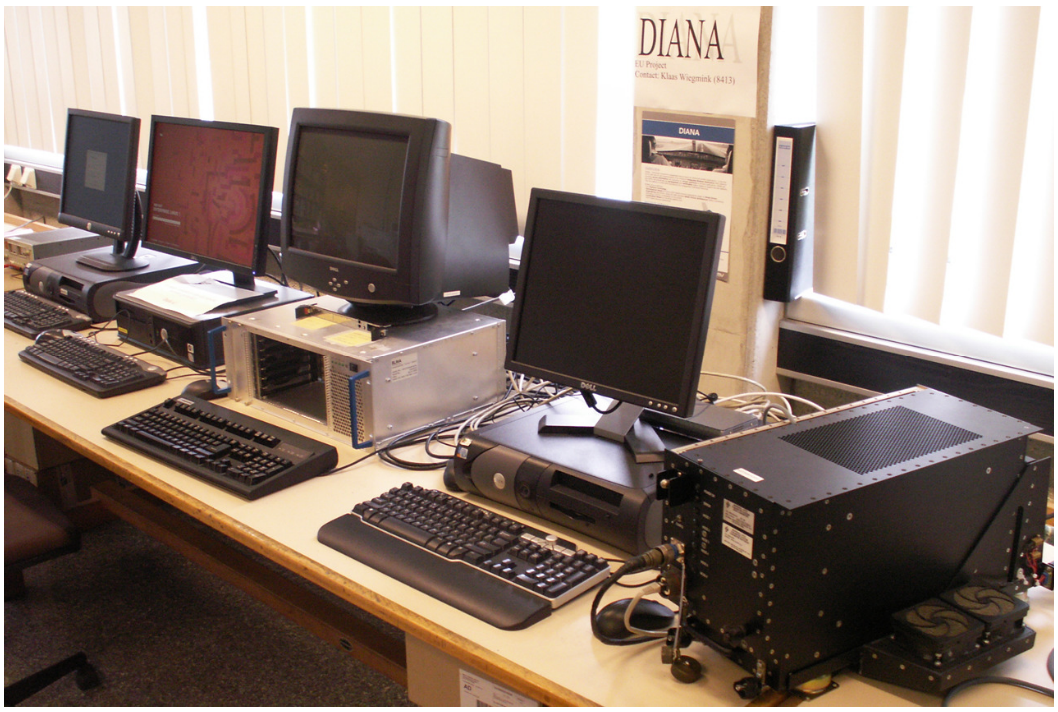


Figure 11: Demonstrator Test Bench[7]

The computers on this picture are (from left to right):

- Target 2-PC;
- Target 3-PC;
- The PPC development board for the VxWorks system (Target 1-PPC);
- The display control station, running Vincent;
- The real target computer (used as Target 1-PPC for demonstration purposes).

[7] © NLR, 2010, all rights reserved

4.3 THE AIDA DEVELOPMENT ENVIRONMENT

The AIDA tool chain is quite complex. It comprises artefacts on the system-wide platform level, the module level and the partition level. Moreover, it integrates tools from different vendors such that output from one set of tools has to match the expected input of another set of tools. See Figure 12 as an illustration of the SIMA-based tool chain:

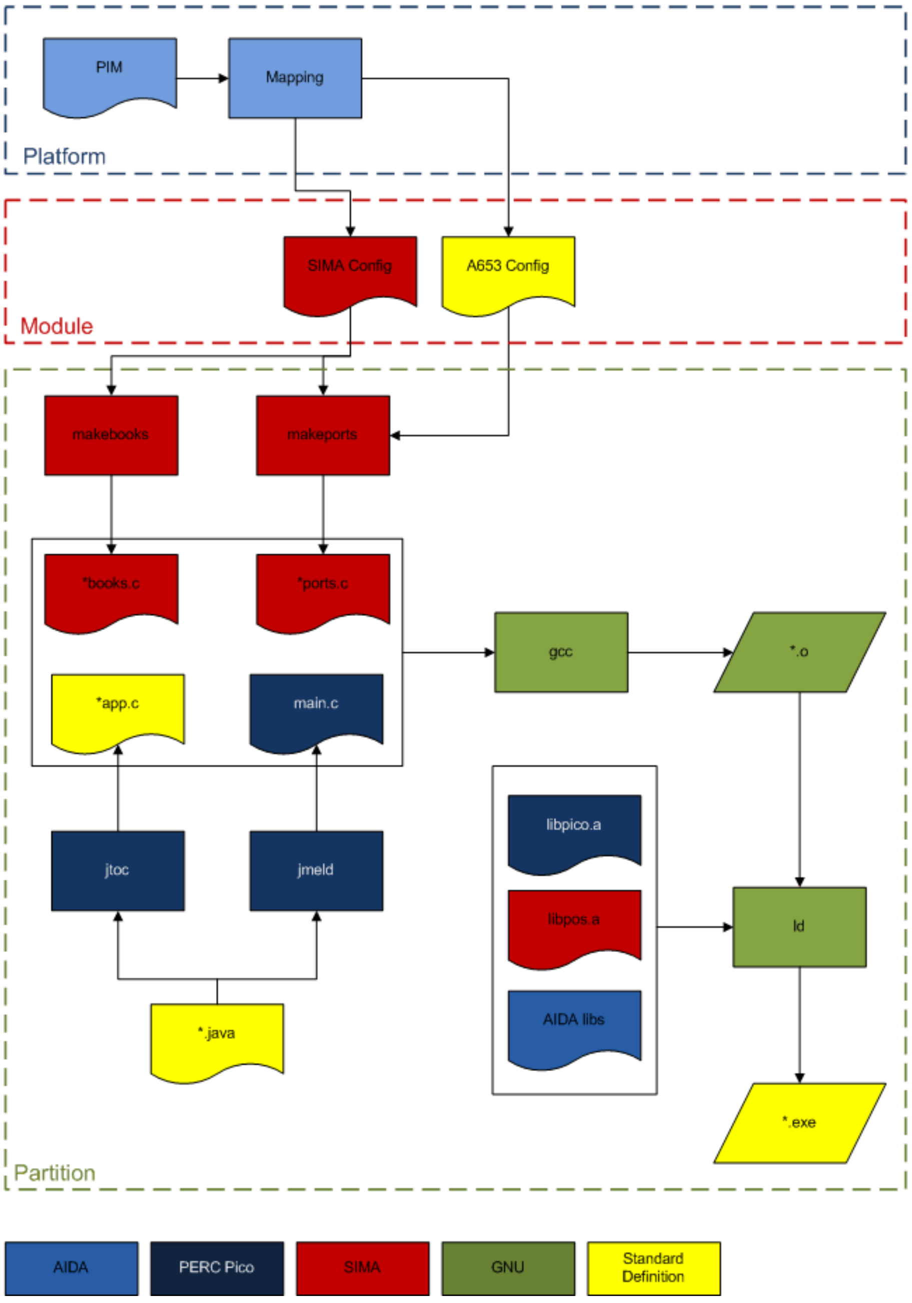


Figure 12: AIDA Toolchain for SIMA

The top-most input to the tool chain is the Platform Independent Model (PIM) that describes the system on architecture level. The PIM is mapped onto the available hardware and the ARINC 653 platform. This is done by means of the mapping editor, a tool developed in the scope of the DIANA project. The output of the tool is a set of configuration and code artefacts that are introduced into the target platform tool chains. In particular, the mapping tool creates an ARINC 653 standard configuration file and the SIMA-specific configuration file. The configuration files have module scope, but the further processing of the configuration in the development tool chain enters directly the application (i.e. partition) level. Note that the configuration is used also during execution of the module, but this is of course not visible in this figure.

A set of code artefacts is needed for ports and logbooks; these artefacts are created by the *makebooks* and *makeports* tools from the SIMA tool chain.

The application-specific C code may come from different sources. In the case of Java applications, the C code is generated by PERC Pico-specific tools.

For applications, written directly in C, this code is generated by human engineers or code generators outside the AIDA tool chain in the strict sense, such as SIMULINK.

Note that Ada was not used for the ECS demonstrator and, hence, no Ada-specific tool is shown in the figure.

When all C artefacts are created the tool chain flows into the standard GNU compile chain: The C files are compiled with GCC and linked with LD, adding a set of libraries, coming from SIMA (the partition operating system), PERC Pico (the Java Virtual Machine) and AIDA (Java APEX API, Logbooks middleware, Broker middleware and so on).

The lower part of the process, the partition level, is iterated over all partitions in the system. Note that for multi-static configurations, the process has to be additionally iterated, on platform level, over all configurations. This step is not shown in Figure 12 to keep the diagram readable.

4.4 JAVA ON ARINC 653 APEX

The porting of PERC Pico to the ARINC 653 APEX interface raises few difficulties concentrated on very few points, such as threading and priority inheritance, in particular [AD.5]. PERC Pico relies on a simple memory model, a set of annotations, and a powerful static verification tool. It compares advantageously to solutions based on *Scoped Memory*, a concept proposed by the Real-Time Specification for Java (RTSJ), in particular when modularity and runtime safety — or, the other way round, testing effort — are concerned.

PERC Pico supports the scheduling model proposed by the Safety-Critical Java Technology (SCJT), which is fixed priority pre-emptive scheduling with FIFO order within priority and with the priority ceiling emulation protocol as priority inversion control mechanism.

The implementation of this scheduling on top of APEX can be problematic. The main problem with APEX (and with many other operating system), is the absence of the priority ceiling emulation protocol for locks.

For making up this limitation and implementing correctly the Java scheduling model on top of operating systems such as APEX, PERC Pico does not use a one-to-one mapping for scheduling and consequently a Java thread is not equivalent to an APEX process. PERC Pico handles the scheduling of Java threads internally.

Fully controlling the scheduling of Java threads allows PERC Pico scheduler to guarantee a bounded execution time for every Java synchronisation mechanism (synchronized methods and wait/notify/notifyAll operations).

Automatic Garbage collection technologies are not considered certifiable for the moment since the memory heap, which is modified concurrently by multiple threads and the garbage collector itself, is too complex for static analysis.

The RTSJ specification has defined another allocation mechanism for Java programs based on scoped memory areas. The allocation and de-allocation time of objects inside RTSJ Scoped Memory areas can be deterministic. In spite of this the analysis needed to prove that the usage of scopes and the reference assignment to a given scoped object will not raise any runtime exception can be very difficult.

Concerning the allocation of scoped memory areas, the RTSJ specification just says that the area will not be allocated from the current memory area, leaving the programmer in doubt about the success of this operation, especially in case of memory fragmentation.

PERC Pico has adopted a more restrictive approach, in which every scope is allocated on the Java stack in a strict LIFO order. This way the memory fragmentation is avoided, and the maximum memory usage computation of a program is reduced to the computation of the maximum stack usage which is a tractable problem.

PERC Pico introduces a series of annotations that allow the programmer to specify in which context an object will be used. Instead of requiring the programmer to allocate every scope and to handle the scope change using the RTSJ API, PERC Pico automatically creates a local scope for every method and uses the programmer's annotations to determine where to safely allocate objects when a “new” operation is performed. The annotations do not prevent the code to run with any other Java VM.

During the DIANA project, PERC Pico was ported to VxWorks 653, PikeOS and SIMA. The porting activity revealed that the behaviour of the implementations are very similar — thanks to the compliance to the ARINC 653 standard — but nevertheless show some significant differences. Indeed, every standard leaves decisions to the implementation; this guarantees that (existing) systems with different design approaches may fruitfully compete implementing the standard. Concerning the selected platforms, there are differences in initialisation, configuration and the application of error recovery mechanisms. For the applications running on top of PERC Pico, there should be no visible difference in the behaviour – it is the main objective of Java in AIDA to hide those differences.

The ECS application, namely the Zone Controller and the Pack Controller, was ported to Java, using the PERC Pico memory annotations. The resulting code was compiled with the tool chains for SIMA and VxWorks and run on both systems without any code changes on application level. The behavior produced by the components on different platforms was the same and components hosted on different modules interoperated without any problem.

Since SIMA had already been tested against the Conformity Test specified in part 3 of ARINC 653 GMV and the DIANA project team were confident that the behaviour of the simulator would be very close to the behaviour of any ARINC 653 compliant RTOS. However, the porting of a Safety Critical Java VM was an excellent demonstration of the strict compliance of SIMA to the ARINC 653 standard. It demonstrated at the same time that the compliance was not paid with decreased flexibility. In the contrary, the simulator did not form any obstacles to the porting of the VM and its integration with the demonstrator applications.

4.5 AIDA LOGBOOKS

AIDA Logbooks are platform-wide service components that can be plugged into a system by means of configuration files (Service Definition Descriptors). From these descriptors, OS specific configuration files and glue code is generated and introduced into the target tool chain.

AIDA Logbooks provide services similar to the ARINC 653 Logbooks, e.g.:

- `CREATE_LOGBOOK`
- `WRITE_LOGBOOK`
- `READ_LOGBOOK`
- `GET_LOGBOOK_STATUS`

The difference between AIDA Logbooks and ARINC 653 Logbooks is location and scope: ARINC 653 Logbooks exist only in the context of a partition; a failure of the module on which this partition is hosted will also result in a failure of the logbook. If a backup instance of the application needs to continue to write the logbook, it must be ensured by the function developer that the logbook instances of the two application instances are written in parallel.

AIDA Logbooks are location transparent and may contain one or more instances. A service like `WRITE_LOGBOOK` will result in writing a message in all instances at the same time without the necessity for any further application activity.

The DIANA implementation of AIDA Logbooks uses ARINC 653 Logbooks and communication via ARINC 653 queuing ports to request write and read access to AIDA Logbooks. Each instance of an AIDA Logbook implements an ARINC 653 Logbook. When a user application invokes a service like `WRITE_LOGBOOK` a message is sent to all instances requesting to engrave the logbook entry folded into the message into the non-volatile memory. The logbook instances then perform a write operation on their ARINC 653 Logbook.

The client side communication is implemented in a middleware layer that provides the logbook services to the application in the same partition. Internally, these services are mere communication stubs that exchange messages with the logbook instances. AIDA applications may be coded in Java; AIDA Logbooks provide, hence, a Java language binding for this communication stubs.

Figure 13 shows an AIDA logbook system with three redundant instances and one user application using this logbook. The red lines around the components show partition and, for redundancy reasons, hardware boundaries. It is technically possible, of course, to host more than one replica of a logbook or one of the replicas and the user application together on the same computer; for redundancy reasons, this is not useful.

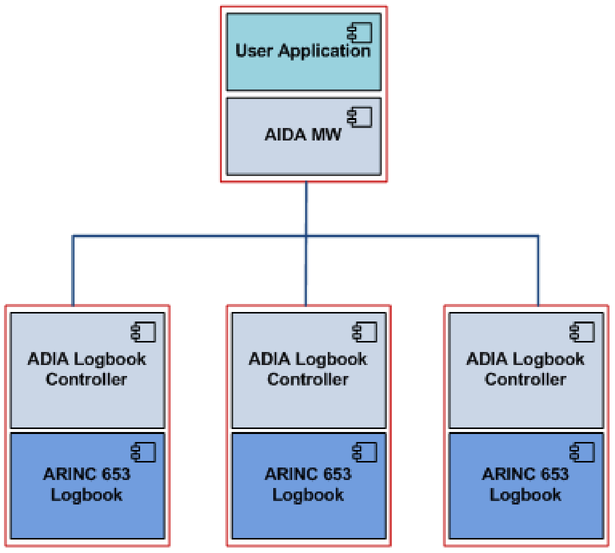


Figure 13: AIDA Logbooks

Most ARINC 653 compliant RTOS, including VxWorks 653, do not yet implement ARINC 653 logbooks. SIMA, however, does. The AIDA logbooks system was therefore implemented directly on top of SIMA. This limited the available modules available to host logbook servers to Linux/SIMA nodes. However, clients could still be hosted on any ARINC 653 compliant system.

ARINC 653 defines logbooks with a two-phase writing algorithm. When an application requests writing a message this message is first stored in a buffer in volatile memory. It is later written to the non-volatile storage medium. This way, the time necessary for engraving the message is not directly imposed on the calling process; instead, the implementation has to define a policy for the scheduling of writing messages.

SIMA uses a system partition to implement such a policy. The time, necessary for engraving the message to non-volatile memory is taken from scheduling windows of this partition. Moreover, the fact that the engraver is part of a separated partition eases the encapsulation of system-specific code.

The message buffer that temporarily holds the messages before they are engraved is implemented by a shared memory segment between the system partition and the application partition. The following figure illustrates the design:

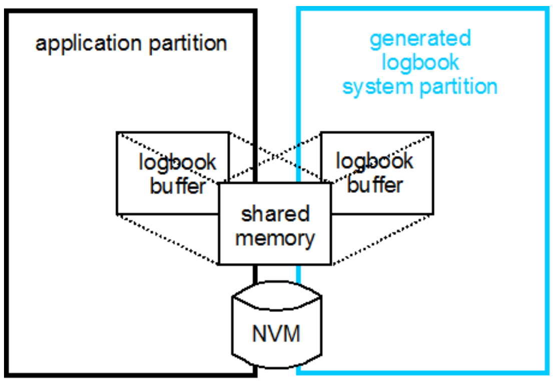


Figure 14: SIMA Logbook

In the ECS demonstrator a logbook for the pack controller, PACK_LOGBOOK, was implemented. The instances of the logbook were distributed on the two SIMA targets in the system. Note that the logbook instances can only run on SIMA, since VxWorks does not provide ARINC 653 Logbooks. However the logbook client can be used in both systems; it is, hence, possible to use the service even from a VxWorks system. To achieve this, it was not necessary to change any code between VxWorks and SIMA to provide the functionality to both systems.

The use of a simulator turned out to be extremely useful. As a research project, DIANA aimed at implementing new and experimental features that are, partly by nature, not available in COTS that respond to strict safety demands. As a simulator, SIMA is not expected to fulfil real on-board safety requirements. It is, hence, much easier and less expensive to implement advanced features like the ARINC 653 extended services or entirely new services proposed by research activities.

4.6 MULTI-STATIC RECONFIGURATION

AIDA extends IMA by supporting a first and limited, yet extensible, level of reconfiguration. To avoid a growth of software complexity beyond acceptable limits (in particular, in terms of certification effort), reconfiguration capabilities are actually restricted: At start-up, an AIDA compliant system selects autonomously the configuration that matches the system's health state among a pre-defined and pre-qualified set of configurations. This approach is called multi-static reconfiguration.

The first step of the algorithm, that takes place during the *definition phase*, determines the current health state of the system. In order to do so, all modules in the same reconfiguration domain exchange their private health state, represented by the result of the power-up built-in test (PBIT). To ensure, all modules will finally agree on the same system health state, a Byzantine Agreement Protocol is used [AD.4].

The second step of the algorithm is to apply a configuration that corresponds to the system health state. This is achieved by a predefined mapping of health states to possible configuration. If there is a configuration that maps the system health state and this configuration is not the currently selected one, the system sets this new configuration. Setting a configuration is basically done, by changing the entry point to the configuration. The entry point is a file that defines which binaries and configuration data should be loaded at boot time. When the configuration has been set the system is rebooted and the two steps of the algorithms are repeated. If there was no new failure in the system, the algorithm shall deduct the same configuration as in the first run and, hence, the current configuration.

If the configuration that results from the algorithm is identical to the current configuration the system leaves the definition phase and proceeds to the *operation phase*. If there is no configuration that maps the current system health state the module is passivated.

The heterogeneity of the systems had to be taken into account in the design, coding and parameterisation of the multi-static reconfiguration. First, an overall timeout must be found for all modules. To achieve this, the algorithm was benchmarked on the different target systems. One of the problems was the start-up procedure. No synchronised start-up had been defined for the demonstrators and, even worse, different start-up scenarios – for demonstration to an audience and for benchmarking in the lab – had been identified. Therefore, different tolerance delays, between two seconds and two minutes, and overall timeouts, between twenty seconds and three minutes, were chosen for different demonstration purposes.

Another issue that must be solved is the module reset and passivation mechanism. For RESET, the ARINC 653 health monitor was used on VxWorks and VxSim: An application error is raised by the reconfiguration engine that is not handled within the partition and, hence, propagated to the partition health monitor where RESET was defined as the corresponding error response action.

On SIMA, a system-specific *shutdown* service is available that is defined as follows:

```
procedure apx_shutdown
   (MODE        : in SHUTDOWN_MODE_TYPE;
    RETURN_CODE : out RETURN_CODE_TYPE) is
   error
          when (current partition is not allowed to issue this
                command) =>
                RETURN_CODE := INVALID_CONFIG;
          when (MODE does not identify a valid shutdown mode) =>
                RETURN_CODE := INVALID_PARAMETER;
   normal
          if (MODE is APX_SHUTDOWN_HALT) then
                stop module;
          else if (MODE is APX_SHUTDOWN_RESET) then
                reboot module;
          end if;
          RETURN_CODE := NO_ERROR;
end apx_shutdown;
```

In spite of being a SIMA-specific interface, the service is defined in the style of ARINC 653 services. It takes two arguments: the *MODE* and the *RETURN_CODE*. As in almost all ARINC 653 services, the *RETURN_CODE* is used to pass error information back to the caller. Possible errors are *INVALID_CONFIG, INVALID_PARAMETER* and *NO_ERROR*. The MODE parameter determines which of two possible actions shall be applied: *HALT* or *RESET*. On SIMA, the `apx_shutdown` service with *RESET* mode was used to implement the reset action.

Concerning passivation, the *shutdown* service was, again, the natural choice on SIMA. On VxWorks 653, however, where no such functionality is available, the Multiple Module Schedules service was exploited instead. Instead of shutting down the system, the reconfiguration engine requests to switch to an empty module schedule. The module continues to work, but no application is ever scheduled.

The system-specific code was encapsulated in a system partition that answers service requests by the reconfiguration engine. There is one generic system partition per module, implementing also other system-specific services that may be requested by applications, *e.g.* ARINC 653 logbooks. This way, the overhead, introduced by the reconfiguration approach, was kept to a minimum.

The reconfiguration engine is hosted on one partition per module. This partition is connected to the reconfiguration engines on other modules by queuing ports that implement the channels of the Byzantine Agreement protocol. This is depicted below:

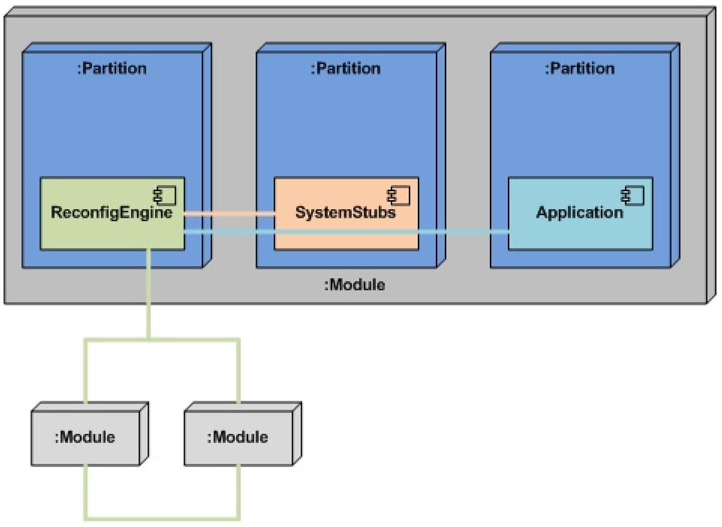


Figure 15: IMA Reconfiguration Engine

Again, the Multiple Module Schedules service was used to keep the overhead as small as possible: After a successful completion of the algorithm, i.e., when at the end of the algorithm the new configuration is equal to the current configuration, a switch to a schedule is requested that does not contain the execution windows for the reconfiguration engine anymore. In consequence, the reconfiguration engine will not consume any time resources after the system has entered operational phase.

For the ECS demonstrator, three configuration scenarios were defined:

- Configuration C0 is the basic configuration with all hardware nodes available;
- Configuration C1 is a degraded configuration with the VxWorks node failing;
- Configuration C2 is the degraded configuration with one of the SIMA nodes failing.

Figure 16 illustrates this approach:

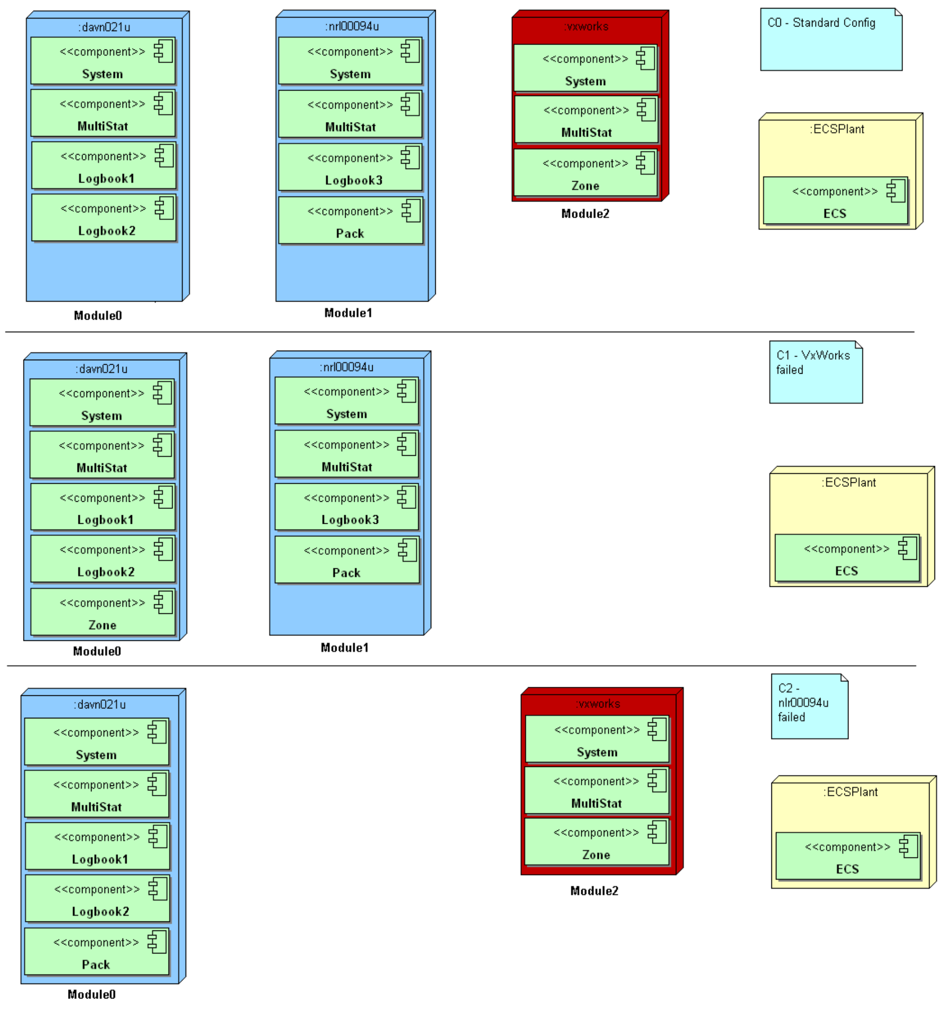


Figure 16: Reconfiguration Scenario for ECS Demonstrator

In Figure 16, modules depicted in blue are Intel desktop PCs running Linux with the SIMA environment. Modules depicted in red are PPC-based computers running VxWorks.

In configuration C0, Module0 serves as spare for the pack controller hosted on Module1. Additionally, it hosts two logbook instances of the `PACK_LOGBOOK`.[8] A third instance of the logbook is hosted together with the primary pack controller on Module1. Module2, the PPC system, hosts the zone controller.

All modules have a system partition and a partition for the reconfiguration engine (*MultiStat*). The system partitions implement all system specific code, needed for the reconfiguration engine. The system partitions on Module0 and Module1 additionally drive the logbook instances.

In configuration C1, Module2 has failed. The zone controller is now hosted on Module0. In configuration C2, Module1 has failed. The plant controller is now running on Module0.

[8] Note that hosting two instances of the same logbook on the same hardware node is of course not useful in terms of redundancy. This architecture was chosen, because there were only two computers in the demonstrator, running SIMA.

Note that a possible third configuration with Module0 failing is not interesting, since none of the components on Module0 can be hosted on one of the other modules in a meaningful way.

The ECS Plant is a single-partition SIMA application that runs on the Concurrent system. It acts as a communication bridge between the EuroSim environment simulation [AD.9] and the ECS application. Since it is not an AIDA component, it was not integrated in the reconfiguration approach. Instead, the communication lines to zone and pack controller were duplicated and the correct line was selected, depending on the current physical location of these components.

All modules, the PPC module with VxWorks as well as the SIMA system on Linux, performed the reconfiguration algorithm without any problem. In a homogeneous environment (*i.e*. SIMA only), the whole reconfiguration algorithm ran within less than two seconds (worst case). In a heterogeneous system, the synchronisation issues described above had to be taken into account. This resulted in a longer definition phases, up to a minute for the lab benchmarks.

5 THE DEMONSTRATOR AT AVIONICS 2010

To complete the demonstrator, the displays were integrated into NLR's APERO flight simulator. APERO, Avionics Prototyping Environment for Research and Operations, is a fixed base research flight simulator built to provide a flexible avionics prototyping and cockpit simulation system.

This completely integrated demonstrator was presented at the exhibition of the Avionics Event 2010 in Amsterdam. The APERO flight simulator was configured for an Airbus A320 cockpit. The ECS control panels were integrated into the cockpit. People using the flight simulator could change the cabin temperature and control the effect by means of the EuroSim [AD.9] output displayed on a screen next to the flight simulator (on the left hand side of **Figure 17**):

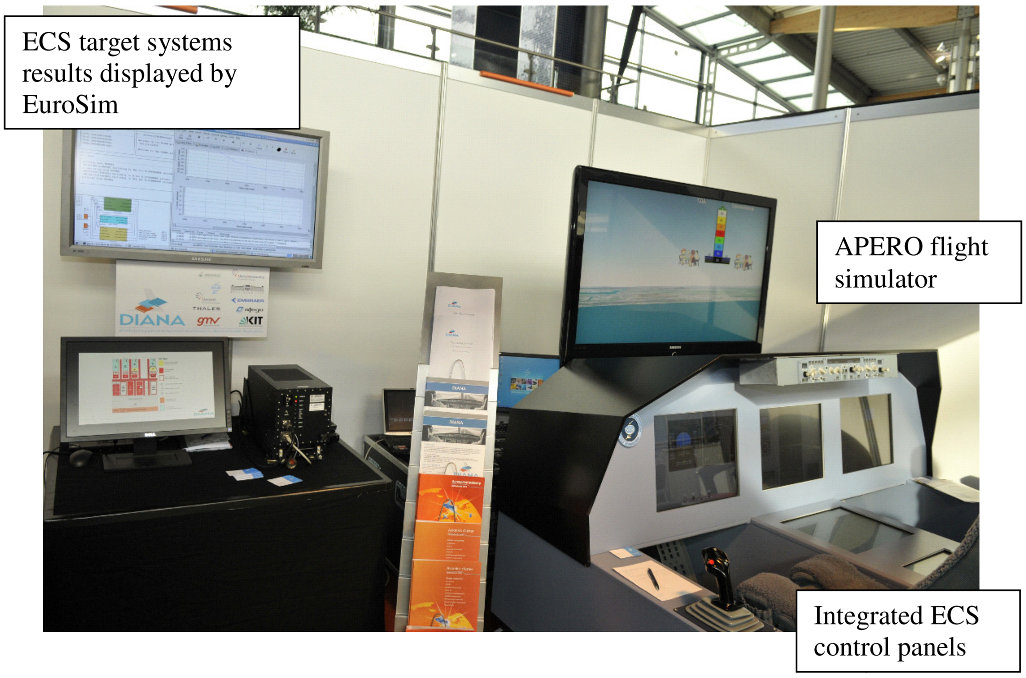

ECS target systems results displayed by EuroSim
APERO flight simulator
Integrated ECS control panels

Figure 17: ECS Demonstrator at Avionics 2010[9]

On the table on the left hand side of Figure 17, the on-board computer, running VxWorks is visible. The PCs running the SIMA systems are not visible in the picture. However, the *simout* output is displayed on the screen to give a visual impression of the partitioning concept.

The demonstrator ran for several hours per day with this set-up. No failures related to the operating systems, simulators or AIDA components occurred. The DIANA project team had sufficient confidence in the system to demonstrate it with this heterogeneous environment. This confidence was mainly inspired by the quality of COTS components that were used to built the demonstrator.

[9] © NLR, 2010, all rights reserved

6 CONCLUSIONS

In the scope of the DIANA project, SIMA turned out to be extremely useful. SIMA ran real-world avionics applications for several hours and days without interruption, without errors or memory leaks and even without deadline misses. The use in the DIANA project shows impressively that the tool is stable and robust, both in the functional domain and in timeliness.

SIMA also turned out to be an excellent tool for prototyping applications. The SIMA tool chain is extremely easy to use compared to real-target systems. Application code was developed, integrated and tested within hours. This advantage was exploited during the porting of the Java VM, PERC Pico, to the ARINC 653 APEX and the prototyping of the initial C-code of ECS application components.

The low effort and, hence, low cost of the development for the SIMA environment, enabled the project team to prototype and compare different designs. This way, the project achieved remarkable high quality of software components, in particular the APEX version of PERC Pico, the AIDA Logbook system, the AIDA Reconfiguration Engine and, of course, the ECS application.

Engineers, not familiar with the ARINC 653 APEX, benefited from SIMA's easy-to-use, yet realistic tool chain. Also, the good quality of documentation and sample code eases studying the behaviour of the ARINC 653 services in detail.

An important factor in the project was SIMA's proven compliance to ARINC 653. Like VxWorks 653 and PikeOS, SIMA underwent a conformity test, using GMV's ARINC 653 Validation Testsuite (AVT), the reference implementation of the ARINC 653 Part 3. The project could, hence, rely on the fact that the real-target RTOS, VxWorks 653, and the SIMA simulator on Linux would produce the same functional behaviour.

The porting of a complex environment such as a complete Safety Critical Java VM to the ARINC 653 APEX, using SIMA and real-target RTOS, was an excellent demonstration of the strict compliance of SIMA to the ARINC 653 standard. It demonstrated at the same time that the compliance was not paid with lack of flexibility. In the contrary, the simulator did not form any obstacles to the porting of the VM and its integration with the demonstrator applications.

As a research project, DIANA aimed at implementing new and experimental features that are not available in real-target systems. Certifiable RTOS have to fulfil extremely demanding safety requirements. This makes the development of new, experimental and potentially complex features difficult and costly. As a simulator, SIMA is not expected to respond to real safety challenges. It is, hence, much easier and less expensive to implement advanced features like the ARINC 653 extended services or to integrate entirely new concepts proposed by research activities. A simulator like SIMA is, hence, a valuable means in particular for aeronautical research programmes.