Hybrid (Real Time) Test Benches based on Real Time Linux

Bjoern Kircher[(1)], Denis Chatelais [(2)], Wilfrid Gonzalez[(2)]

[(1)] ASTRIUM Satellites
Claude-Dornier-Str.
D-88090 Immenstaad BW
Email: bjoern.kircher@astrium.eads.net

[(2)]ASTRIUM Satellites
31 rue des Cosmonautes
Z.I. du Palays
F-31402 Toulouse Cedex 4
Email: denis.chatelais@astrium.eads.net
wilfrid.gonzalez@astrium.eads.net

INTRODUCTION

The use of Linux for simulation is state of the art. Linux has become a well-established operating system for several use cases. And since the standard Linux kernel is real time capable there is growing market supporting real time applications.

Most of Astrium's non real time simulations are executed on Linux whereas the real time simulations (hybrid simulation interfacing hardware in the loop) are still based on specific real time operating systems. These specialised operating systems are very effective but most of them are much adjusted for a dedicated set of use cases and the usage of these operating systems is normally not very handy compared to a normal, well-known system like Linux. In addition, as various simulation elements (e.g. component models) need to run in both, the non-real time and the real time simulations, adjusted models for the different environments must be developed, tested and maintained.

Since there is a growing demand on powerful, flexible hybrid simulations within the current running and planned satellite projects, the time is right to review the currently used real time platforms.

Why not using Linux for the real time simulations, too?

Real Time Linux or Linux for embedded systems is a promising technical field opening a lot of powerful options for real time simulation. Use of Real Time Linux for hybrid test benches has been introduced in several satellite and R&D projects. The question were: How does Real Time Linux compare to the previously used real time operating system, is Real Time Linux meeting the needs for hybrid simulations, is Real Time Linux capable of being integrated into the available test processes and what are the advantages respectively drawbacks of a Real Time Linux based simulation.
The tests were performed using simulation code and available hardware I/O devices from actual satellite projects to stimulate and monitor the Real Time Linux based simulation under "real" test bench conditions.

This paper presents the inclusion of Real Time Linux as simulation platform for real time satellite test facilities within Astrium.

HYBRID SIMULATION USE CASES

Term of "hybrid" is used when simulation is associated to real equipments operation. In these configurations, pure software simulation is coupled with a set of hardware interfaces in order to operate the equipments under test. Using real equipments implies to respect "real life" interface constraints (such as bus timing, reactivity between real and simulated equipment…), this introducing the term of real time architecture.
There are different basic hybrid configurations within the satellite development life cycle in Astrium. All these configurations are based on a Real Time Test Environment (RTE) which corresponds to a software simulator coupled to hardware interface.

The main configurations are:

- STB (Software Test Bench): The STB is used to support the development, verification and validation of the low level parts of the Onboard SW (HW-SW interface) and to calibrate the OBC numerical simulator against the real OBC. A STB consists of OBC (in general non redundant breadboard) and a RTE with limited interfaces for OBC.
- EFM (Electrical / Functional Model): The EFM test bench is used to validate the functional chains and to verify the HW/SW compatibility through open and closed loop tests and to prepare the PFM verification and AIT campaign. Operational validation of all AIT EGSE is also done on this bench. An EFM consists of a full redundant engineering model of the OBC, other engineering models of the flight HW (AOCS equipments), the RTE and possible and needed SCOEs (TM/TC, power….).
- PFM / FM ((Proto) Flight Model): The PFM / FM test bench is associated to the flight configuration of the satellite, and is used for AIT operation. This is a configuration where all equipments are real, but may also be supported at intermediate stages by simulated units of the real-time test environment, should the need arise. PFM test bench is based on the same architecture as the EFM and is also based around a RTE.
- SIMEFM: This is a hybrid configuration used to operate real equipments with the real Onboard SW and this without the need of an OBC. SIMEFM is build around the representative OBC model inherited from the other numerical simulation benches used for the Functional Validation. OBC model is associated to hardware interfaces (e.g. 1553 bus Controller I/F) needed to operate the equipments. This can be done in the EFM or in a PFM configuration (e.g. interface with payload).

All the RTEs used in these configurations consist of a real time capable simulator based on SimTG (Astrium's Simulation Environment: Simulator Third Generation) and a Simulation Front-End (SimFE/Avionic SCOE) providing the interfaces to the HW to be connected.

LINUX BASED RTE ARCHITECTURE

The Linux based RTE consists of a PC running on a Real Time Linux, the SimFE (also called Avionic SCOE) hosting the interface cards to the connected HW in the loop (HITL), a time & synchronization module and PCs to monitor and control the RTE.
The RTE part with the real time critical functionality is the SimFE, it is the main interface to the connected HW and thus the SimFE have to be operated under real time conditions.
The SimFE is available in two major configurations. The first configuration is the SimFE slave configuration which means that the SimFE is completely controlled by simulation and there is no autonomy within the SimFE. The second configuration is an intelligent SimFE which means that the SimFE itself is capable to handle the interfaces to the HW by itself. In some of the cases the two SimFE configurations are combined meaning that some of the HW interfaces are controlled by the Simulation (slave mode) and other interfaces are directly controlled by the SimFE itself. In all cases, the SimFE is based on a VME system whereas the Real Time Linux PC is based on PCI bus system.

The basic architecture is detailed in the figure below.

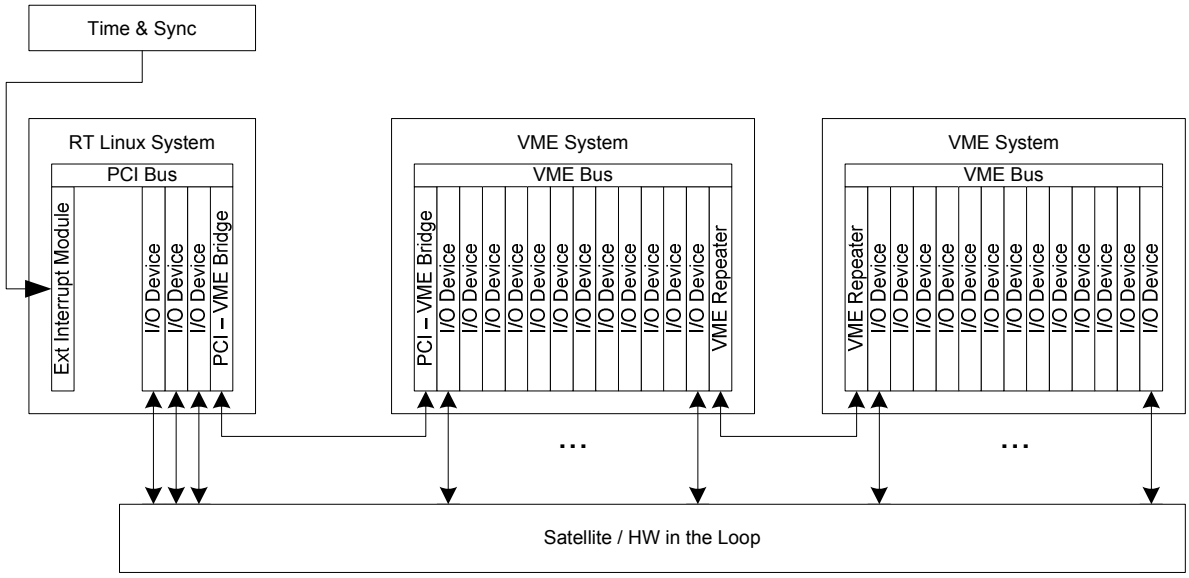

Fig. 1. RTE basic architecture with introduction of Linux platform

As shown in the figure above the two main building blocks of a RTE, the simulation and the SimFE, are connected via a PCI – VME Bridge. The bridge allows a bidirectional access from PCI to VME and the other way around. Thanks to this bus bridge it is possible to exchange data between the two systems and to control one side from the other.
Since the PCI – VME bridge adds additional latency between the VME I/O cards and the simulation, all time critical interfaces and interfaces with high data rates (like 1553, SpaceWire or UART) are directly located within the Real Time Linux system based on PCI or PCI-Express. This arrangement of interfaces allows an optimal load balancing between the SimFE and the PCI – VME Bridge and the processes in the Linux system.

Introducing time critical hardware interfaces directly in Linux system implies to manage real time software aspects. Thus, use of a standard Linux solution is not sufficient and a real time operating system is required. Use of Linux as a real time operating system for simulation has been introduced in Astrium in the beginning of the 2000's, where the simulation was interfaced (using a PCI/VME bridge) to simFE (Avionic SCOE). For performance reasons, only one hardware interface (the bridge) was plugged on the PCI bus of the Linux platform. Real time behaviour of the Linux kernel (2.4) was achieved using early releases of "low latency" and "preemptible" kernel patches from the open source community. Real time performances obtained were excellent but the whole integration of the kernel, the platform and the necessary driver for PCI/VME bridge was specific to the solution. The drawback was that an upgrade of one element (kernel, platform or driver) requires a new global integration activity.

Since these old times, the Linux kernel 2.6 family officially integrated "low latency" and "preemptible" mechanisms, which greatly ease the integration of Linux platform with hard real time capabilities. Added to that, the platform performance has continuously increased together with the number of processors/cores in the system.

HARD REAL TIME WITH LINUX

Linux by itself is not a RT operating system. However, since kernel 2.6, Linux can be configured to be preemtible and to react with low latency (enabling PREEMPT_RT flag). It provides as well SMP affinity system calls allowing to attach interrupts handling and tasks to dedicated processors or cores.
Based on this, and using the current multi processors/multi-cores platform, designing a hard real time system under Linux requires to allocate real time tasks to dedicated processors/cores and protect them from preemption by non real time tasks. At least one core of the system must be kept free for basic operating system activity (daemons services…).
By design, the real time tasks must avoid the use of system routines which may trigger high latencies (direct I/O calls, memory allocation…)
The latest PC architectures greatly improve the scalability of such solution by the increase of core number per processor.

The following figures illustrate the jitter of a real time task (scheduled by a 1 KHz hardware interrupt) depending on various types of non real time kernel activities (tests software exercising different types of activity on the platform : Memory management, matrix computation, network exchanges….)
In this test, based on a bi-processor platform, real time task is bound to one processor while the kernel runs on the other one.

The results show that jitter never exceeds 10 µs whatever the load of the non real time part of the system.

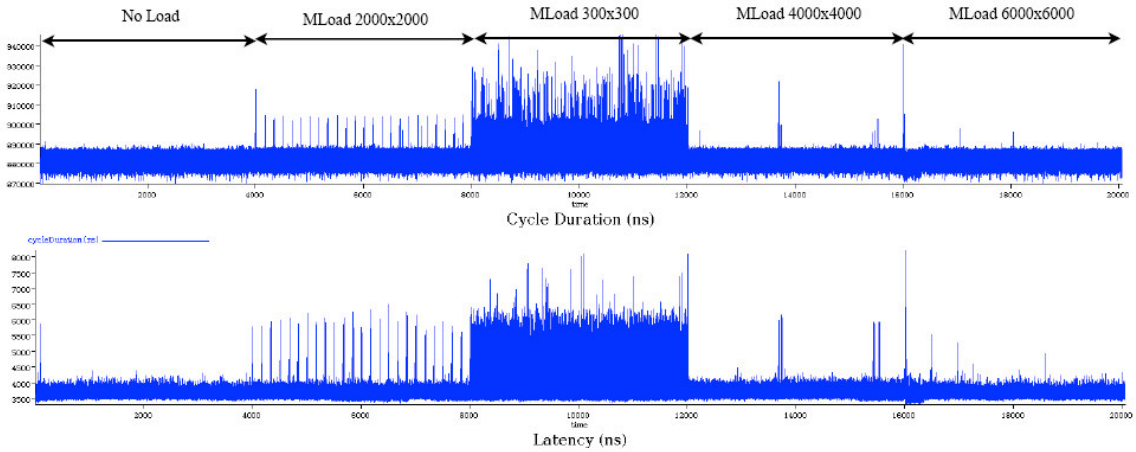

Fig. 2. Time measurement on a real time process under heavy load on the platform

Performances obtained are well adapted for RTE configuration as described in Fig. 1. In the RTE architecture, a multi-processor/multi-core platform (Xeon two CPU dual core) is used and real time design rules presented above are applied for the repartition of processors/core in regards of hard real time tasks associated to I/O cards. There is one core allocated to handle activities related to the PCI-VME bridge (handling of the SimFE slave hardware) and one core per other I/O card hosted in the Real Time Linux system (e.g. one core to handle the 1553 interface). Remaining cores are not dedicated for real time operation and are used by kernel/operating system and simulation infrastructure (SIMTG).

EFM / PFM USE CASE

For EFM and PFM configuration, RTE is configured with a large number of I/O interfaces in order to interface all avionic equipments of the satellite under test. As PC platform with PCI bus offers capabilities for a limited I/O cards and also to ensure continuity with previous RTE solution based only on VME, part of interfaces are still based on VME (Fig. 1.). Use of centralised software in Linux platform implies to manage at PCI side all the VME cards.

PCI – VME Memory Mapping

Accessing VME memory from PCI side is possible thanks to the memory mapping functionality provided by the PCI–VME Bridge. In a SimFE slave configuration as explained above, all I/O cards in the VME system are controlled from the Real Time Linux system. The mechanism of memory mapping is used to establish the control link between the two systems.
Controlling a VME card from Real Time Linux means that the VME card driver is running on the Real Time Linux system. To allow that, all the required address space of a VME card has to be mapped to a memory window on the PCI side. This common memory window between PCI and VME allows the Real Time Linux system to have a transparent memory access to the VME I/O cards via the bridge.
Thanks to this transparent memory access it is possible to implement VME device drivers running on Real Time Linux. The only difference to a single bus system is that the base address or bus address of an I/O card is the address of the memory window created on the PCI - VME Bridge. To run a set of device drivers it is reasonable to create one memory window per I/O device instead of mapping one memory window covering all I/O devices (see figure below).

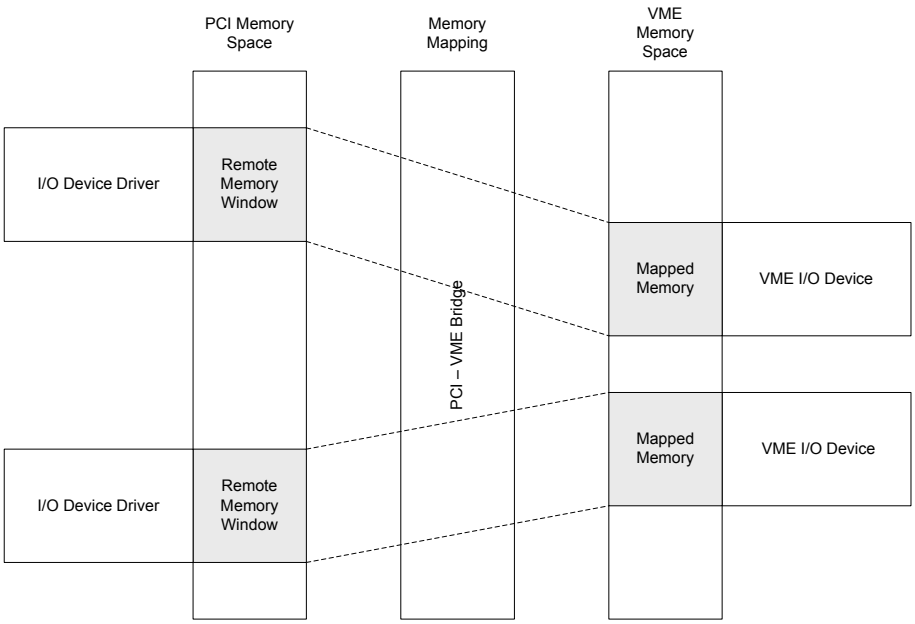

Fig. 3. Addresses mapping of VME I/O cards through the PCI-VME bridge

VME Interrupt handling

In addition to the memory mapping between PCI and VME it is also required to process the interrupts produced on VME side on the Real Time Linux system. VME interrupts are transferred through the PCI – VME Bridge and raised on Real Time Linux side as PCI interrupts. This means that the VME interrupts are looped through the bridge to the Real Time Linux system.
One special point of releasing VME interrupts is that there are two possibilities to do that. First ROAK (Release On AcKnowledge), the mostly used interrupt release mechanism. This mechanism is directly implemented in the VME devices and handled on HW level. The second VME interrupt release mechanism is RORA (Release On Read Access). This RORA mechanism is more challenging compared to ROAK because releasing an RORA interrupt means to read a dedicated register to release the interrupt manually.
The standard way of handling VME interrupts on Real Time Linux side is, that the bridge raises an PCI interrupt which causes the bridge driver running in the Linux kernel space to call a user defined interrupt service routine (ISR) running in the Linux user space. This not critical for ROAK because the interrupts are released directly and everything is conform to VME Bus timing requirements. This is different for interrupts which have to be released using RORA. Doing the RORA read access in the user ISR running in the Linux user space is too late and violates the VME Bus specification. Thus, the user ISRs for VME I/O devices have to be implemented differently for ROAK and RORA devices. To use a RORA device, the part of the ISR which implements the RORA interrupt release have to be implemented in the Linux kernel space.

RTE Time Synchronisation

Using a RTE to support the satellite development requires a common time base for all entities which are contained in a satellite test bench. As well as the common time base an accurate synchronisation of the several entities is needed. This synchronisation is normally based on the synchronisation strobe of the satellite called PPS (Pulse per second).
The RTE is synchronised to this PPS using a specific Time & Synchro module which consist of a time generator, a switch and converter unit and an event input module located in the Real Time Linux system.

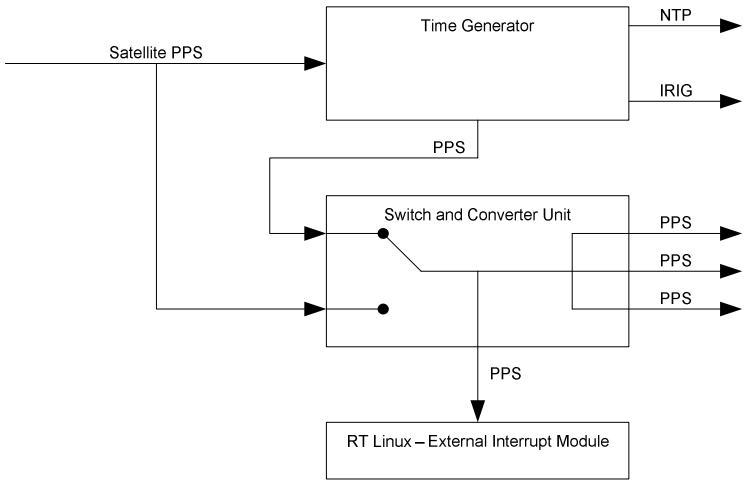

Fig. 4. Linux platform synchronization mechanism with external PPS

The time generator is driven by the satellite PPS and provides, based on this PPS, a time distribution service using NTP (Network Time Protocol) and a time service using the IRIG Timecode protocol (Inter Range Instrumentation Group Timecode). NTP and IRIG are used to synchronise the absolute time of all involved bench entities. This bench reference time (BRT) is normally set to UTC (Universal Time Coordinated).
A third service of the time generator is a PPS output. This PPS output is directly looped through the generator based on the satellite PPS. In case of an absence of the satellite PPS the time generator is able to provide its own PPS. Which means that the complete bench could be time synchronized without the need of a satellite PPS.
The above mentioned switch and converter unit (SCU) is used to distribute the PPS coming from the satellite or time generator to all relying entities (e.g. Simulator, SCOEs). The SCU is also used to do signal conversions for the dependent PPS sinks. Typical output signal types of the SCU are RS422, LVDS or TTL.
The simulation itself is synchronized to one of SCU outputs. This SCU output is connected to a real time interface module in the Real Time Linux PC. This interface module provides several interfaces for external interrupt inputs as well as several interrupt output interfaces. One of the external interrupt inputs is used to receive the PPS and provide it to the simulation software running on Real Time Linux. The PPS information provided by the real time interface module is used the drive the scheduler of the simulator.

SIMEFM USE CASE

Full numerical simulators (also called numerical benches) are used all along the satellite Functional Validation and are based on a numerical and representative OBC model. SIMEFM configuration is a mix of a numerical simulator and a part of RTE (from EFM or PFM) in order to interface real equipment. With this configuration we are able to operate real equipment (typically over a 1553 bus) and to realise different type of test, up to complete AOCS close loop tests, without the use of a real OBC.
This, therefore, allows to 'spare' the cost of the real OBC and their necessary hardware interfaces. Thus, a "light" & low-cost EFM can be set up for early characterisation, integration or validation of equipment.
Various extensions of this use case are identified:

- Equipment (AOCS, payload…) caractérisation
- Early coupling activities between equipment and their external stimulation SCOE (e.g. STR, Rx GPS…)
- P/L interface compatibility integration
- Preparation of Flight Model integration procedures
- AOCS – FSW early validation
- Validation of test sequence for different AIT tests (in a limited configuration)
- AOCS Functional validation with real equipment
- FSW maintenance (investigation on flight problems with some equipments)

The SimEFM is build around a complete numerical satellite simulation, including the OBC model, interfacing with external equipments via a real 1553 interface board and synchronisation mechanism. Thus, SimEFM provides a full test environment allowing to:

- Issue telecommands from the operator workstation (CCS) and to route them towards the equipment under test (payload or platform equipment) via 1553 bus
- Acquire Equipment telemetry as seen from the satellite platform
- Distribute synchronisation (PPS or other) to equipment under test (if necessary)
- Acquire/stimulate (depending of equipment) electrical signals at equipment level. These information are then sent to equipment/models and dynamic model (for close loop test).

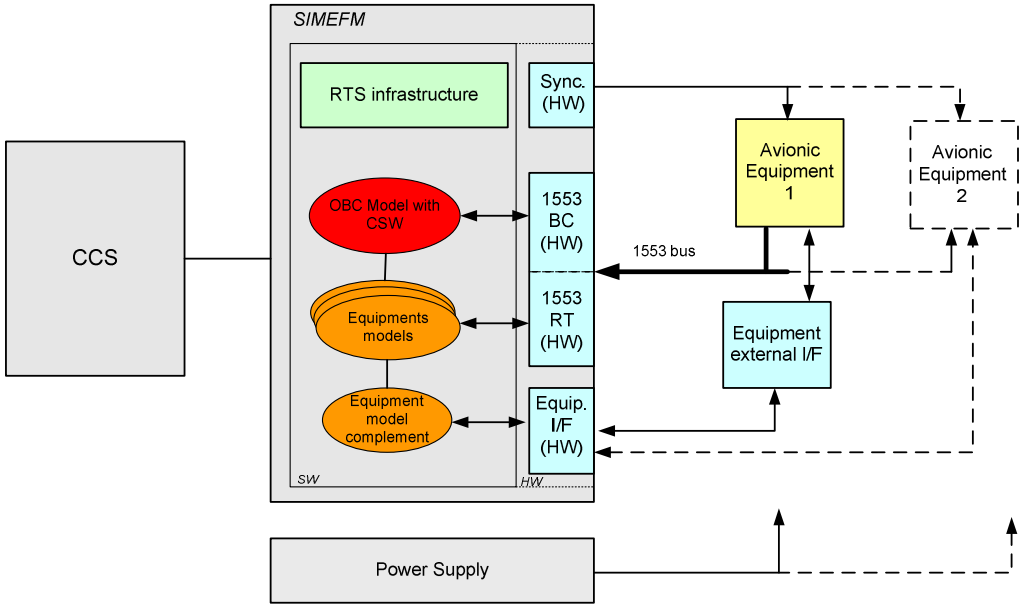

Fig. 5. SIMEFM architecture, based on numerical simulator coupled to hardware interfaces

Numerical simulators use also Linux platform, but real time capabilities are not mandatory for all their use cases. Introducing hardware (such as 1533 interface) in the SIMEFM configuration imposes real time performances at interfaces level in order to respect real time timing on exchanges with the external equipments. Using Real Time Linux configuration as described above for SIMEFM allows to implement hardware interfaces without any modification in the overall numerical simulation.
In addition to real time constraints for interface, SIMEFM configuration also requires to execute OBC model faster than real time. This is necessary to be able to preserve sufficient time for hardware exchanges programming.
Up to date PC multi-processors platform gives this significant computing power to solve this constraint and this with efficient real time performances.

CONCLUSION

The previously explained Real Time Linux architecture for different hybrid simulators shows that this solution is comparable in term of real time performance to the PowerPC/VxWorks based configurations. In this Linux based architecture, legacy VME I/O cards are reused together with their drivers thanks the transparent PCI – VME memory access provided by the bridge.

Real Time Linux provides powerful and scalable computing power (multi core & multi processor with the highest frequency) given a cost effective ratio. The introduced system is meeting all the needs for hybrid simulation like real time capability, possibility to synchronize to an external event source and adequate computation power.

Moreover, using Linux for both hybrid and full numerical simulators is the warrant of functional computation consistency: the simulation models in all cases run on the same processor/operating system/compiling chain. Today, we reach the point where models libraries are binary compatible in all simulators configuration.