Electronic system-level development: Finding the right mix of solutions for the right mix of engineers.

Nowadays, System Engineers are placed in the centre of two antagonist flows: microelectronic systems are increasingly complex whilst the time budget for development is constantly shrinking. Even if any 'microelectronic system' comprises a more or less integrated mix of hardware and software, it is not obvious that there is a common answer to this new productivity challenge. Why is that? Because there are different types of systems and different types of system engineers.

Any **electronic system-level (ESL) development approach** aims at developing a system at an abstraction level located 'above' the traditional hardware (RTL) and software levels. At the end of the process the specified system functionalities are 'optimally' partitioned onto a set of hardware and software computational resources. This 'top-down' approach opposes to traditional 'bottom-up' methodologies, where the software is built after hardware availability, with very little interactions between the software and the hardware development teams. Before ESL has become a major concern to fill the so-called 'electronic productivity gap', engineers have been developing systems for long, often with a mix of a top-down and bottom-up approaches. After all, ESL is nothing but translating the system specifications on a very formal way. Good and well-implemented ESL techniques are believed to be a solution to make better system, in shorter time, that ensures a good legacy for system evolution. Whereas this concept is well understood, nobody would claim that ESL methodologies are today widely deployed or that there is one single way to implement them. To further analyse this, let's have a closer look at who the system engineer is, and what type of system he develops.

Different system engineers

A **system engineer** always manages hardware and software. However, given the traditional way to organise hardware and software teams separately in companies, system engineers are likely to have different hardware and software concerns, according to their past experiences and education. In an attempt to characterise the system engineer community, we may distinguish the 3 following groups:

- SoC (system-on-chip) engineer;
- Embedded system engineer;
- System-on-board engineer.

In the first group, the **SoC engineer** mostly comes from the ASIC industry; as such, he is mainly a hardware engineer, used to cycle-accurate RTL design in Verilog or VHDL, and familiar with semiconductor manufacturers design flows. This one has been the main target of EDA vendors, for products like synthesiser or static timing analysis tools. His primary role has been to specialise hardware to grant performance to a given application. Now, because most of new ASIC designs turn out to be SoC designs, he is being asked to take software into account in his quest of an optimal system. In SoC (and also ASIC design), mistakes can be very expensive in time and money; therefore, the SoC engineer remains 'hardware-centric'.

In the second group, the **embedded system engineer** mostly comes from the software world. He has a bottom-up approach, starting from standard hardware components, generally with a software platform (OS or middleware) and puts applications on top of that – mainly in

Byte Paradigm – info@byteparadigm.com – www.byteparadigm.com
Chaussée de Namur, 119, bte 1 – B-1402 Nivelles - Belgium

C/C++. Because he develops software for a specialized set of microprocessor and peripherals, he may be described as a 'software designer with a deep hardware concern'. This hardware concern is more about choosing the right 'off-the-shelf' components than having a truly dedicated piece of hardware. This engineer is familiar with software emulation, debug techniques through ICE (in-circuit emulator) or other OCD (on chip debug) resources. In embedded system design, most mistakes may be fixed with a software update. However, the choice of the hardware remains crucial to guarantee the intrinsic performance of the system and its evolution.

System type	Engineer
SoC	HW-centric
Embedded System	SW-centric
System-on-board	HW / SW / PCB /… 'system-centric?'

Table 1 : Types of systems and system engineers

The third group, the **system-on-board engineer**'s group, is somewhat more blurry. Actually, this is the only group that has *always* been busy with system development. To simplify, they are where ASIC is overkill and where embedded systems are not specialised enough. They do both software and custom hardware (mostly CPLD and FPGA); they select off-the-shelf components like microcontrollers, memories and peripherals; they design PCBs and test them. They are proficient in schematics drawing, RTL and (low-level?) software coding. They are used to board measurements, and functional debug. They use parts of the ASIC design flow techniques (such as synthesisers) and are familiar to microprocessor application development and debug.

System design productivity is not a matter of language only

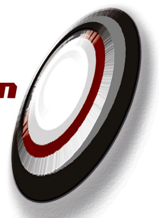
Together with the recent interest to implement efficient ESL methodologies, it is somewhat regrettable that the system development productivity problem is often reduced to a choice between ESL languages. It is no doubt that there is a real hype around SystemC, System Verilog and other extended VHDL languages adoption. It is true that engineers are looking for improving the way they describe, analyse and validate their system designs. This probably requires the use of higher-level language, in replacement to – or side by side with - the traditional VHDL, Verilog or C/C++. However, looking at its roots, the system development productivity gap cannot be limited to a matter of language only. These roots are numerous – we chose to describe 2 of them.

Functional verification time is extended.

With today's design complexities, it is not uncommon that system engineers spend from 40% to 70% of their design time for functional verification and debug. There are many origins to this serious bottleneck:

- More powerful technologies (more MIPS, more gates/mm², higher bandwidths) bring more complex systems, with more functionalities. Checking them all, and the way they interact naturally demands more development time. Aside, a more complex specification also brings more potential interpretation errors – and hence more functional bugs to solve.
- To cope with shorter products lifetime and reduced market window opportunities, system developers cannot afford to design *everything* in-house. As a consequence, a system heavily relies on third-party functional blocks, such as silicon intellect properties (IPs), standard components or standard board modules. Because the

Byte Paradigm – info@byteparadigm.com – www.byteparadigm.com
Chaussée de Namur, 119, bte 1 – B-1402 Nivelles - Belgium



system engineer does not master all the details of these third-party functionalities, his work time shifts from *designing them* to *verifying them*. In theory, the system engineer gains at reusing 3rd party functionalities… *only if* the system engineer has checked that:

- the 3rd party solution is bug-free;
- the 3rd party solution correctly 'fits' (performance, interface) within the system; in other words, if he has validated that the 3rd party solution corresponds to his system architecture.

… In the reality, the 2 above conditions are really problematic.

- Traditional hardware RTL cycle-accurate simulations require increasing simulation run-times to functionally check a multi-million gates system; they are often be limited to a few boot code cycles. If RTL simulation is used, the validation time is lengthened; the functional coverage of it remains low in comparison with the overall system complexity.

Hardware and software development teams are mostly separate.

Past bottom-up system design methodology shaped the organisation of most system houses. As a result, the system hardware development team is used to working independently from the system software development team and often keeps on working so.

With today's pressures on overall system design time, nobody can afford to completely start developing software *after* hardware. This is especially true for SoC, where it is crucial to verify the smooth integration of software with the hardware *before* this one is actually available. To a less critical extend it is also true for systems developed with high-end FPGAs, that integrate a microprocessor. Hardware and software development *must* be seen as a concurrent and parallel process, ideally with bidirectional communications between the hardware and software development teams. This is where actually ESL languages pop-up as *supporting tools*. It makes no sense for hardware and software engineering communities to keep on fighting for VHDL/Verilog against C/C++. It is very unlikely that an embedded software designer will suddenly use SystemC for his next application development. Nevertheless, ESL languages may gain the following essential roles:

- They can be used as a formal system specification description language. As such, they'd help limit the interpretation errors of the system specification.
- They can be used to define models and interfaces between the system hardware and software elements, so that a real top-down concurrent development may take place.

The bottom line is that the choice of ESL methodologies and tools must take the very specific characteristics of an organisation into account. We have already seen that the 'system engineers' community is not homogenous. It is likely that a given set of ESL methodology and tools will not be efficient in *any* company. Why would it be otherwise, as the basic challenge is an interaction problem?

Finding the right mix of solutions

As previously stated, there is no single system engineer, and there is no single type of microelectronic system. Nonetheless, there is the same challenge for everyone: **accelerate functional verification and debug**; and this, the earliest in the design process. Now, let's have a look at where this happens in the system development process.

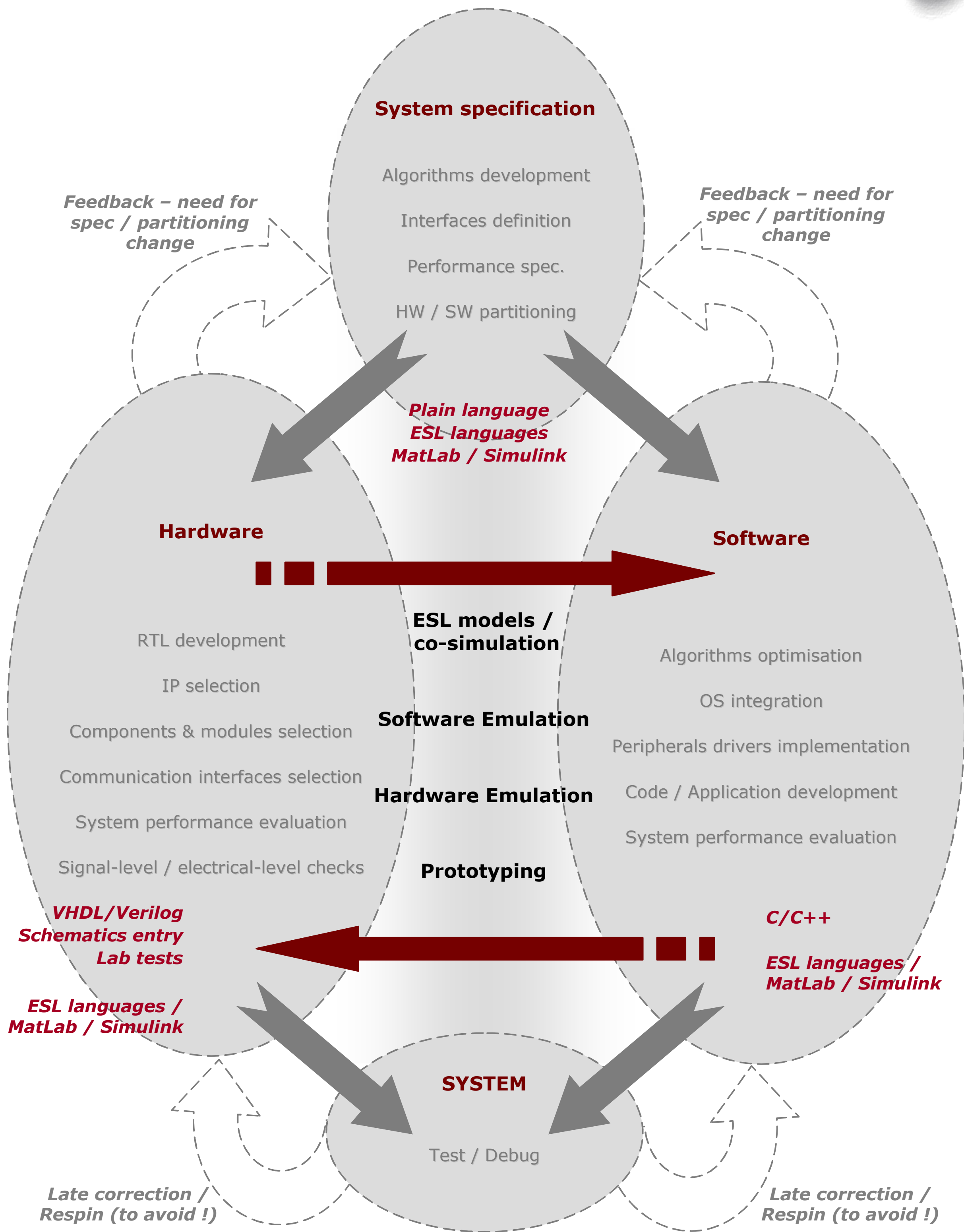


Figure 1 : Synthetic view of a system design flow

Figure 1 proposes one synthetic view of a system top-down design flow. It all starts with defining the system specification: basically, the various algorithms and functionalities required by the system, the way they interact (definition of the function interfaces and the essential system data flows), and also the

definition of what the performance of the system should be. This phase results in the hardware / software partitioning. That is, a distribution of the system functionalities onto a set of hardware and software computational resources.

Specifying a system has always been the world of plain language, pen-and-paper engineering, and functional architecture drawing. Today, ESL languages are proposed, in an attempt to formalise the system algorithms and architecture description while avoiding 'freezing' it into an arbitrary hardware/software partition. While it is commonly agreed that the system specification would benefit from such a formal approach – given the growing system functional complexity – it is not clear however if a given ESL language will be widely adopted. Still, engineers throughout the world *are* designing systems, and often with success.

As previously stated, the **system-on-board engineers** always have. Not surprisingly, some of them have put in place tools to better specify systems from the start. The most representative examples of this are **Matlab** and **Simulink**, from The MathWorks. Matlab is a very powerful and intuitive modelling language that makes it easy for designers to quickly model their algorithmic functionalities; today, it also offers bridges to develop embedded code and/or RTL code from its M language. As such, it is not generally described as a real 'ESL language'… but as we previously stated, ESL methodologies is not just a matter of language: the point is efficiency and productivity.

Once the system specification is stable enough, it is to be decided how to partition it between hardware and software. Depending of the system, this partition may be reviewed in a later step, according to the feedback of the **validations** performed by the hardware and software development teams. As stated before, this has become a critical point: how can HW and SW guys team up to ensure a productive system functional verification?

Table 2 : System functional validation solutions

Validation solution	*Use*	*Advantages*	*What can be better*	*Type of system*
RTL simulation	Custom hardware validation; IP evaluation.	System hardware internal visibility, cycle-accurate results.	Excessive run-times, preventing from simulating 'real' SW and HW together.	SoC, System-on-board Limited for embedded systems (co-processors, companion chips and glue logic)
ESL models / co-simulation	Delivers hardware functional models at a higher abstraction level than RTL for software development. Ex: ISS model of a processor (Instruction Set Simulator). Allows the simulation of HW and SW in a common environment.	Allows viewing/simulating hardware from a functional viewpoint; SW/HW simulation speed.	HW/SW teams organisation. No ESL language generally adopted.	SoC, System-on-board

Byte Paradigm – info@byteparadigm.com – www.byteparadigm.com
Chaussée de Namur, 119, bte 1 – B-1402 Nivelles - Belgium

Validation solution	*Use*	*Advantages*	*What can be better*	*Type of system*
Software emulation	Embedded software development and debug. System debug.	Uses real processor at (near) system speed; can be used to functionally validate and debug µP-based HW environment; can be part of a system prototyping approach.	Only offers a visibility on the system through the processor (!)	SoC, Embedded System, System-on-board
Hardware emulation	System hardware validation	Internal hardware visibility, re-usability, acceptable run speed.	Setup length, user friendliness, poor software validation abilities, cost.	SoC
Prototyping	Validation of 3rd party functionalities (IPs, modules …). Explore technology options; provides a first 'draft' of what the actual system will be.	Validation of 3rd party elements (IPs, modules, …); (near) real system speed execution. Functionally very close to the system being developed. Cost. Can comprise software emulation as well.	HW visibility / HW debug abilities, prototype reuse, and setup length.	SoC, Embedded System, System-on-board

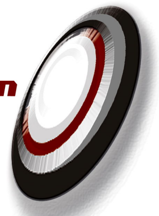
The above table summarises the major available (or available soon) methodologies for functional validation. The following points should be noted:

- In the 'What can be better' column are quickly summarised the drawbacks of each methodology *from a system development viewpoint.* For instance, software emulation only offers a visibility on the system through the processor… well, that's actually what the software emulation is meant for! The conclusion in this case, is that software emulation can be complemented with other techniques in order to gain better system visibility.
- It is no doubt that a lot of drawbacks preventing one given methodology from being widely applied have non-technical roots. For instance, the business models in the embedded system world are somewhat different from these of the SoC world, with the consequence that there may be some delay for a 'SoC methodology' to be accepted by embedded system engineers.
- Lots of actual system development methodologies are in fact a mix of several approaches. Table 1 summarises the dominant trends, and is not exhaustive.

Which key information does this inventory hold?

First of all, a given system development technique does not apply to any system.
Let's take hardware emulation as an example.

Byte Paradigm – info@byteparadigm.com – www.byteparadigm.com
Chaussée de Namur, 119, bte 1 – B-1402 Nivelles - Belgium



Originally, hardware emulation aimed at providing a fast validation technique for big ASIC designs. Hardware emulation generally provides a unified software/hardware environment to map a logic-equivalent version of the future hardware onto an array of programmable logic devices, such as FPGAs. The emulator offers many debug options to generate test benches and apply the stimuli's onto the mapped hardware. This approach greatly increases the validation speed in comparison with the classical RTL / gate level simulations, while maintaining a high hardware internal visibility. Modern hardware emulation systems bring new features to help software debug, and are progressively oriented towards real 'system emulation'.

Hardware emulation can be particularly helpful for complex SoC designs; it is less relevant for system-on-board and almost useless for embedded systems:

- Developing a system-on-board implies *de facto* the development of one or several boards, used as prototype during development to validate both the architecture and the PCB design issues.
- The central point for embedded systems is the embedded software. 'Classical' software emulation offers a more efficient development environment than hardware emulation systems.

➔ Because prototyping has to be used in these 2 cases, adding hardware emulation to the overall system development methodology does not bring sufficient added value[1] in general.

Second of all, functional system validation is all about finding the right mix of techniques, according to the system being developed and the available engineering resources.

It is interesting to see that the major hold back to implementation of recent ESL methodologies is **not** that engineers *don't want* ESL. This is actually what they use when describing their specification with Matlab. This is actually what they do when they prototype a system-on-chip on a board with FPGAs and debug hardware/software interactions through software emulation.

In brief, **there is currently no technique that 'fits them all'**. Once again, this is due to the wide variety of systems and the wide variety system development teams.

In the subsequent sections, we'll focus on **prototyping** as a system validation technique and check what can be improved for better system development productivity.

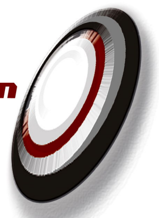
Improving prototyping

Today, **software emulation for embedded system development** is one of the most successful applications – yet very specialised – of the prototyping as a system validation technique. Let's examine its main characteristics:

- With SW emulation, the functional system validations are conducted under **conditions very close (if not identical) of those of the definitive system**. Bugs related to a bad modelling can be avoided. Validations are run at (near) system speed, shortening the overall validation time. This enables a very good development and validation productivity.
- SW emulation presents a **good integration between the system engineer development environment and the prototype**. Practically, the developer never

[1] Aside, the price and the cost of use of hardware emulation probably do not justify its use in these cases.

Byte Paradigm – info@byteparadigm.com – www.byteparadigm.com
Chaussée de Namur, 119, bte 1 – B-1402 Nivelles - Belgium



quits its familiar code development and debugging environment, even if the code is run on the embedded platform itself.

- SW emulation is possible because the **system hardware is available with on-chip debug features accessible through proprietary or standard (e.g. JTAG) interfaces**.

Nevertheless, software emulation has unique characteristics that ease a prototyping approach:

- An important part of the hardware (embedded microprocessor and peripherals) is mostly standard, predictive (it is well known early in the design that a given µP and a set of peripheral will be enough for the system) and limited (the functional hardware is concentrated on very few devices). Many standard development kits are available, and there is very little risk for the prototype HW resources to be under evaluated.
- Embedded systems are processor-centric. Despite specialised functionalities for which a custom hardware may be necessary, most of the functionality is added with software on top of an OS or a given firmware. **Using the processor as the main (and unique) access point often brings enough system visibility to complete the validation process.**

System prototyping generally implies developing a board that is functionally the most equivalent to the targeted system. Sometimes, the prototype is very close to the definitive system (as for system-on-board, where it is also a first draft of the future PCB); sometimes it sacrifices some performance for functional verification (as for the prototyping of a SoC with a FPGA and a stand-alone microprocessor).

As previously stated, the 3 major drawbacks of system prototyping (the embedded system SW emulation solution put apart) are:

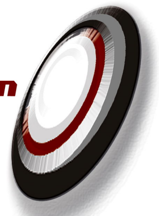
1. Poor prototype **reusability.**
2. Excessive prototype **setup length.**
3. A lack of **visibility** for the system debug.

The **reusability** issue is unfortunately difficult to address with a system prototyping approach. Basically, if the system requires a specialised hardware development, either the prototype will be excessive in terms of computational resources, or it will just fit the resources required for the development. In the first case there is just a better chance that the same prototype will fit for another development. Using programmable devices such as FPGA with numerous IOs and a good package 'forward compatibility' can improve the prototype reusability. Foreseeing standard connectors on which functional piggy-back extensions can be plugged can also guarantee the prototype legacy. Nevertheless, the prototype reusability is an investment protection issue to be examined together with the product evolution strategy.

The **prototype setup length** issue mostly comes from the fact that there is fundamentally a rupture between the development environment and the prototype validation environment. The latter is the world of labs, with intensive use of multimeters, oscilloscopes, logic analysers, and protocol analysers. The development environment is the world of test benches, RTL simulators, where stimuli's and analysis programs can be easily developed. In general, continuity between development and prototype functional validation environment does not exist, as for the software emulation.

The **visibility issue** comes side by side with the prototype setup length**.** Whereas having one single access point is enough for software emulation, prototype system hardware validation does not offer the same visibility inside the design. Of course, hardware validation and debug

Byte Paradigm – info@byteparadigm.com – www.byteparadigm.com
Chaussée de Namur, 119, bte 1 – B-1402 Nivelles - Belgium



often requires observing hardware at gate level and with clock cycle accuracy. Similarly to embedded processor, developing a custom hardware with good prototyping qualities requires to foresee the adequate access points. For example, this can be a dedicated port IP in a FPGA, connected to a few debug pins; systems busses can be left open, and connected to a board debug connector; dedicated access to registers and memories can be available to be able to monitor a system state.

As a conclusion, to benefit the most from prototyping as a system validation methodology it requires:

- Multiplying the prototype access points to increase the prototype visibility.
- Having the adequate set of tools to benefit from this new visibility while holding the prototype setup length as short as possible.

Bottom line

Electronic system development faces today an important problem of productivity. Improving it certainly requires finding new methodologies and it makes no doubt that a real ESL approach can help[2]. Not surprisingly, system validation has become one of the most critical problems that should be addressed to increase the development productivity. Given the increasing complexity of current systems, traditional validation techniques progressively show their limitations. Another important element is that 'validation' is a point where the hardware and software teams meet, requiring a harmonious system-oriented methodology in order to really validate the 'system' as a whole and benefit from both SW and HW viewpoints.

Some techniques and (partial?) solutions already exist for a real system validation and debug. Some others still need maturity. Nevertheless, solving the 'validation problem', and hence, part of the 'system productivity problem' likely requires finding the adequate mix of solutions. Why is that? Because there are different types of systems and different types of system engineers.

At Byte Paradigm, we believe that successful system development requires a specific approach that best fits your company organisation. We also believe that its success depends on the good association of complementary techniques, especially for the system validation. If they are improved, these techniques will contribute to increase your system development productivity.

For instance, increasing system prototype visibility by offering efficient access points and a real continuity between the system development environment and the system prototype would greatly enhance your prototype validation.

Byte Paradigm delivers PC instruments to test and debug electronic and embedded systems. Byte Paradigm sees tests and debug on real hardware as one of the key elements to efficient system development.

http://www.byteparadigm.com.

About the author
Frédéric Leens is Sales and Marketing Manager at Byte Paradigm.
He can be reached at: frederic.leens@byteparadigm.com

[2] And as we have seen with the successful use of Matlab as an algorithmic abstraction layer for system-on-board design, an ESL approach is not limited to chosing the right 'ESL language'.