Verifying, Validating and Monitoring the Open Ravenscar Real Time Kernel[1]

R. Maia, F. Moreira, R. Barbosa, D. Costa
Critical Software, SA
Coimbra, Portugal
rmaia@criticalsoftware.com

Patricia Rodriguez
SoftWcare, S.L.
Vigo, Spain
rodriguezdapena@softwcare.com

Kjeld Hjortnaes
ESTEC/ESA
Noordwijk, The Netherlands
Kjeld.Hjortnaes@esa.int

Luis Miguel Pinho
Polytecnique Institute of Porto
Porto, Portugal
lpinho@dei.isep.ipp.pt

Abstract

Business and mission critical real-time systems need to be fully predictable, in order that their behaviour is known before deployment, even in the presence of faults. The Open Ravenscar Real Time Kernel (ORK) is a small size with reduced complexity kernel designed to be used in this type of applications. This kernel was implemented to be fully compliant with the Ravenscar profile, which defines a subset of the tasking features of Ada which can be used to implement a small and reliable kernel. Critical Software has recently conducted an evaluation on this kernel as well as started a new project to provide a monitoring tool for ORK, thus further enhancing the already existing range of hard real-time verification and validation tools. This paper summarizes the objectives and results of the ORK evaluation and presents the main goals and functionalities of the monitoring tool.

1 Introduction

The consequences of failures in hard real-time computer systems are, unfortunately, well known. Examples of these are the Mars Pathfinder mission that began experiencing total system resets, which resulted in losses of data due to a priority inversion problem [1], and, particularly important due to the severe nature of the consequences, the Therac-25 medical system, that caused the lost of three lives due to radiation over dosage [2]. Therefore, computer systems, and in particular hard real-time systems, must be subject to intensive testing activities before their deployment. However, assuring that these systems behave appropriately to unusual or exceptional events requires more than traditional testing. Furthermore, when safety critical software products are concerned, characteristics other than functional, such as safety and reliability should also be expressly verified.

Critical Software in a consortium with SoftWcare is presently conducting a project (STADY) that aims to research and demonstrate an innovative technique for the verification of the safety and reliability characteristics of software. As part of this study an evaluation of the Open Ravenscar Real-Time Kernel (ORK) [3] was performed.

Within a different approach, a monitoring and profiling tool for ORK is also being developed, which will allow the observation of the dynamic behaviour and results of an implemented system, consequently allowing the system's engineers to have a more in depth view of the system behaviour. This tool will enhance the already existing range of hard real-time verification and validation tools (Xception™ [4]), in order to non-intrusively monitor and profile business and mission critical real-time systems.

This paper is organized as follows. Section 2 presents an overview of the technologies involved in these projects, namely the ORK and the Ravenscar profile. Section 3 provides a description of the STADY project particularly considering the ORK evaluation, the problems that were found and potential

[1] This work is being supported by ESA/ESTEC (project STADY – Applied Static And Dynamic Verification Of Critical Software, Contract Nr. 15751/02/NL/LvH), by ADI (project BSCAN4FI, ref. ADI-POSI-BSCAN4FI) and by FCT (project TERRA, ref. POSI/2001/38932).

improvements. Section 4 provides an introduction to the monitoring and profiling tool, as well as an overview of the involved technologies. Finally, Section 5 provides some conclusions.

2 Open Ravenscar Real Time Kernel

The ORK [3] is an open source Real Time Operating System (RTOS) of small size and complexity, developed with sponsorship of ESA to be used in space based applications. ESA is assessing the use of this RTOS as a base layer of On-Board Software for future missions.

The ORK aims to be fully compliant with the Ravenscar profile [5], which defines a restricted set of Ada tasking features that are allowed and features that are disallowed.

2.1 Ravenscar profile

The Ravenscar profile (Reliable Ada Verifiable Executive Needed for Scheduling Critical Applications) is the result of the 8th International Real Time Ada Workshop (IRTAW). Since then it as already been revised at least two more times, namely at the 9th and 10th IRTAW. The goal of the Ravenscar profile is to achieve:

- Improved memory and execution time efficiency, by removing features with a high over head.
- Improved reliability, by removing non-deterministic and non-formally analysable features.
- Improved timing analysis, by removing non-deterministic and non-analysable features.
- Security certifiability for both the kernel and the software running on top of it.

A description of this profile can be found in [5] and the rationale in [6], but, in a nutshell, this profile forbids the use of task entries, dynamic allocation and unchecked deallocation of protected and task objects and dynamic priorities on programs. Although all these and more restrictions are defined in this profile, some other features are supported by it like ceiling locking protocol, FIFO within priorities dispatching and protected procedures as statically bound interrupt handlers.

Most of the profile's restrictions can be enforced at compile time by using the appropriate set of identifiers with the pragma restrictions available in the GNAT. However, since not all Ravenscar restrictions can be enforced by these standard restriction identifiers, another approach was adopted by ORK developers, which consisted in the implementation of a defined pragma, pragma Ravenscar, so it could establish the complete set of restrictions.

2.2 ORK Architecture

ORK was built on top of GNAT compilation system. The GNAT/ORK runtime system includes the following components which were subjected to the STADY methodology (see Figure 1):

- A specialized version of GNARL, the GNU Ada Runtime Library.
- A specialized version of GNULL, the GNU Low-Level Library.
- The ORK kernel itself.

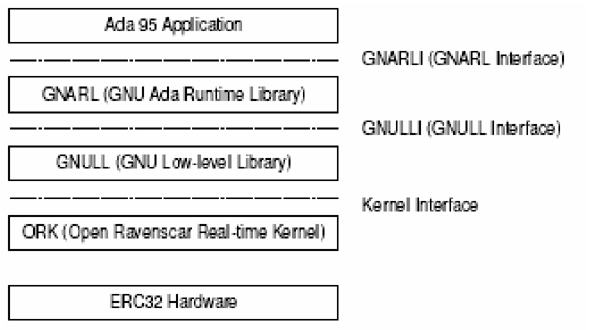

Ada 95 Application
GNARLI (GNARL Interface)
GNARL (GNU Ada Runtime Library)
GNULLI (GNULL Interface)
GNULL (GNU Low-level Library)
Kernel Interface
ORK (Open Ravenscar Real-time Kernel)
ERC32 Hardware

Figure 1. GNAT/ORK run-time system architecture.

The ORK Kernel Layer consists of the following packages:

- `Kernel` – which is the root package (empty interface)
- `Kernel.Threads` – this package provides the necessary thread management, including synchronization and scheduling control functions;
- `Kernel.Time` – this package provides clock and delay services;
- `Kernel.Memory` – this package provides the necessary functions for storage management;
- `Kernel.Interrupts` – this package provides the functions for Interrupt handling;
- `Kernel.Parameters` – this package contains the base parameters for kernel configuration;
- `Kernel.CPU_Primitives` – this package provides the processor dependent definitions and operations;

- `Kernel.Peripherals` – this package provides the support for peripherals in the target board;
- `Kernel.Peripherals.Registers` – this package contains the necessary definitions related to input-output registers of the peripheral devices; and
- `Kernel.Serial_Output` – this package provides the support for serial output to a console.

Although all these functionalities that are provided, they are not intended to be used directly from Ada programs. The interface to these functionalities is provided by the GNU Ada Runtime Library (GNARL). This library is used so that Ada95 tasking constructs can be directly accessed by the real-time application programmer.

3 ORK Evaluation

3.1 The Verification and Validation Method

The main goal of the STADY project [7] is to demonstrate that the combined use of static analysis techniques, like Software Failure mode and Effect Analysis (SFMEA) or Software Fault Tree Analysis (SFTA), with the dynamic analysis methods, like robustness/stress testing, is applicable to the verification of safety and reliability characteristics of critical software, in the domain of high integrity applications.

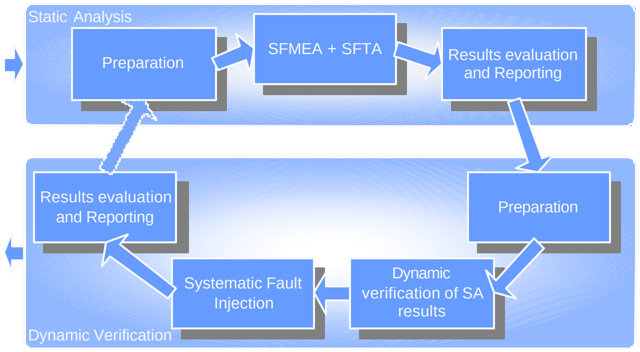

Static Analysis
Preparation
SFMEA + SFTA
Results evaluation and Reporting
Results evaluation and Reporting
Preparation
Systematic Fault Injection
Dynamic verification of SA results
Dynamic Verification

Figure 2. Workflow of the STADY method.

The run-time system of ORK has been selected as the product to be evaluated as a case study of the STADY project. The evaluation consists in a sequence of steps (Figure 2), beginning with a static analysis which results in a set of software faults with associated recommendations for improvements of the product under evaluation. This is followed by a dynamic verification which has two main purposes:

1. Confirm the set of software faults really comprise the system and evaluate the effect of the appropriate recommendations providing the feedback to the static analysis process;
2. Analyze the product dynamically (under execution) exercising the system by injecting software faults;

3.2 Evaluation Results

This section presents the most relevant ORK faults found during the application of the STADY method and provides some general comments for ORK runtime system improvement.

The following table provides a statistical view of the results achieved. Faults were grouped according to their types and classified as minor or critical. The classification is performed based on the possible impact of the fault concerning mission critical systems.

Fault Type	Minor	Critical	Total
Ravenscar profile violations		3	3
Missing Variable or Parameter Test Conditions		26	26
Incorrect Initialized variables		1	1
Use of variables not initialised		1	1
Performance	1		1
Dead or Unreachable code	7	10	17
Documentation faults	10		10
Total Faults	18	41	59

Table 1 - Number of faults found according to type and criticality.

3.3 Ravenscar Profile violations

3.3.1 Tasks do not terminate

The Ravenscar profile requires that tasks do not terminate (*RP30 – Dynamic Semantics*). ORK Software Requirements Specification [8] states that the default action of invoking a task termination is to raise a Program_Error exception (*RQ4 – Effect of task termination*). What was found during this study was that the implementation does not obey the requirements and the default task termination action is to remove the thread's descriptor from the running queue when the execution falls back to the Kernel.

Did ORK developers judge this not to be the best termination action and implemented in a different way?

By forcing the default task termination to be the raising of a program error, the whole application will stop when a single task terminates. The question raised here is: "Is the raising of an exception the best approach?" Other options might be considered, e.g.

putting tasks that invoke termination in a wait queue permanently.

3.3.2 Use of forbidden features

Several features not allowed by the Ravenscar profile are provided by the kernel and can be called from the user application level. A few examples are described below.

The Ravenscar profile specifically states that dynamic priorities are not allowed (*RP7 – Forbidden features - Dynamic Priorities*) but a user task can change its own Base_Priority using either kernel level procedure `Kernel_Threads.Set_Priority` or `System.Task_Primitives.Operations.Set_Priorty`.

The `Kernel.Memory.Protect_Segment` procedure used by the kernel during the initialisation can be called by user level applications violating the Ravenscar profile (*RP50 – Static_Storage_Size*).

Dynamic creation of threads is possible contrary to what is stated by requirement *RQ1: Task creation* and requirement *RQ2: Static tasks only*. This can be achieved by calling one of the following procedures:

```
Kernel.Threads.Thread_Create

System.Task_Primitives.Operations.Create_Task
```

The restriction of forbidden features should be enforced at compilation time not allowing these features to be used at the application level.

3.4 Missing Variable or Parameter Test Conditions

During the analysis some cases were found where checks for invalid/unexpected parameters values are missing, e.g. null pointers. The systematic check to overcome the usage of these invalid/unexpected values (defensive programming) is a good practice that should be enforced. A few examples are described below.

3.4.1 Null Parameters Check

In procedure `Kernel.Threads.Thread_Create` no check for null pointer is done regarding parameter code. If a null pointer is given as argument to `Thread_Create` procedure, then when the new created thread is schedule the system hangs.

No check for null pointer is made in procedure `System.Task_Primitives.Operations.Create_Task` regarding to parameter wrapper. If a null pointer is given as an argument then when the new created thread is scheduled to run the system hangs.

3.4.2 Return value check

In procedure `Thread_Create` in the `Kernel.Threads` package after getting the first free position to the new thread:

```
Id := Queues.Get_New_Thread_Descriptor;
```

if the `Id` variable is `Null_Thread_Id`, it should not be reserved any space for the stack. Stack space should only be reserved if a Thread descriptor is successfully obtained.

3.4.3 Range validity

Since GNAT/ORK is compiled with the runtime constraint checks disabled, it is imperative that simple range checking is done on functions and procedures parameters (defensive programming). The recommendation is that GNAT/ORK specify clearly the policy to be followed regarding to the verification of input data. Either user applications are required to validate the inputs before performing API calls or GNAT/ORK shall validate the parameters provided with the user application calls. In the latter case the layer at which the inputs shall be verified shall be specified. Otherwise out of range values passed to internal functions might lead to unexpected results.

Parameter `Priority` is not being checked, thus it is possible create a thread with any Integer value for its priority (instead of a value fitting in the defined ORK priority range).

Integer values (including negative values) values are accepted for `pragma Priority` when defining a task. Priorities are defined as an Integer value ranging from 0 to 255 in the settings used.

No checks in priority parameter value are made in procedure `Mutex_Init` and the `mutex` is effectively created with any value passed.

No checks are made in procedure `New_Stack`. The new stack is created with any value passed. Tests revealed that a negative number really forces the stack index to go back.

In case procedure `Mutex_Unlock` in `Kernel.Threads` is not well used (e.g. it is called before first acquiring the lock), the `Lock_Nesting_Level` field of the ATCB running task (Natural type) with a 0 value can be decremented causing unexpected results. For instance, tests concluded that task/thread can actually change its `Active_Priority` misusing procedure `Mutex_Unlock`.

In procedure `Protect_Segment` if base or ending input parameters are not greater than `16#2000000#`, after these operations:

```
Base_Address_Aux := (Base_Address_Aux -
                      16#2000000#) / 4;

End_Address_Aux := (End_Address_Aux -
                     16#2000000#) / 4;
```

considering that type `Segment_Address` has the range:

```
type Segment_Address is range 0 .. 2**23-1;
```

the following type conversions might cause unexpected results:

```
Base_Register_Aux.SEGBASE :=
     KPR.Segment_Address (Base_Address_Aux);

End_Register_Aux.SEGEND :=
     KPR.Segment_Address (End_Address_Aux);
```

During test cases execution it was verified that if procedure `Protect_Segment` is called with parameter base equal to `16#200_0000#`, regardless of parameter ending's value, the application hangs and the processor entered in HALT mode. There is a comment in the source code stating that this procedure is to be used only during the initialization and that the user should not be able to access this procedure. If this is correct, then it should be enforced by design or compilation time.

3.4.4 Unchecked conversions

To prevent the truncation effect or any other kind of type conversion problems, the use of instantiations of the `Unchecked_Conversion` generic function should be avoided.

In Procedure `Initialize_Protection_Entry` in `System.Tasking.Protected_Objects.Single_Entry`, variable `Init_Priority` is defined as Integer. The Ceiling field in the Object record is Integer range `0 .. Standard'Max_Interrupt_Priority`.

Therefore, a type conversion in next sentence may cause unexpected results:

```
Object.Ceiling :=
     System.Any_Priority (Init_Priority);
```

In Procedure `Create_Restricted_Task` in `System.Tasking.Restricted.Stages` the parameter Priority is Integer and `System.Any_Priority` is an Integer subtype starting at 0. Unexpected results may happen in next sentence:

```
Base_Priority :=
     System.Any_Priority (Priority)
```

In Procedure Task_Wrapper in `System.Tasking.Restricted.Stages` the following function call is made:

```
SST.SS_Init (Secondary_Stack_Address,
               Integer (Secondary_Stack'Last));
```

Since variable `Secondary_Stack` is an array defined as:

```
Secondary_Stack : aliased SSE.Storage_Array
  1..SSE.Storage_Offset(ID.Common.Stack_Size *
          Parameters.Sec_Stack_Ratio / 100));
```

If `Parameters.Sec_Stack_Ratio` is non-natural value then the `Secondary_Stack` variable will be an empty array (without any elements) which means that the thread might not have any stack at all. Also, unexpected results may occur from the call:

```
SST.SS_Init (Secondary_Stack_Address,
               Integer (Secondary_Stack'Last);
```

3.5 Considerations

Several problems were uncovered during this evaluation of the ORK. The most serious problems found are related with the access to the ORK Kernel.

The ORK Kernel layer is a very sensitive part of the system (as it would be expected) and it does not include any parameter validation. As demonstrated by the results, it is very simple to inadvertently crash the whole system or compromise Ravenscar profile restrictions.

In our opinion some of the problems found during the evaluation, and addressed in this paper, should be carefully analyzed/corrected before using ORK in a specific mission.

4 Non-Intrusive Monitoring

One of the fundamental aspects of software testing is the capability of non-intrusive observation. The goal of the Xpy project is to develop a non-intrusive application of monitoring and profiling using the technological solutions developed in a currently undergoing project (Exploiting IEEE 1149.1 Boundary Scan for Fault Injection), with specific functionalities for the SPARC/ERC32 [9] and ORK target system.

4.1 SPARC/ERC32

The SPARC/ERC32 is a processor developed in the scope of the European Space Agency (ESA) space based applications. This processor is being developed using Radiation Hard technology, and it contains appropriate fault tolerance mechanisms. The SPARC/ERC32 has a hardware module for application support (OCD – On-Chip Debugger) that is accessible through Boundary Scan.

4.2 BSCAN4FI

One of the projects currently being developed in Critical Software, the BSCAN4FI (Exploiting IEEE 1149.1 Boundary Scan for Fault Injection), aims to develop new technologic solutions for advanced critical systems testing, which involve fault injection. The platform used in this project is the SPARC/ERC32 processor. Testing and fault injection are made by accessing the processor through boundary scan technology. The OCD is also used as a support mechanism for testing.

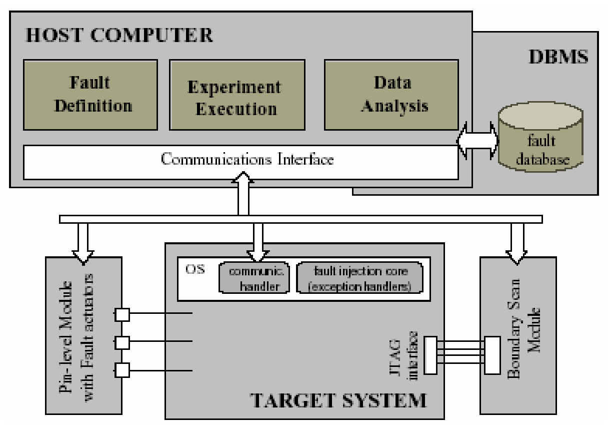


Figure 3. Xception Architecture.

4.3 XCEPTION™

The *Xpy* tool is being developed to integrate an already existing fault injection line of products in Critical Software, namely the Xception™ [4].

Xception™ is an automated fault injection environment that enables accurate and flexible V&V (verification & validation) and evaluation of mission and business critical computer systems using fault injection. Xception is designed to accommodate a variety of fault injection techniques (according to a wide range of configurations of the tool) and emulates in this way different classes of faults, with particular emphasis to hardware and software faults. One key aspect of Xception is the high degree of automation provided by the fault injection environment, which enables the users to plan and perform fault injection experiments in a straightforward way. Figure 3 shows the Xception's architecture.

4.4 Xpy in Xception

Xception is composed of two software components, namely the Experiment Management Environment (EME) and the EME Proxy. EME is the graphical user interface of Xception™, allowing the user to define, execute and control experiments and perform result analysis. EME Proxy is the middleware software component that handles the communications within Xception. For each target system, a target specific plug-in must be implemented, allowing. Xception™ to support different target systems.

Although this product is being developed to integrate the Xception™ line of products in Critical Software, there is the goal to provide a general-purpose tool for monitoring and profiling. Therefore, the tool will be added to Xception using one plug-ins (Figure 4).

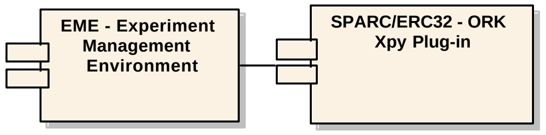

EME - Experiment Management Environment
SPARC/ERC32 - ORK Xpy Plug-in

Figure 4. Xpy in Xception EME

4.5 Xpy Requirements

This project is being developed assuming that the target environment is based on the SPARC/ERC32 processor. Some characteristics that will be designed and implemented will be particularly related for this processor. Particularly, the access will be made through Boundary Scan and the Test Access Port, and the On-Chip Debugger is also planned to be used as a support mechanism.

It is also assumed that the base kernel for which this application is being developed is the ORK. Although this assumption is being made, the choice is still under discussion. Therefore, although currently being developed to operate in a specific target environment (composed by the SPARC/ERC32 processor running the ORK real time kernel), it is the goal to maintain its kernel independence as much as possible.

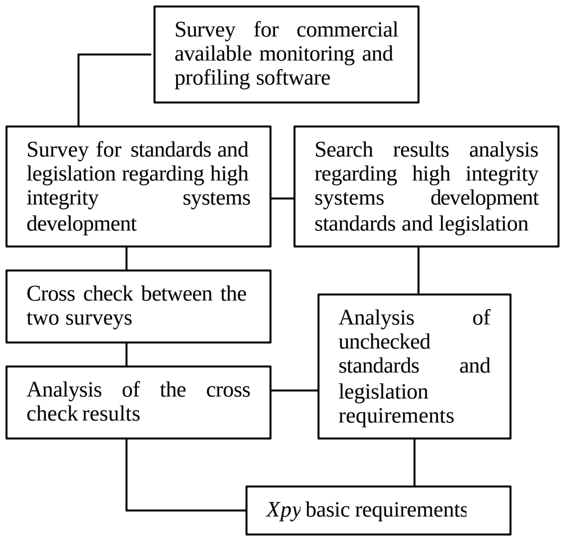
Due to the requirements imposed by the target, *Xpy* is also currently being developed as an instrument of analysis compliant with the "ISO – Guide for the Use of the Ada Programming Language in High Integrity Systems" [10] and the "Guide for the use of the Ada Ravenscar Profile in High Integrity Systems" [6].

4.6 Requirements Process

In order to get the highest quality, reliability and usability for this application, the basic requirements went by a rigorous selection criterion. Figure 3 shows the selection process used.

The process was started by a search for commercially available profiling and monitoring applications and applications with similar characteristics. Another search was made for standards and legislation regarding high integrity hard real time application development. From these searches the more important and relevant results were select and analyzed. After a cross check between the two searches the first *Xpy* requirements were found. Afterwards an analysis was made to the uncross-checked requirements between the standards and legislation survey and the commercial survey, and from that analysis, some requirements that are until know commercially unimplemented were added to the project.

With this process we were able to select the most important requirements in terms of usability for hard real time system analysis.

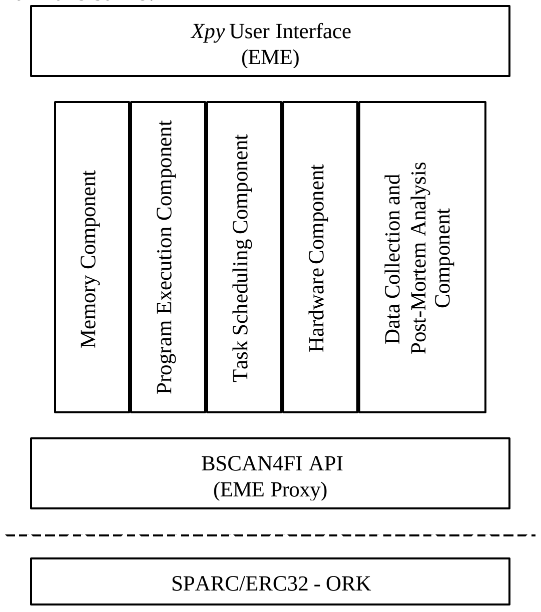


Survey for commercial available monitoring and profiling software
Survey for standards and legislation regarding high integrity systems development
Search results analysis regarding high integrity systems development standards and legislation
Cross check between the two surveys
Analysis of unchecked standards and legislation requirements
Analysis of the cross check results
Xpy basic requirements

Figure 5. Requirements Selection Process

4.7 Xpy Architecture and Functionalities

Figure 6 presents the foreseen architecture of *Xpy*. This architecture is still a work in progress. Some changes may occur but the basic layout will in general remain the same.



Xpy User Interface (EME)
Memory Component
Program Execution Component
Task Scheduling Component
Hardware Component
Data Collection and Post-Mortem Analysis Component
BSCAN4FI API (EME Proxy)
SPARC/ERC32 - ORK

Figure 6. Xpy Architecture

The *Xpy* is divided into five groups of monitoring and profiling (M/P) features, each one with specific characteristics:

- *Memory M/P Features*: such as Kernel memory usage, per task memory usage, contents of the Execution Stack and the Interrupt Stack, …;
- *Program Execution M/P Features*: Code sequence, ready and wait Queues, what code is executing in the processor and if it is either kernel or user code, …;
- *Task Scheduling M/P Features*: Graphical tasks scheduling, worst case measured times, timing forecasts of sporadic tasks, ...;
- *Hardware M/P Features*: Processor usage, context switches, processor resources (UARTs, GPI, Interrupt controller) usage, ...;
- *Data Collection and Post-Mortem Analysis M/P Features*: collect and store data on predefined events, both default and user defined data;

Currently, the *Xpy* tool is on the early stages of development. The architecture and the system design are almost complete.

5 Conclusions

It is well known that the testing of critical real-time systems is an extremely important procedure in order to assure the safety and reliability requirements imposed by the system environment. In this paper two different approaches for testing the ORK were presented.

In the first approach, and evaluation of the ORK itself was presented, which allowed detecting the possibility of misbehaviour both in the Kernel itself and at the application level. This paper provided a summary of this approach, and of the evaluation results.

The second approach considers that in order to monitor and profile the ORK together with the application, a non-intrusive tool is an important requirement. This paper provided also an overview of such tool, which is currently being developed.

References

[1] Mike Jones, "What really happened on Mars Rover Pathfinder", Risks Digest, December 1997.

[2] N. Levenson, C. Turner. "An Investigation of the Therac-25 Accidents", IEEE Computer, Vol.26, No.7, July 1993.

[3] Juan A. de la Puente, José F. Ruiz. "Open Ravenscar Real-Time Kernel – Operations Manual", March 2001. www.openravenscar.org

[4] R.Maia, L. Henriques, D. Costa. “Xception™ - Enhanced Automated Fault-Injection Environment”, 2002. www.xception.org

[5] B. Dobbing, A. Burns. “The Ravenscar Tasking Profile for High Integrity Real-Time Programs”. In Proc. of SIGAda’98 Conference, Washington, USA, pp. 1-6.

[6] A. Burns, B. Dobbing, T. Vardanega. “Guide for the use of the Ada Ravenscar Profile in High Integrity Systems”, Technical Report Nr.YCS-2003-348, January 2003.

[7] ESA Contract 15751/02/NL/LvH. “Applied Static and Dynamic Verification of Critical Software”. ESA, March 2002

[8] Juan A. de la Puente, José F. Ruiz. “Open Ravenscar Real-Time Kernel - Software Requirements Specification”, version 1.6, 05.05.2000, UPM.

[9] Atmel, “TSC695F Rad-Hard Embedded Processor 32-bit SPARC User’s Manual”, March 2001. www.temic-semi.com

[10] ISO/IEC JTC1/SC22/WG9, “Guide for the use of the Ada Programming Language in High Integrity Systems”, July 1999.