# Guide to the Industrial Sessions

15th International Conference on Reliable Software Technologies
Ada-Europe 2010

**CONTENTS**

Abstracts and schedule of Industrial Presentations

# Contents and Schedule

# HRT-UML and Ada Ravenscar Profile: A Methodological Approach to the Design Of Level-B Spacecraft Software

Roberto López, Ana Isabel Rodríguez
roblopez@gmv.com, airodriguez@gmv.com
GMV, Spain

This presentation will provide feedback on the use of Hard Real-Time Unified Modeling Language (HRT-UML) and Ravenscar Profile in the design definition of the Instrument Control Module (ICM) of Ocean & Land Color Instrument (OLCI), developed in the context of Sentinel-3 satellite. The ICM software, build around an ERC32 microprocessor, is a critical software (ECSS-E-ST-40C critical level B) responsible for interfacing with the satellite central computer (Satellite Management Unit), as well as controlling the rest of instrument units. The aim of this paper is discussing the advantages and disadvantages of using this approach in a real on-board critical software development.

HRT-UML method aims to provide a comprehensive solution to the modeling of Hard Real Time systems. Its goal is to define a customized version of UML to express the HRT-HOOD methodology, making the most of both standards and also capturing and compensating for the respective weaknesses. The resulting design method permits static scheduling analysis of the system and also caters for automated generation of Ada 95 code that complies with the Ravenscar Profile. The only supporting toolset, provided by Intecs s.p.a., has been used in our development.

The Ravenscar Profile is a subset of the Ada tasking model, restricted to meet the real-time community requirements for determinism, schedulability analysis and memory-boundedness, as well as being suitable for mapping to a small and efficient run-time system that supports task synchronization and communication, and which could be certifiable to the highest integrity levels. The concurrency model promoted by the Ravenscar Profile is consistent with the use of tools that allow the static properties of programs to be verified. In this development, we have used the High Integrity Ravenscar Run Time for ERC32 (GNAT Pro for ERC32), provided by AdaCore.

The synergy between both technologies, which in fact has been the main reason for selecting this approach, is the possibility of using static verification techniques such as schedulability analysis and model checking. These techniques allow analysis of a system to be performed throughout its development life cycle, thus avoiding the common problem of finding only during system integration and testing that the design fails to meet its non-functional requirements.

By complying Ravenscar Profile tasking model (automatically checked at model level) and estimating the timing requirements of each task and protected object (Period, WCET, Deadline and WCET of each protected method), HRT-UML allows performing assignment of fixed priorities to the different tasks and protected objects according to Ravenscar Profile scheduling model, schedulability analysis based on the previous assignment, and CPU load analysis. WCET can be estimated at first stages of the design and refined in subsequent phases, making possible to check the non- functional timing requirements along the whole development life cycle.

On the other side, some difficulties have been faced when using the proposed approach. Some of them are related to the HRT-UML methodology itself, and some others can be understood as improvements to the HRT-UML supporting toolset. The following list summarizes the proposed improvements:

- HRT-UML model restrictions to comply Ravenscar Profile seem to be more restrictive than the profile itself. The constraint that forbids a passive object to use a non-passive object is a HRT-UML constraint not derived from the Ravenscar profile. We understand that the aim of this constraint is making easy the automatic schedulability analysis, as it is simplest way to make a map of the protected objects being used by each task. This HRT-UML restriction makes difficult the design and probably could be solved by providing a more complex algorithm to analyze relationships between objects.
- The difference between classes and types is not clearly understood. From UML point of view, an Ada type is exactly the same UML class concept.
- The way the types are managed by HRT-UML tool makes difficult the maintenance of the design in large systems. The same maintainability problem can be observed at code level if the provided code generator is used.
- Toolset does not allow other UML diagrams necessary to complete the design, such as use cases, sequence diagrams, etc. The only supported diagrams are HRT-UML object diagrams.

Finally, an alternative to the selected Ravenscar run time system (GNAT Pro High Integrity Ravenscar Run Time for ERC32) has been assessed in the context of this project. It must be taken into account that the Sentinel 3 OLCI ICM software, including the run-time system, needs to fulfill the criticality requirement (ECSS-E-ST-40C critical level B). The reason for evaluating an alternative to the selected run time has been trying to mitigate the impact of the GNAT Pro High Integrity Ravenscar Run Time for ERC32 qualification risk, which is currently being performed by Ada Core. The proposed alternative is based on the use of RTEMS, which is also being qualified to level B by RTEMS Centre. The solution is possible because the Ravenscar restrictions can be reproduced on top of RTEMS, which is providing the same scheduling model (pre-emptive fixed priority scheduling and priority ceiling protocol when accessing to shared sections). However, it is not possible to use the RTEMS Ada API for two reasons: OAR has stopped to support Ada for RTEMS, and the only qualified API is the RTEMS Classic API. So, the proposed solution is based on using GNAT Pro for ERC32 with Zero-Foot-Print run time system (i.e. no run time system) on top of RTEMS Classic API.

# APPLYING MODEL-DRIVEN ARCHITECTURE AND SPARK ADA – A SPARK ADA MODEL COMPILER FOR XTUML

Erik Wedin, M.Sc. (C.Sc. & E.)
erik.wedin@saabgroup.com
Senior Specialist - Software Systems Architecture
Software Development
Development & Technology
Saab Bofors Dynamics AB, Sweden

Saab Bofors Dynamics has worked with MDA (Model-Driven Architecture) and xtUML (Executable and Translatable UML) and their precursors since the early 1990's. The methodology brings full automation of the translation of models to other models and finally to code, and the reuse of application models as well as architecture models between different types of systems.

The translation rules are formalised in a model compiler. A model compiler is in fact a reusable software architecture containing architecture metamodels expressed in xtUML, translation rules, target source code components and marks. Marks are used to control how the translation of the xtUML application models is performed.

The presentation covers how SPARK Ada has been used on a joint embedded software safety-related programme with a partner company in a Model-Driven Architecture context.

Initially the software parts produced were temporally isolated from the surrounding safety-related software, so an in-house model compiler generating full Ada code was used successfully. The surrounding software was implemented in SPARK Ada by the partner company.

When the temporal separation was removed, i.e. the software parts produced were executing concurrently with the safety-related software parts, a new model compiler generating SPARK Ada was developed. It was developed from scratch since SPARK Ada affects all aspects of a software architecture thus it was not realistic to re-design the existing full Ada architecture.

A number of requirements on the architecture were identified including, high execution performance, small footprint and to support SPARK analysis – dataflow, information flow and proof of absence of run-time errors.

The software architecture was developed during a number of technical workshops where the partner company contributed with SPARK knowledge and Saab Bofors Dynamics with MDA, xtUML and model compiler expertise.

The result was a SPARK Ada software architecture suitable for automatic translation from xtUML being used live. The architecture was formalised into an xtUML model compiler for SPARK Ada, generating 100% complete SPARK Ada code, including annotations, from 100% executable platform-independent models in xtUML. The existing xtUML models were regenerated without being modified.

The presentation shares experiences and reflections from the design of the software architecture, how it was formalised in a model compiler and from its usage in the project.

# ADA95 USAGE WITHIN THE AIRBUS MILITARY ADVANCED REFUELLING BOOM SYSTEM

Ismael Lafoz
Ismael.Lafoz@military.airbus.com
Control Systems Software Department, AIRBUS MILITARY
Po. John Lennon, 2, 28906 Getafe (Madrid) – Spain

The presentation will show the usage of the Ada95 programming language during the development and the final implementation of the Fly-By-Wire system of the Advanced Refuelling Boom System (ARBS) designed, developed and commercialized by Airbus Military.

The ARBS has been installed in the A330 MRTT, which is a conversion of a basic Airliner Aircraft into a Tanker Aircraft to transfer fuel in-flight from the main fuel tanks to receiver aircraft. It consists of a telescopic mast or Boom, attached to the underside fuselage of the aircraft, and the relevant electronic and mechanical systems, which make the mast deployment possible from the stowage position, its extension and connection with the receiver aircraft, the supply of fuel and, after the refuelling, the mast disconnection, retraction and stowage.

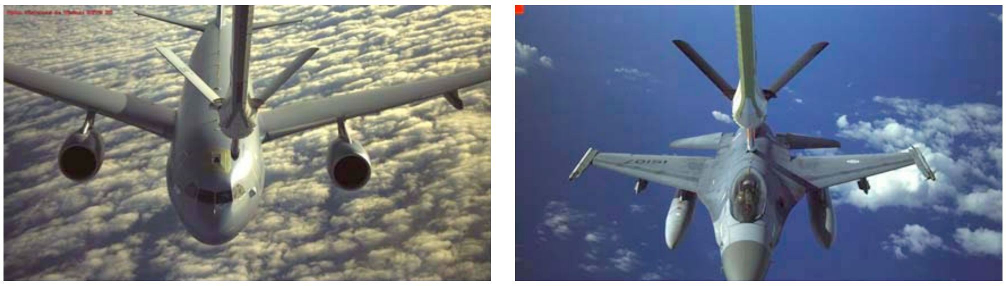


*Figure 1: Flying Boom in the pre-contact position with an A330 MRTT receiver (left). Boom coupled with an F-16 receiver (right).*

The core system of the ARBS is the Boom Control and Computing System (BCCS), a redundant control/monitor architecture that comprises four computers. The basic functionality of the BCCS is to receive inputs from the operator through flight control sticks, sensors and aircraft systems, to compute the flight control laws, to determine the system operational mode and to control and monitoring the actuators, which are mainly connected to the aerodynamic surfaces. Additionally, the BCCS is in charge of the management of the control and monitoring of the extension/retraction system and the control and monitoring of the hoist and uplock system for raising/lowering and locking/unlocking the boom. Besides, the BCCS is in charge of the management of the Pilot Director Lights, which are used by the boom operator to guide the receiver aircraft to the right contact position for the refuelling operation. The BCCS is also in charge of providing the failure detection, recording and isolation system and of managing the redundancy mechanisms. Instruction operations are also allowed using two flight control sticks in order to provide training capabilities when the Boom is in flight. Every computer that comprises the BCCS is based on a PowerPC 750 CPU and a VME chassis where additional boards are included for supporting all the following physical interfaces: discretes, analogues, ARINC 429 and CAN Bus.

The SW architecture defined for this system is based on ARINC 653 architecture, so it comprises different partitions where the functionality implementation is deployed. Such partitioning- based architecture is based on a Real Time Operating System that supports the ARINC 653 specification, as it is VxWorks 653 from WindRiver.

The major part of the Application SW has been manually implemented in Ada95, but there is also some C code automatically generated from models taking advantage of the model-based development for such complex algorithm easily defined, simulated and implemented using modelling tools. Additionally, the Board Support Package (BSP) and the interfaces drivers, both supplied by the HW provider, were developed also in C.

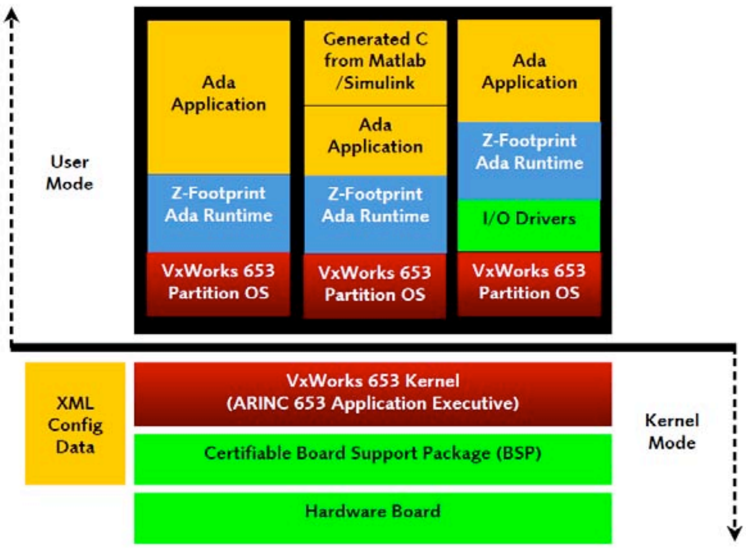


*Figure 2: Types of partitions of the ARINC 653 SW Architecture defined for the BCCS system.*

The safety assessment of the system defined the DAL of the BCCS as Level A, so all the development processes, the architecture, the implementation and the verification of the developed SW were performed in order to be compliant with the DO-178B certification standard and all its objectives for Level A SW.

The system was mainly developed using Ada95 as programming language, taking advantage of the safety-related Ada95 features, the usage of an Ada95 high integrity compiler, a safe subset of Ada95 according to a specific Coding Standard and a specific safe Ada95 profile provided by the compiler provider.

Ada95 code was also automatically generated using homemade generation tools from the SW Design UML model for the implementation of the code mainly related to the ICD requirements and the ARINC 653 artifacts (partitions, intra and inter-partition communication and processes). Model-based design methodology was used in this project mainly for the development and implementation of the control algorithms or flight control laws, so an specific mechanism was designed for connecting or modeling the models performed in modeling or simulation environments as Simulink, within an UML Design model. The SW components for connecting both the C code automatically generated from the Simulink model and the Ada95 code manually implemented, were automatically generated in Ada95 in order to ensure the right and proper mapping between both languages.

Specific tools for Ada95 were used for the verification of the compliance with the Coding Standard rules, so qualification, in terms of DO-178B, of such tools as verification tools were performed during the development of the project. The robust–ness, safety and reliability of the system have been demonstrated through hundreds of test flights with the actual aircraft platform, with a fully absence of incidents. So, the flight test phase is almost finished and the final certification phase of the system has been already started with the current Spanish Military Certification Authority.

# ADA95 USAGE WITHIN THE AIRBUS MILITARY GENERIC TEST ENVIRONMENT

Bartolomé Lozano
Bartolome.Lozano@military.airbus.com
Airbus Military
Spain

The current abstract summarizes the proposed Industrial Presentation to show the usage of the Ada95 programming language and the GNATPRO suite during the development and the final implementation of the Generic Test Environment "SEAS".

Aircraft electronics systems in recent years have increased their complexity and sophistication, to achieve their demanding requirements of performance, reliability and security.

Verification and validation process, of such systems, requires large tests sessions, at different phases of the development route. These system integration benches are required to perform equipment/subsystem/system verification tests against equipment/subsystem/system specification, and to perform equipment, subsystem, and system validation tests to check that the product as implemented meets the expectations of the product customer.

For reasons of safety and cost most of these tests are passed in laboratory facilities and on ground plane, leaving for flight tests the confirmation that the behaviour observed in laboratory is not affected by the installation in aircraft and the environmental effects.

To achieve laboratory tests the so-called Integration benches are needed, which use real equipment, connected to each other as in the aircraft and the possibility of interaction with the drivers / operators, reproduce dynamically the input interfaces to a system; monitor the behaviour and feeding back the responses to simulated environment. To support this functionality Integration Benches must have large capabilities for simulation, stimulation of entries, data acquisition, recording, sequencer, configuration and presentation. All these capabilities are provided by the so-called test system what is the "brain" of the Integration bench.

All AI Military  integration benches use a standardized solution for system testing, the SEAS (Stimulation, Acquisition and Simulation, System). SEAS is a computer aided test environment responsible for the preparation, execution, analysis, configuration and data distribution, simulation, recording, replay, sequencer, HW interface and instrumentation of the  tests required by A/C equipments/subsystems/systems validation and verification process. [fig 1]

"SEAS" Generic Test environment is a set of modular, scalable and distributed HW and SW items, which acts in combination, and in an integrated way, to provide support for the testing activities, throughout Aircraft equipments/subsystems/systems development life-cycle processes, starting from virtual, in an early design stage, to real on all development stages, or even during flight test and during in-service maintenance by using a common environment for Engineering Simulators, SW Benches, Functional test benches, Target Benches and Final Assembly Lines Aircraft Interface Modules.

Test Bench used to having closed methods and tools set with A/C electronic systems. Both domains share  ICD's, models, test cases and when appropriate programming languages.

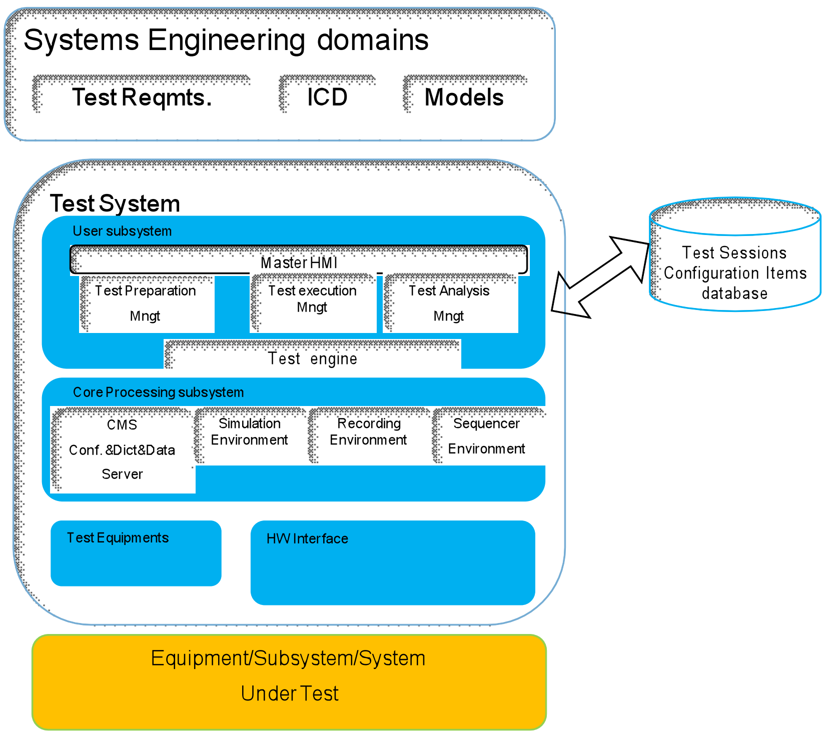


*Fig 1. SEAS Context Diagram*

The challenge was how to build test environments with a common core, following Modular Open System Approach principles (MOSA), with sufficient open standards for HW/SW interfaces, services, and supporting formats to enable properly engineered components to be utilized across a wide range of test systems with minimal changes, to interoperate with other components on local and remote test systems, and to interact with users in a style that facilitates portability offering:

- Continued access to cutting edge technologies and products from multiple suppliers of processors, avionics and non-avionics HW Interfaces, Instrumentation, Models, ICD's and test requirements.
- Greater reliability, reconfigurability , portability, interoperability, reusability, scalability and maintainability.
- Supports timely and affordable technology insertion (reduced cycle time) for Simulation, ICDs, Test cases, HW interfaces and instrumentation.
- Lower Risk mitigating the risks associated with technology obsolescence
- Reduced Life Cycle Cost.
- Adaptable to evolving requirements and threats.

In the current nightmare scenario of continuous evolution of technology:

- Increasingly powerful multi-core processors.
- Increasingly networking computing technology used in A/C systems.
- Increasingly networking computing technology used in instrumentation (LXI).
- Web technology for Service Oriented Architecture (SOA) of engineering and configuration tools frameworks.
- New avionics interfaces.
- The exchange of information is increasing in volume and speed.
- Protocols are used with more complicated interfaces.

The AI Military test environment solution show how thanks to the GNATPRO suite it was possible to fulfil the strong requirements of such test systems, taking advantages to provide a backbone to articulate the integration of components with mixed languages, mixed components and a variety of Commercial Off The Shelf.

This backbone is the glue for the big variety of HW/ SW test bench items like Chassis, I/O Interface boards, Fault Insertion Break-out boards, Commutation Matrix boards, Relays Boards, processors Boards, third parties tools, semi-detached tools, integrated tools, Laboratory Instruments, power distribution control, User stations, Simulations, recording, sequencer, Data dictionaries and configuration control, Test preparation, test execution and test analysis, graphical data-visualisation, numerical display, trend display, synoptic and user defined components.

The presentation is focussed in the SEAS Core SW developed using GNATPRO suite like:

- IO board drivers binding of a wide family of interfaces including aircraft multiplexed links like AFDX, MIL-STD-1553, MIL-STD-3910, CANbus; Point to Point links ARINC429, Discretes, Analogues, Synchros, LVDT, RVDT.
- Test bench configuration management.
- Signals, interfaces, buses configuration and scaling objects distribution to remote processors by using GNATPRO GLADE.
- The integrated development environment for Simulations explaining how these simulations are seamless integrated from user written models or Simulations written in languages like Ada , C, C++, FORTRAN, binary code or mixed languages.
- Examples of HMI.

This test environment shares signals, buses topology, simulations, data-visualisation across different facilities needed by a/c system life cycle, starting from Desktop Simulators, Virtual Test Benches, Equipment/subsystem/system Integration Benches, and Aircraft Interface Modules for Final Assembly Lines.

SEAS is a SW/HW components federation used as the basic building block that it allows bench scalability to form more complex federations for a multi-system integration bench.

The maturity, robustness, safety and reliability of the system have been demonstrated throughout hundreds of test facilities in use with this common test environment including Engineering Simulators for Aircraft Refuelling Boom System, System Integration Benches for Multirole Tanker Aircrafts, A400M, Lights&Medium Transport Aircraft, Full Integrated Tactical Systems and Aircraft Interface Modules for Final Assembly Lines of A400M, Multirole Tanker Aircrafts and L&MT aircrafts.

# IMPLEMENTING POLYMORPHIC CALLBACKS FOR ADA/C++ BINDINGS

Maciej Sobczak
Maciej.Sobczak@cern.ch
CERN
Switzerland

## Context of the actual work

The work that has lead to the development of presented solution was done during the last year on the YAMI4 project, which is an open-source set of communication libraries for multilanguage distributed systems. Ada, C++ and Java are currently supported by the project.

The core part of the library set is implemented in C++ and implements basic communication services. This part is used by high-level libraries implemented in Ada and C++, which handle message routing, progress tracking and user-provided actions.

The messaging system needs a way to propagate event notifications between the low-level core components and the high-level message handlers. These notifications were implemented in the form of object-oriented callbacks. The challenge was to preserve the object-oriented notions (dispatching calls via class-wide types) across the language boundaries in the binding between Ada and C++.

## Generic nature of the problem

The problem of notifications between layers in the software stack is not limited to messaging systems. In fact, this problem pattern can be identified in many other systems, and as such can be extracted and presented in the form of a simplified example – a separately provided presentation handout serves as a reference and a starting point for more elaborate implementations.

## The technical details

The actual problem involves a C++ library with a callback engine, where an abstract class is used as a callback interface for notifications.

The Ada binding that preserves the object-oriented notions in the callback includes two translation layers that are added for the Ada and C++ parts – these layers reduce the high-level language constructs (class-wide types and dispatching calls) to a simpler subprogram call model that is appropriate for standardized language linking that is supported by a relevant pair of Ada and C++ compilers.

In such a multi-layered architecture, the high-level constructs that are visible at the level of each language are preserved without reliance on obscure or undocumented compiler conventions.

An important property of the presented solution is its non-intrusiveness and loose coupling that allows the individual layers to be reused in other architectural combinations.

### Biographical note

Maciej Sobczak works as the middleware team leader at CERN, where he is responsible for central communication services in the accelerator control system. In addition to this work, he is an open-source contributor and provides independent consulting services.

You can contact him at http://www.inspirel.com/

Reliable Software Technologies, Ada-Europe 2010 – Industrial Presentation

# *Implementing Polymorphic Callbacks for Ada/C++ Bindings* (presentation handout)

Maciej Sobczak

## C++ layer – callback engine with wrapper

```
// base.h:
class Callback
{
public:
    virtual void call() = 0;
};

void registerCallback(Callback * c);
void fireAll();
```

```
// base.cpp:
#include "base.h"
#include <cstdio>
#include <vector>

std::vector<Callback *> allCallbacks;

void registerCallback(Callback * c)
{
    std::puts("base: register callback");
    allCallbacks.push_back(c);
}

void fireAll()
{
    std::puts("base: fire all!");
    for (std::vector<Callback *>::iterator it =
        allCallbacks.begin(); it != allCallbacks.end(); ++it)
    {
        (*it)->call();
    }
}
```

```
// wrapper.cpp:
#include "base.h"

extern "C" typedef void (*CallbackFunctionType)(void *);

class WrappedCallback : public Callback
{
public:
    WrappedCallback(
        CallbackFunctionType function, void * object)
        : f_(function), obj_(object) {}

    virtual void call()
    {
        // call into the Ada translator procedure
        f_(obj_);
    }

private:
    CallbackFunctionType f_;
    void * obj_;
};

// functions "exposed" to the Ada layer:

extern "C" void wrapped_registerCallback(
    void * function_addr, void * object)
{
    // brute-force conversion from
    // raw procedure address obtained from Ada
    // to C++ function pointer
    union
    {
        void * raw_pointer;
        CallbackFunctionType function_pointer;
    } converter;

    converter.raw_pointer = function_addr;
    CallbackFunctionType function = converter.function_pointer;

    registerCallback(new WrappedCallback(function, object));
}

extern "C" void wrapped_fireAll()
{
    fireAll();
}
```

## Ada layer – adapter with example user program

```
--  callbacks.ads:
package Callbacks is

   type Callback is interface;
   type Callback_Access is access all Callback'Class;

   procedure Call (Self : in Callback) is abstract;

   procedure Register_Callback (C : in Callback_Access);
   procedure Fire_All;

end Callbacks;
```

```
--  callbacks.adb:
with System.Address_To_Access_Conversions;

package body Callbacks is

   subtype Void_Ptr is System.Address;

   package Conversions is
      new System.Address_To_Access_Conversions
     (Object => Callback'Class);

   --  helper translator,
   --  will be directly called by the C++ wrapper:
   procedure Callback_Translator (Obj : in Void_Ptr);
   pragma Convention (C, Callback_Translator);

   procedure Callback_Translator (Obj : in Void_Ptr) is
      C : Callback_Access :=
        Callback_Access (Conversions.To_Pointer (Obj));
   begin
      --  actual dispatching call to the Ada implementation:
      C.all.Call;
   end Callback_Translator;

   procedure Register_Callback (C : in Callback_Access) is
      procedure Wrapped_Register_Callback
        (Fun : in Void_Ptr; Obj : in Void_Ptr);
      pragma Import (C, Wrapped_Register_Callback,
                     "wrapped_registerCallback");
   begin
      Wrapped_Register_Callback
        (Callback_Translator'Address,
         Conversions.To_Address (Conversions.Object_Pointer (C)));
   end Register_Callback;

   procedure Fire_All is
      procedure Wrapped_Fire_All;
      pragma Import (C, Wrapped_Fire_All, "wrapped_fireAll");
   begin
      Wrapped_Fire_All;
   end Fire_All;

end Callbacks;
```

```
--  example.adb:
with Ada.Text_IO;
with Callbacks;

procedure Example is

   type Some_Callback is new Callbacks.Callback with null record;
   overriding procedure Call (Self : in Some_Callback);

   overriding procedure Call (Self : in Some_Callback) is
   begin
      Ada.Text_IO.Put_Line ("Ada: Some Callback called");
   end Call;

   type Other_Callback is new Callbacks.Callback with null record;
   overriding procedure Call (Self : in Other_Callback);

   overriding procedure Call (Self : in Other_Callback) is
   begin
      Ada.Text_IO.Put_Line ("Ada: Other Callback called");
   end Call;

   SC : aliased Some_Callback;
   OC : aliased Other_Callback;

begin

   Ada.Text_IO.Put_Line ("Ada: registering callbacks");
   Callbacks.Register_Callback (SC'Unchecked_Access);
   Callbacks.Register_Callback (OC'Unchecked_Access);

   Ada.Text_IO.Put_Line ("Ada: fire all!");
   Callbacks.Fire_All;

end Example;
```

# A Reusable Work Seeking Parallel Framework for Ada 2005

Brad Moore
Brad.Moore@gdcanada.com
General Dynamics
Canada

The Ada programming language is seemingly well positioned to take advantage of emerging multicore technologies. While it has always been possible to write parallel algorithms in Ada, there are certain classes of problems where the level of effort to write parallel algorithms outweighs the ease and simplicity of a sequential approach. This can result in lost opportunities for parallelism and slower running software programs.

This presentation explores Ada's concurrency features to see whether it is possible to easily inject iterative and recursive parallelism to code written in Ada, without having to resort to special language extensions or non-standard language features.

This paper identifies a “work-seeking” technique, which can be viewed as a form of compromise between work-sharing and work-stealing, two competing strategies described in the literature. The presentation then goes on to propose how parallelism pragmas could be added to Ada to further facilitate parallelism. The presentation concludes by suggesting how the approach might be applied to a Battlefield Spectrum Management application.

# DATABASE PROGRAMMING WITH ADA

Frank Piron
frank.piron@konad.de
KonAd GmbH, Freiburg
Germany

**Keywords:** Database Programming, GUI Development, Automatic Layout, Multitasking Konada.Db, GWindows, Oracle Call Interface

## Summary

Since 10 years ago the KonAd team develops Oracle Database Applications based on the Oracle Call Interface and the Developer 2000 tool chain. By the beginning of the millennium ORACLE cancelled further development of the PL/SQL-based Client-Server tools and decided to switch to Java. Since Java was not the language we wanted to work with, KonAd started to build an own database development framework in Ada based on the Oracle Call Interface, but database independent by design.

The talk will give an introduction to the Konada.Db library with code examples and real project experiences. Especially the work done since our presentation at the Ada-Europe 2006 conference [1] will be presented.

The presentation will concentrate on the structure and use of the library but will also show the reasons why we chose Ada for database programming and which experiences we made.

## Why Database Programming in Ada?

We wanted to use a language with object orientation and suitable for the development of large applications. Further programming on different OS-Platforms should be possible. At this point the standardization of Ada came in. The detection of errors at compile time was a very important feature because runtime errors in online transaction database applications with hundreds of users would be difficult to handle.

Finally we considered the built in multitasking capabilities and the similarity between Ada and the Oracle procedural language PL/SQL as an Ada83 derivative, and decided to use Ada in our future development.

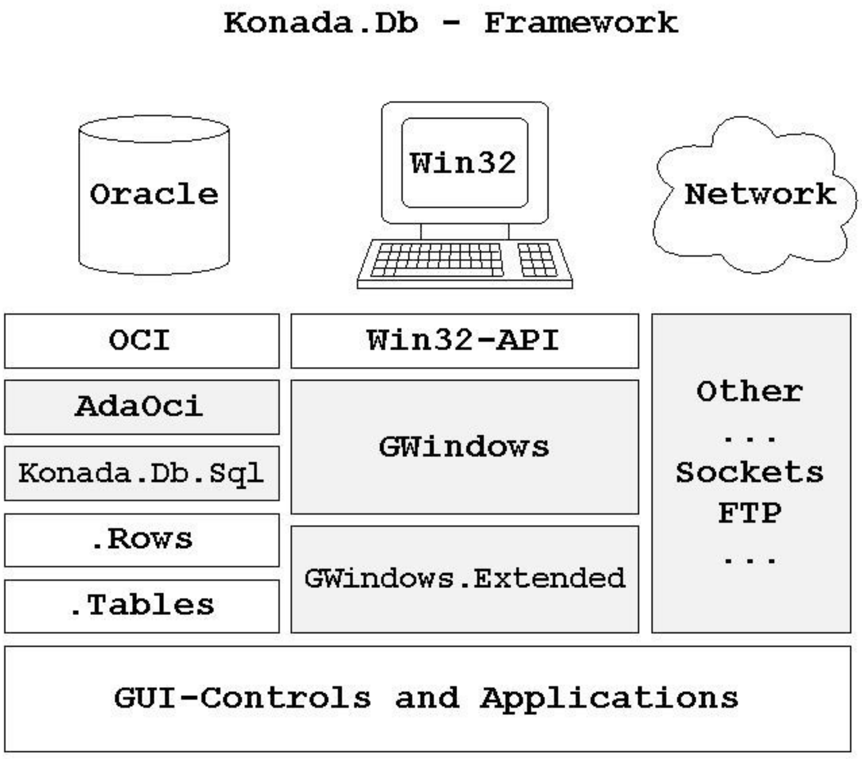
## The Konada.Db Library

The Konada.Db library contains database and GUI services. The database services are organized in four layers.

- AdaOci, a thin binding to the Oracle Call Interface written by Dmitryi Anisimkov, Blob-enhancement by Frank Piron
- Konada.Db.Sql, a SQL-based thick binding
- The Konada.Db.Row container datatype to hold a table row of data
- The Konada.Db.Tables datatype which allows data manipulation without use of SQL

The GUI services have three layers

- Gwindows, a thick binding to the Win32-API (David Botton)
- Gwindows.Extended, extensions written by KonAd
- GUI-Controls for direct user interaction



*The Konada.Db Framework - a structural view (shaded boxes are available under GMGPL)*

**GUI-Controls**
The GUI-controls of our framework are the basic components for complex database applications with state of the art user interfaces for the win32-platform. They provide default functionality for the presentation and manipulation of data. For that only few lines of code are necessary since layout is done automatically by the library using high-level layout directives. A flexible and extensible event-model is provided to enhance the controls with application specific code. Since the GUI are created dynamically all information about layout and item types like checkbox | listbox | textitem… may be hold in the database and may be changed without recompilation.

The presentation will conclude with examples and project reports including demonstration of running applications

- Simple table maintenance programs with Konada.Db
- A complete ERP-Solution built with Konada.Db
- Non GUI-database applications on Linux/Solaris

**References**
[1] Frank Piron, "The Development and Deployment of a Workflow System Partially Written in Ada95", Ada Europe 2006, Industrial Presentation. *http://www.hurray.isep.ipp.pt/activities/ae2006/index2.html*

# FUTURE ENHANCEMENTS TO THE U.S. FEDERAL AVIATION ADMINISTRATION'S (FAA) EN-ROUTE AUTOMATION MODERNIZATION (ERAM) PROGRAM AND THE NEXT GENERATION AIR TRANSPORTATION (NEXTGEN) SYSTEM

Jeffrey O'Leary
Software Development and Acquisition Lead, En Route and Oceanic Services Directorate, US Federal Aviation Administration, Washington DC (USA)

Alok Srivastava
alok.srivastava@auatac.com
TASC Inc, (formally Northrop Grumman IT), Washington DC (USA)

This presentation will discuss the future enhancements to the En-route Automation Modernization (ERAM) program, the biggest modern Ada software engineering based system with focus on new lessons learned during the deployment, short falls and enabling new technologies. The new functionalities and enhanced capabilities will be piggyback on ERAM's robust software infrastructure therefore will continue to significantly use Ada.

The presentation will also talk about FAA's most ambitious new undertaking, the Next Generation Air Transportation System that would replace the current radar- based air traffic control system in which data, communications and instructions flow to and from a handful of ground control facilities, to a satellite-based system that would allow aircraft to locate each other and communicate with each other. This would allow more efficient use of congested air space and airport facilities. The NextGen is expected to be in place by 2025 with few capabilities that can be achieved in the mid-term, from 2012 to 2018. The current plan includes five major programs, Automatic Dependent Surveillance Broadcast (ADS-B), System- Wide Information Management, NextGen Data Communications, Network Enabled Weather and the National Airspace Voice Switch. The presentation will also discuss how the NextGen components will be integrated with ERAM and what enhancements in Ada Software Engineering and products the FAA would like to see to accomplish such goals.

# SYSTEM ARCHITECTURE VIRTUAL INTEGRATION CASE STUDY

Bruce Lewis
bruce.a.lewis@us.army.mil
US Army Aviation and Missile Command
USA

The System Architecture Virtual Integration project (SAVI) is a major multi-phase program being sponsored within the Aerospace Vehicle Systems Institute (AVSI).

Originally a Boeing initiative, the AVSI is now an international organization sponsoring pre-competitive research primarily in the domain of aviation and aerospace. The participants in the SAVI first phase project and demonstration were Airbus, Boeing, and Lockheed Martin, as system integrators, and BAE Systems, GE and Rockwell Collins as system suppliers, with the involvement of the DoD, the FAA, and the Software Engineering Institute.

The SAVI project is very ambitious, providing a new paradigm for system development. It is to demonstrate and then develop for production use an architecture centric "virtual integration" approach based on quantitative model based, component based and proof based engineering for system acquisition, development and lifecycle upgrades. The SAVI paradigm will significantly impact the commercial and military aviation industry first, then other domains with similar requirements for real-time, safety, security, and predictable performance.

At this point, the first phase has been completed and this report will provide a summary of the results based on the published SAVI Case Study and the author's personal participation in the project. The next phase is about to begin.

Driving the industry to work together on a solution, software/system integration costs are now a major component, if not the largest component, of system development cost and schedule risk and threaten each company's ability to develop the next generation of aircraft. However, the technology solution pieces to this driving need can now be leveraged by small programs and the concepts applied to embedded real-time system upgrades. The key concept is to address system requirements and design errors as early as possible through quantitative modeling, in effect to provide a virtual validation or feasibility assessment incrementally throughout the program.

The focus is not on component correctness, which is now less of an issue, but on the integration of components. The participants agreed that the most expensive errors discovered in integration were in the software/system architecture. Hence, SAVI phase 1 shifts the discovery process forward by using a precise architecture description language (AADL) to model early and incremental advances in the architecture's definition, driving from a unified system model, many dimensions of analysis of critical architecture qualities and constraints with incremental enhancements in fidelity as the design matures. This semi-formal architectural approach enables early discovery. In fact, the SAVI process starts pre-acquisition and involves the exchange of quantitatively analyzable, integratable models throughout the development process.

Thus the SAVI catch phrase, "Integrate, then Build". The first phase of SAVI has demonstrated the ability to do model based, component based and proof based architectural analysis supporting virtual integration in an acquisition process for the incremental validation of requirements and design. The first phase project included: 1) development of the model-based architecture centric acquisition process, 2) selection of analysis approaches from experiences in system integration, 3) development of the shareable but protected repository for the unified system architectural model, based on strong architectural semantics, 4) model bus transformations to various analysis tools and to a UML modeling tool, 5) demonstration of the model-based acquisition and development process using the selected analysis methods through multiple stages of the process and multiple tiers of the architecture, and finally 5) analysis of expected Return on Investment.

Each of these elements will be summarized in the presentation.

## LESSONS LEARNED FROM THE FIRST HIGH ASSURANCE (EAL 6+) COMMON CRITERIA SOFTWARE CERTIFICATION

David Kleidermacher
davek@ghs.com
Green Hills Software Inc.

**Overview**

An operating system has recently been certified to the highest Common Criteria security level (EAL 6+ High Robustness) ever achieved for a software technology. This case study will describe the certification requirements, including formal methods and NSA penetration testing, lessons learned navigating the certification process, and the security principles that guided development.

Outline for the presentation:

1. Introduction
    - Brief Overview of Common Criteria, Assurance Levels
    - Comparison of Operating System Protection Profiles
    - Issues Relating to Validating a Protection Profile and using custom Security Targets

2. Overview of Certified Software
    - What Drove the Requirement for Certification
    - What is Meant by EAL 6+ and "High Robustness"
    - Historic Certifications Performed on Similar Classes of Software

3. EAL 6+ Requirements and Lessons Learned from Meeting Them
    - Configuration Management
    - Testing
    - Understanding the Applicability of other Reliability Standards to this Certification Effort
    - Development Security and Secure Delivery
    - Formal Functional Specification, Design, and Implementation Representation; Lessons Learned with Formal Methods
    - Assured Maintenance Process and Lessons Learned from Subsequent Applications
    - Tools Assurance
    - Vulnerability Assessment and Lessons Learned working with US NSA Penetration Testers

4. Conclusion
    - How the Common Criteria Standard Performed in this Effort
    - Applications, Cost, and Importance of High Assurance Software Security Certifications

# AN INTRODUCTION TO PARASAIL: PARALLEL SPECIFICATION AND IMPLEMENTATION LANGUAGE

S. Tucker Taft
stt@sofcheck.com
SofCheck, Inc.
Burlington, MA
USA

This presentation will provide an introduction to "ParaSail", a new programming language being designed from scratch, in the belief that a well-designed programming language can result in more productive programmers building higher quality software. In the particular area of high-integrity software, including both safety-critical software and high-security software, there is all the more reason to use the very best programming language you can, because the problems you are trying to solve and the level of quality required is at the very limits of what can be accomplished.

ParaSail is meant to address the goals of producing inherently safe and secure software, while taking advantage of the wider availability of true parallel processing in the form of multi-core chips. It is intended to promote a formal approach to software, where the program text includes pre- and postconditions, liberal use of assertions and invariants, etc., with tool-supported proof of correctness with respect to the formal annotations.

The language is named ParaSail as an acronym for Parallel Specification and Implementation Language. ParaSail is a completely new language, but it steals liberally from other programming languages, including the ML family, the Algol/Pascal/Ada family, the C/C++/Java family, and the region-based languages (especially Cyclone). Perhaps one significant deviation from the excellent baseline established by ML, Eiffel, Java, Scala, etc. is that ParaSail is intended to avoid "fine-granule" garbage collection in favor of stack and region-based storage management. The other major deviation from the above-named language families is that ParaSail is inherently parallel. The programmer has to work harder to force sequential evaluation. By default, evaluation proceeds in parallel for almost all constructs.

As far as language design philosophy, ParaSail tries to minimize implicit operations, implicit parameters, implicit dynamic binding (virtual function calls), implicit initializations, implicit conversions, etc. This is both in the name of clarity for the human reader, and in the name of formal testability and verifiability. ParaSail uses a small number of concepts to represent all of the various composition mechanisms such as records, packages, classes, modules, templates, capsules, structures, etc. Arrays and more general containers are treated uniformly.

On the other hand, ParaSail allows many things to proceed in parallel by default, effectively inserting implicit parallelism everywhere. Parameter evaluation is logically performed in parallel. The language disallows uses that would make the result depend on the order or concurrency of parameter evaluation. The iterations of a for loop are by default executed in an arbitrary order. Explicit ordering must be specified if it is required by the algorithm. Even sequential statements are essentially converted into a data-flow based DAG, which is then evaluated in parallel in so far as possible. In all cases, the language disallows code that could result in race conditions due to inadequately synchronized access to shared data. Race conditions are avoided either by using data structures specifically designed to support concurrent access (either lock-based or lock-free), or by relying on *handoff* semantics (similar to that of linear types, distributed languages like Hermes, or the UVM virtual memory system). Handoff semantics ensures that once a variable is passed as a writable parameter as part of a call, it is no longer available for other use until the called routine returns, effectively eliminating both race conditions and unintended aliasing.

Much of this kind of implicit parallelism is possible in pure functional languages, and ParaSail will support a functional programming style where it works naturally. Unfortunately, doing certain relatively straightforward things in pure functional languages can be awkward, while a normal assignment statement is something that most developers understand intuitively, even though it potentially breaks the referential transparency that pure functional languages can provide.

This presentation will provide examples of the features of ParaSail as currently designed, compare and contrast it with other existing languages, and discuss the rationale behind the choices made in its design.  We will also identify the open issues needing resolution prior to completing the design of ParaSail.