

# Automotive Electronics: Trends and Challenges

**Alberto Sangiovanni-Vincentelli**

The Edgar L. and Harold H. Buttner Chair of Electrical Engineering and Computer Science, University of California at Berkeley

Copyright © 2000 Society of Automotive Engineers, Inc.

## ABSTRACT

The car as a self-contained microcosm is undergoing radical changes due to the advances of electronic technology. We need to rethink what a "car" really is and the role of electronics in it. Electronics is now essential to control the movements of a car, of the chemical and electrical processes taking place in it, to entertain the passengers, to establish connectivity with the rest of the world, to ensure safety. What will an automobile manufacturer's core competence become in the next few years? Will electronics be the essential element in car manufacturing and design? We will address some of these issues and we will present some important developments in the area of system design that can strongly impact the way in which a car is designed.

## INTRODUCTION

The world of electronics is witnessing a revolution in the way products are conceived, designed and implemented. The ever growing importance of the web, the advent of microprocessors of great computational power, the explosion of wireless communication, the development of new generations of integrated sensors and actuators are changing the world in which we live and work. The new *key words* are:

- *transparent electronics*, i.e., electronics has to be invisible to the user; it has to help unobtrusively.
- *pervasive computing*, i.e., electronics is everywhere; all common use objects will have an electronic dimension.
- *intelligent environments*, i.e., the environment will react to us with the use of electronic components. They will recognize who we are and what we like.
- *wearable computing*, i.e., the new devices will be worn as a watch or a hat. They will become part of our clothes. Some of these devices will be tags that will contain all-important information about us.
- *write once, run everywhere*, i.e., any information we write down will be recorded and digested by the

intelligent environment. We will never have to enter the same information twice.

- *know more, carry less*, i.e., the environment will be able to know more about us so that we will not need to carry all the access paraphernalia we need today: keys, credit cards, personal I.D. access cards, access codes will soon be forgotten.

The car as a self-contained microcosm is experiencing a similar revolution. We need to rethink what a "car" really is and the role of electronics in it. Electronics is now essential to control the movements of a car, of the chemical and electrical processes taking place in it, to entertain the passengers, to establish connectivity with the rest of the world, to ensure safety. What will an automobile manufacturer's core competence become in the next few years? Will electronics be the essential element in car manufacturing and design?

The challenges and opportunities are related to

- how to integrate the mechanical and the electronics worlds, i.e., how to make mechatronics a reality in the automotive world,
- how to integrate the different motion control and power-train control functions so that important synergies can be exploited,
- how to combine entertainment, communication and navigation subsystems,
- how to couple the world of electronics where the life time of a product is around 2 years and shrinking, with the automotive world, where the product life time is 10 years and possibly growing.
- how to develop new services based on electronics technology
- how are the markets shaping up (for example, what will be the size of the after-market sales for automotive electronics?)

We will pose these questions while quickly reviewing some of the most important technology and product developments of the past few years. The entire electronics industry is undergoing a major restructuring that is favoring horizontal integration and vertical disintegration. In this framework, we point out how collaboration among

different industry segment is essential to bring new products to market. The main part of the paper is about system design methodology that focuses on two main principles: orthogonalization of concerns and platform-based design. System design is definitely the most important technology to master to increase the quality and the value of the electronic component of the car.

## CONSIDERATIONS ON THE FUTURE OF ELECTRONICS AND INTEGRATED CIRCUITS

### DEVICES, INFRASTRUCTURE AND BUSINESS ORGANIZATION

By the year 2002, it is estimated that more information appliances will be sold to consumers than PCs (*Business Week*, March 1999). This new market includes small, mobile, and ergonomic devices that provide information, entertainment, and communications capabilities to consumer electronics, industrial automation, retail automation, and medical markets. These devices require complex electronic design and system integration, delivered in the short time frames of consumer electronics. The system design challenge of the next decades is the dramatic expansion of this spectrum of diversity. The introduction of small, low-power, *embedded* devices will accelerate, as microelectronic mechanical system (MEMS) technology becomes available. Microscopic devices, powered by ambient energy in their environment, will be able to sense numerous fields, position, velocity, and acceleration, and communicate with substantial bandwidth in the near area. Larger, more powerful systems within the infrastructure will be driven by the continued improvements in storage density, memory density, processing capability, and system-area interconnects as single board systems are eclipsed by complete systems on a chip.

*Data movement and transformation is of central importance in such applications.* Future devices will be network-connected, channeling streams of data into the infrastructure, with moderate processing on the fly. Others will have narrow, application-specific UIs. They will be highly adaptable and configured automatically, and yet provide strong guarantees of availability and performance. Applications will not be centered within a single device, but stretched over several, forming a path through the infrastructure. In such applications, the ability of the system designer to specify, manage, and verify the functionality and performance concurrent behaviors is essential.

Electronics in the automotive domain has been growing by leaps and bounds. There are three basic domains for the electronics for the car:

- Power-train management;
- Body electronics, including dashboard and temperature control;

- Infotainment, a horrible neologism for the electronic subsystems devoted to the information processing, communication with the outside world and entertainment.

The first domain is very typical of a transportation system and is characterized by tight safety and efficiency constraints. Control algorithms are core competence together with software and mixed mechanical-electrical hardware design and implementation.

Body control involves the management of a distributed system that resembles more and more a network with protocols that are likely to have different requirements than standard communication protocols. Guaranteed services are the essence here.

The infotainment system is more amenable to side effects from other industrial domains that are growing and progressing in the technology race at a faster rate than what I have seen in the automotive domain. This is the domain where most profits are to be gained in the short-medium term. Customers are now starting to make their buying decision with an eye more towards the performance of its engine and its handling. Hence, the essential question of what will be the core competence of a car company. Will the electronic components be “the car” and the mechanical components an accessory?

A recent quote from Daimler-Chrysler executives says that more than 80% of innovation in the automotive domain will be in electronic components. It is then clear that this domain will be the battlefield among players in different industrial segments: consumer electronics, wireless communication, car electronics, car manufacturing. I believe that the winners will be able to conjugate the knowledge of the car as a microcosm with the knowledge of electronic design and technology. Strategic relationships have already been formed to tackle the most difficult problems. I also believe that the interaction and the potential integration of the three environments will be essential to propose interesting solutions.

### SUPPLIER CHAIN FOR ELECTRONIC SYSTEMS

No single company will be able to master the design and the production of these complex devices and of the necessary infrastructure. Due to more and more stringent time-to-market constraints, the electronics industry is restructuring so that system level design companies like Nokia focus more and more on product definition and marketing, while outsourcing the design and engineering of the actual products to specialized design services companies like Cadence Design Systems. The design of system products is itself the result of the collaboration among Intellectual Property (IP) providers and system integrators. The production is then deferred to specialized manufacturing companies thus resulting in substantial

investment savings and better time-to-market. This trend has been visible for the past few years in board manufacturing where companies like Solectron and Flextronics have been taking the lead in contract manufacturing but is now also quite popular in the Integrated Circuit domain where foundries like TSMC, UMC and Chartered Semiconductors are growing at a very fast rate, fueled by the success of fab-less semiconductor companies and by the interest of system companies like Cisco to invest in proprietary design technology independent of standard semiconductor companies. I believe that the companies that will be able to leverage maximally this new business organization will be the ones to dominate.

To be able to leverage this situation, a new design flow has to be put in place. The importance of clean interfaces and unambiguous specifications will be essential. In addition, the thorny issue of IP protection and IP property has to be addressed. In the automotive domain, this is even more important as the supplier chain is deeper than in other industrial segments. Automotive manufacturers such as Mercedes and BMW must manage internal development groups as well as outside groups like Bosch who then work in collaboration with semiconductor companies and possibly with printed circuit board manufacturers. Hence, best business practices have to be developed if we do not want to enter into an era even more litigious than the present one.

## ELECTRONIC SYSTEM DESIGN: ISSUES AND SOLUTIONS

The overall goal of electronic embedded system design is to balance *production costs with development time and cost in view of performance and functionality considerations*. Manufacturing cost depends mainly on the *hardware components* of the product. Minimizing production cost is the result of a balance between competing criteria. If one considers an integrated circuit implementation, the size of the chip is an important factor in determining production cost. Minimizing the size of the chip implies tailoring the hardware architecture to the functionality of the product. However, the cost of a state-of-the-art fabrication facility continues to rise: it is estimated that a new 0.18 $\mu\text{m}$  high-volume manufacturing plant costs approximately \$2B today.

In addition, the NRE costs associated with the design and tooling of complex chips are growing rapidly. The ITRS predicts that while manufacturing complex System-on-Chip designs will be practical, at least down to 50nm minimum feature sizes, *the production of practical masks and exposure systems* will likely be a major bottleneck for the development of such chips. That is, *the cost of masks will grow even more rapidly for these fine geometries, adding even more to the up-front NRE for a new design*. A single mask set and probe card cost for a state-of-the-art chip is over \$.5M for a complex part today [1], up from less than \$100K a decade ago (*note: this does not*

*include the design cost*). At 0.15 $\mu\text{m}$  technology, SEMATECH estimates we will be entering the regime of the "million dollar mask set." In addition, the cost of developing and implementing a comprehensive test for such complex designs will continue to represent an increasing fraction of a total design cost unless new approaches are developed. These increasing costs are strongly prejudicing manufacturers towards *parts that have guaranteed high-volume production form a single mask set* (or that are likely to have high volume production, if successful.) Such applications also translate to better response time and higher priorities at times when global manufacturing resources are in short supply.

As a consequence of this evolution of the Integrated Circuit world, if one could determine a common "hardware" denominator (which we refer to as a hardware *platform*) that could be shared across multiple applications, production volume increases and overall costs may eventually be (much) lower than in the case when the chip is customized for the application.

Of course, while production volume will drive overall cost down by amortizing NRE, it is important to consider the final size of the implementation as well since a platform that can support the functionality and performance required for a "high-end" product may end up being too expensive for other lower-complexity products. Today the choice of a platform architecture and implementation is much more an art than a science. *We believe that a next-generation, successful system design methodology must assist designers in the process of designing, evaluating, and programming such platform architectures, with metrics and with early assessments of the capability of a given platform to meet design constraints*

As the complexity of the products under design increases, the development efforts increase dramatically. At the same time, the market dynamics for electronics systems push for shorter and shorter development times. It will be soon imperative to keep to a strict design time budget, no matter how complex the design problem, with as little as six months from initial specification to a final and correct implementation. To keep these efforts in check and at the same time meet the design time requirements a design methodology that favors reuse and early error detection is essential. The use of programmability as a mechanism for making low-cost, in situ design iterations is also very important in these situations. In this regard, we expect the majority of high-volume platforms developed to be programmable, either at the logic/interconnect level (e.g. via FPGA) or using software. However, as explained in more detail later, conventional Von Neumann architectures are unlikely to be sufficient to meet the power, performance and cost targets of this next generation of electronic systems. *Fundamental, new approaches to the programming of silicon-based systems must be developed and deployed.*

Both reuse and early error detection imply that the design activity must be defined rigorously, so that all phases are clearly identified and appropriate checks are enforced. To be effective, a design methodology that addresses complex systems must start at high levels of abstraction. In most of the embedded system design companies as well as IC design companies, designers are familiar with working at levels of abstraction that are too close to implementation so that sharing design components and verifying designs before prototypes are built is nearly impossible. Today, most IC designers think of the highest level of abstraction for their design an RTL language description. For embedded system designers, assembly language or at best C language is the way to capture and to implement the design. These levels are clearly too low for complex system design. The productivity offered by the expressive power of RTL languages is way below critical, lacking a support for software implementations. *In particular, we believe that the lack of appropriate methodology and tool support for modeling of concurrency in its various forms is an essential limiting factor in the use of both RTL and commonly used programming languages to express design complexity.*

Design reuse is most effective in reducing cost and development time when the components to be shared are close to the final implementation. On the other hand, it is not always possible or desirable to share designs at this level, since minimal variations in specification can result in different, albeit similar, implementations. However, moving higher in abstraction can eliminate the differences among designs, so that the higher level of abstraction can be shared and only a minimal amount of work needs to be carried out to achieve final implementation. The ultimate goal in this regard is to create a library of functions, along with associated hardware and software implementations, that can be used for all new designs. It is important to have a multiple levels of functionality supported in such a library, since it is often the case that the lower levels that are closer to the physical implementation change because of the advances in technology, while the higher levels tend to be stable across product versions.

We believe that it is most likely that the preferred approaches to the implementation of complex embedded systems will include the following aspects:

- Design time and cost are likely to dominate the decision-making process for system designers. Therefore, design reuse in all its shapes and forms, as well as just-in-time, low-cost design debug techniques, will be of paramount importance. Flexibility is essential to be able to map an ever-growing functionality onto a continuously evolving problem domain and set of associated hardware implementation options.
- Designs must be captured at the highest level of abstraction to be able to exploit all the degrees of freedom that are available. Such a level of abstraction

should not make any distinction between hardware and software, since such a distinction is the consequence of a design decision.

- *The implementation of efficient, reliable, and robust approaches to the design, implementation, and programming of concurrent systems is essential.* In essence, whether the silicon is implemented as a single, large chip or as a collection of smaller chips interacting across a distance, the *problems associated with concurrent processing and concurrent communication must be dealt with in a uniform and scaleable manner.* In any large-scale embedded systems program, concurrency must be considered as a first class citizen at all levels of abstraction and in both hardware and software.
- Concurrency implies communication among components of the design. Communication is too often intertwined with the behavior of the components of the design so that it is very difficult to separate out the two domains. Separating communication and behavior is essential to dominate system design complexity. In particular, if in a design component behaviors and communications are intertwined, it is very difficult to re-use components since their behavior is tightly dependent on the communication with other components of the original design. In addition, communication can be described at various levels of abstraction, thus exposing the potential of implementing communication behavior in many different forms according to the available resources. Today this freedom is often not exploited.
- Next-generation systems will most likely use a few highly complex (Moore's Law Limited) part-types, but many more energy/power-cost-efficient, medium-complexity ( $O(10M-100M)$  gates in 50nm technology) chips, working concurrently to implement solutions to complex sensing, computing, and signaling/actuating problems.
- These chips will most likely be developed as an instance of a particular platform. That is, rather than being assembled from a collection of independently developed blocks of silicon functionality, they will be derived from a specific "family" of micro-architectures, possibly oriented toward a particular class of problems, that can be modified (extended or reduced) by the system developer. These platforms will be extended mostly through the use of large blocks of functionality (for example, in the form of co-processors), but they will also likely support extensibility in the memory/communication architecture as well. When selecting a platform, cost, size, energy consumption, flexibility must be taken into account. Since a platform has much wider applicability than ASICs, design decisions are crucial. A less than excellent choice may result in economic *debacle*. Hence, design methods and tools that optimize the platform-selection process are very important.

- These platforms will be highly programmable, at a variety of levels of granularity. Because of this feature, mapping an application into a platform efficiently will require a set of tools for software design that resemble more and more logic synthesis tools. We believe this to be a very fruitful research area.

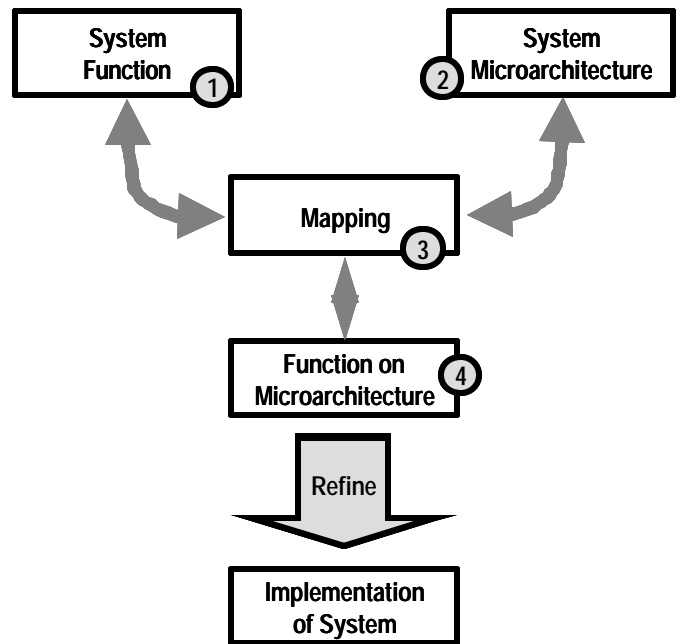
## SYSTEM LEVEL DESIGN METHODOLOGY

An essential component of a new system design paradigm is the *orthogonalization<sup>1</sup> of concerns*, i.e., the separation of the various aspects of design to allow more effective exploration of alternative solutions. An example of this paradigm is the orthogonalization between functionality and timing exploited in the synchronous design methodology that has been so successful in digital design. In this case, provided that the signal propagation delays in combinational blocks are all within the clock cycle, verifying the correct behavior of the design is restricted to the functionality of the combinational blocks thus achieving a major design speed-up factor versus the more liberal asynchronous design methodology. Others more powerful paradigms must be applied to system design to make the problem solvable, let alone efficiently so. One pillar of a design methodology that we have proposed over the years [2,3,4] is the separation between:

- Function (what the system is supposed to do) and architecture (how it does it);
- Communication and computation.

The mapping of function to architecture is an essential step from conception to implementation. In the recent past, there has been a significant attention in the research and industrial community to the topic of Hardware-Software Co-design. The problem to be solved here is coordinating the design of the parts of the system to be implemented as software and the parts to be implemented as hardware blocks, avoiding the HW/SW integration problem that has marred the electronics system industry for so long. We actually believe that worrying about hardware-software boundaries without considering higher levels of abstraction is the wrong approach. HW/SW design and verification happens after some essential decisions have been already made, and this is what makes the verification and the synthesis problem hard. SW is really the form that a behavior is taking if it is “mapped” into a programmable microprocessor or DSP. The motivations behind this choice can be performance of the application on this particular processor, or the need for flexibility and adaptivity. The origin of HW and SW is in behavior that the system must implement. The choice of an “architecture”, i.e. of a collection of components that can be either software programmable, re-configurable or

customized, is the other important step in design. We recall the basic tenet of our proposed design methodology (see Figure 1) next.



**Figure 1: Overall Organization of the Methodology**  
FUNCTION AND COMMUNICATION-BASED DESIGN

We say that a system implements a set of functions, where a function is an abstract view of the behavior of an aspect of the system. This set of functions is the input/output characterization of the system with respect to its environment. It has no notion of implementation associated with it. For example, “*when the engine of a car starts (input), the display of the number of revolutions per minute of the engine (output)*” is a function, while “*when the engine starts, the display in digital form of the number of revolutions per minute on the LCD panel*” is not a function. In this case, we already decided that the display device is an LCD and that the format of the data is digital. Similarly, “*when the driver moves the direction indicator (input), the display of a sign that the direction indicator is used until it is returned in its base position*” is a function, while “*when the driver moves the direction indicator, the emission of an intermittent sound until it is returned to its base position*” is not a function.

The notion of function depends very much on the level of abstraction at which the design is entered. For example, the decision whether to use sound or some other visual indication about the direction indicator may not be a free parameter of the design. Consequently, the second description of the example is indeed a function since the specification is in terms of sound. However, even in this case, it is important to realize that there is a higher level of abstraction where the decision about the type of signal is made. This may uncover new designs that were not

<sup>1</sup> We refer to *orthogonalization* (see orthogonal bases in mathematics) versus separation to stress the independence of the axes along which we perform the “decomposition”.

even considered because of the entry level of the design. Our point is that no design decision should ever be made implicitly and that capturing the design at higher levels of abstraction yields better designs in the end.

The functions to be included in a product may be left to the decision of the designer or may be imposed by the customer. If there are design decisions involved, then the decisions are grouped in a design phase called *function* (or sometimes feature) *design*. The decisions may be limited or range quite widely.

The description of the function the system has to implement is captured using a particular language that may or may not be formal. In the past, natural language was the most used form of design capture. The language used for capturing functional specifications is often application dependent. For example, for control applications, Matlab is used to capture algorithms. For several applications, computer languages such as C are used. However, these languages often lack the semantic constructs to be able to specify concurrency. We believe that the most important point for functional specification is the underlying mathematical model, often called model of computation.

As opposed to the informal nature of component-based design often used to design software today [5], we promote the use of formal models and transformations in system design so that verification and synthesis can be applied to advantage in the design methodology. In fact, verification is effective if complexity is handled by formalization, abstraction and decomposition [6]. Further, the concept itself of synthesis can be applied only if the precise mathematical meaning of a description of the design is applied. It is then important to start the design process from a high-level abstraction that can be implemented in a wide variety of ways. The implementation process is a sequence of steps that remove choice from the formal model. In other words, the abstract representation of the design should “contain” all the correct implementations in the sense that the behavior of the implementation should be consistent with the abstract model behavior. Whenever a behavior is not specified in the abstract model, the implicit assumption is that such behavior is a “don’t-care” in the implementation space. In the implementation domain, the abstract model is source of non-deterministic behavior. The implementation process progresses towards a deterministic system. It is important to underline that way too often system design starts with a system specification that is burdened by unnecessary references to implementations resulting in over-determined representations with respect to designer intent that obviously yield under-optimized designs.

In the domain of formal model of system behavior, it is common to find the term “Model of Computation”, a concept that has its roots in language theory. This term refers more appropriately to mathematical models that

specify the semantic of computation and of concurrency. In fact, concurrency models are the most important differentiating factors among models of computation. Edward Lee has correctly stressed the importance of allowing the designer to express designs making use of any such models of computation, or at least of the principal ones, thus yielding a so-called heterogeneous environment for system design. In his approach to simulation and verification, assembling a system description out of modules represented in different models of computation yields the problem of arbitrating communication among the different models. The concept of communication among different models of computation must be carefully explored and understood [7].

This difficulty has actually motivated our approach to *communication-based design*, where communication takes the driver seat in the overall system design methodology. In this approach, communication can be specified somewhat independently of the modules that compose the design. In fact, two approaches can be applied here. In the first case, we are interested in communication mechanisms that “work” in any environment, i.e., independent of the formal models and specifications of the behavior of the components. This is a very appealing approach if one emphasizes ease of component assembly. However, it is rather obvious that the designer may end up with an implementation that is quite inefficient, especially for high-volume embedded systems applications where production cost is very important. The other approach is to specify the communication behavior and then to use a successive refinement process for optimizing the communication, where the refinement process can leverage all that is known about the modules to interconnect. In this case, the correctness of the overall behavior is not insured by the communication mechanism but by the design process of the communication itself. In this case, a synthesis approach is most appealing since it reduces the risk of making mistakes and it may use powerful optimization techniques to reduce design cost and time.

The most important models of computation that have been proposed to date are based on three basic models: Finite State Machines, Data Flow and Discrete Event [7]. All models have their strengths and weaknesses. It is an important differentiating factor to be able to use these models at their best. Note that each model is composable (can be assembled) in a particular way that guarantees that some properties of the single components are maintained in the overall system. Communication and time representation in each model of computation are strictly intertwined. In fact, in a synchronous system, communication can take place only at precise “instants of time” thus reducing the risk of unpredictable behavior. Synchronous systems are notoriously more expensive to implement and often less performing thus opening the door to asynchronous implementations. In this latter case, that is often the choice for large system design, particular care has to be exercised to avoid undesired and

unexpected behaviors. The balance between synchronous and asynchronous implementations is possibly the most challenging aspect of system design. Globally-asynchronous-locally-synchronous (GALS) communication mechanisms are probably a good compromise in the implementation space [2].

The view of communication in these models of computation is sometimes at a level of abstraction that is too low. We would like to be able to specify abstract communication patterns with high-level constraints that are not implying yet a particular model of communication. For example, it is our opinion that an essential aspect of communication is loss-lessness. We argue that there must exist a level of abstraction that is high enough to require that communication take place with no losses. The synchronous-asynchronous mechanism, the protocols used and so on, are just implementation choices that either guarantee loss-lessness or that have a good chance of ensuring that no data is lost where it matters but that needs extensive verification to make sure that this is indeed the case. For example, Kahn's process networks [7] are important Data Flow models that guarantee lossless communication at the highest level of abstraction by assuming an ideal buffering scheme that has unbounded buffer size. Clearly, the unbounded buffer size is a "non-implementable" way of guaranteeing loss-lessness. When moving towards implementable designs, this assumption has to be removed. A buffer can be provided to store temporarily data that are exchanged among processes but it must be of finite size. The choice of the size of the buffer is crucial. Unfortunately deciding whether a finite buffer implementation exists that guarantees loss-lessness is not theoretically feasible in the general case, but there are cases for which the optimal buffer size can be found. In others, one has to hope for the best for buffer overwrite not to occur or has to provide additional mechanism that composed with the finite buffer implementation still guarantees that no loss takes place. For example, a send-receive protocol can be put in place to prevent buffer over-write to occur. Note that in this case the refinement process is quite complex and involves the use of composite processes. Today, there is little that is known about a general approach to communication design that has some of the feature that we have exposed, even though we have proposed a family of models that are related to each other as successive refinements [8].

Approaches to the isolation of communication and computation, and how to refine the communication specification towards an implementation [9] have been presented elsewhere. In some cases, we have been able to determine a synthesis procedure for the communication that guarantees some properties. In our opinion, this formalism and the successive refinement process opens a very appealing window to system design with unexplored avenues in component-based software design. It is our opinion that the latest advances in component-based software design and in software engineering are

converging, albeit slowly and probably unconsciously towards a more formal model of communication among modules.

## MICRO-ARCHITECTURE

In most design approaches, the next stage of the design process involves the evaluation of tradeoffs across what we refer to as the architecture/micro-architecture boundary, and at this point in our presentation, the class of structural compositions that implement the architecture is of primary concern. While the word architecture is used in many meanings and contexts, we adhere to the definitions put forward in [10]: the *architecture* defines an interface specification that describes the functionality of an implementation, while being independent of the actual implementation. The *micro-architecture*, on the other hand, defines how this functionality is actually realized as a composition of modules and components, along with their associated software.

The instruction-set architecture of a microprocessor is a good example of an architecture: it defines what functions the processor supports, without defining how these functions are actually realized. The micro-architecture of the processor is defined by the "organization" and the "hardware" of the processor. These terms can easily be extended to cover a much wider range of implementation options. At this point, the design decisions are made concerning what will eventually be implemented as software or as hardware.

Consistent with the above definitions, in our work we describe a micro-architecture as a set of *interconnected* components (either abstract or with a physical dimension) that is used to implement a function. For example, an LCD, a physical component of a micro-architecture, can be used to display the number of revolutions per minute of an automotive engine. In this case, the component has a concrete, physical representation. In other cases, it may have a more abstract representation. In general, a component is an element with specified interfaces and explicit context dependency. The micro-architecture determines the final hardware implementation and hence it is strictly related to the concept of platform [11,12] that will be presented in greater detail later on.

The most important micro-architecture for the majority of embedded designs consists of microprocessors, peripherals, dedicated logic blocks and memories. For some products, this micro-architecture is completely or in part fixed. In the case of automotive body electronics, the actual placement of the electronic components inside the body of the car and their interconnections is kept mostly fixed, while the single components, i.e., the processors, may vary to a certain extent. A fixed micro-architecture simplifies the design problem a great deal—especially the software part today—but limits design optimality. The trade-off is not easy to achieve.

In addition, the communication among micro-architecture blocks must be handled with great care. Its characteristics make the composition of blocks easy or difficult to achieve. Standards are useful to achieve component re-use. Busses are typical interconnection structures intended to favor re-use. Unfortunately, the specification of standard busses such as the PCI bus is hardly formal. This makes the design of the interfaces at best haphazard. Ultimately, we believe the verification of such interconnection interfaces will be the limiting factor in design productivity. In addition, we are experimenting with different interconnect structures such as on-chip networks.

## MAPPING

The essential design step that allows moving down the levels of the design flow is the mapping process, where the functions to be implemented are assigned (mapped) to the components of the micro-architecture. For example, the computations needed to display a set of signals may all be mapped to the same processor or to two different components of the micro-architecture (e.g., a microprocessor and a DSP). The mapping process determines the performance and the cost of the design. To measure exactly the performance of the design and its cost in terms of used resources, it is often necessary to complete the design, leading to a number of time-consuming design cycles. This is a motivation for using a more rigorous design methodology. When the mapping step is carried out, our choice is dictated by *estimates* of the performance of the implementation of that function (or part of it) onto the micro-architecture component. Estimates can be provided either by the manufacturers of the components (e.g., IC manufacturers) or by system designers. Designers use their experience and some analysis to develop estimation models that can be easily evaluated to allow for fast design exploration and yet are accurate enough to choose a good micro-architecture. Given the importance of this step in any application domain, automated tools and environments should support effectively the mapping of functions to micro-architectures.

The mapping process is best carried out interactively in the design environment. The output of the process is either:

- A mapped micro-architecture iteratively refined towards the final implementation with a set of constraints on each mapped component (derived from the top-level design constraints) or
- A set of diagnostics to the micro-architecture and function selection phase in case the estimation process signals that design constraints may not be met with the present micro-architecture and function set. In this case, if possible, an alternative micro-architecture is selected. Otherwise, we have to work in the function space by either reducing the number of

functions to be supported or their demands in terms of performance.

## LINK TO IMPLEMENTATION

This phase is entered once the mapped micro-architecture has been estimated as capable of meeting the design constraints. The next major issue to be tackled is implementing the components of the micro-architecture. This requires the development of an appropriate hardware block or of the software needed to make the programmable components perform the appropriate computations. This step brings the design to the final implementation stage. The hardware block may be found in an existing library or may need a special purpose implementation as dedicated logic. In this case, it may be further decomposed into sub-blocks until either we find what we need in a library or we decide to implement it by “custom” design. The software components may exist already in an appropriate library or may need further decomposition into a set of sub-components, thus exposing what we call the *fractal (self-similar) nature of design, i.e.*, the design problem repeats itself at every level of the design hierarchy into a sequence of nested function-(architecture)-micro-architecture-mapping processes.

## APPLICATIONS

This methodology is not empty rhetoric, but it has been already tested in advanced industrial environments. In particular, complex system designs have been cast in this framework and implemented in the field of automotive electronic subsystems, as detailed in three other papers of this conference, and consumer electronics [13]. Its basic aspects have been incorporated in POLIS [2], Ptolemy [14] and in an at least an industrial product [3, 4], VCC by Cadence, and we know of other tools that are being developed based on these concepts.

### Philips VideoTop

COSY is an EEC project on system design and verification involving industry (Cadence, Philips, Siemens) and academia (University of Paris Marie-Curie, Politecnico di Torino, University of Tübingen)[13]. The main goal was to apply the methodology outlined above (and modifying it whenever necessary) to industrial system design. The main driver was VideoTop, a subsystem that incorporates the main functionality of a digital video broadcast system.

The system receives an MPEG2 transport stream, where the user selects the channels to be decoded. The associated video streams are then unscrambled, demultiplexed, and decoded. The user may also define post-processing operations on the decoded streams, such as zooming and composition (picture-in-picture). The functional specifications required the following components:

- An MPEG2 (ISO/IEC 13818-2) de-multiplexer.
- An MPEG2 parser.
- An MPEG2 decoder (H.262 compliant up to main profile and high level).
- A video re-sizer with a zoom factor from 0.16 to 10. (Controlled by the user.)
- A video mixer for a number of arbitrary sized video images. (Positions controlled by the user.)
- A User interface

The functional specifications were captured using a formal model of computations (YAPI [15]) derived from Kahn's process networks. The model consisted of

- 62 processes
- 206 communication arcs
- 25,000 C/C++ lines<sup>2</sup>

According to our methodology, the functional specifications were analyzed and simulated executing the Y-API program. The simulation time on a Linux Pentium 450Mhz was about 80 times slower than real-time behavior. The micro-architecture selection started with the following basic blocks: a MIPS PR39K with 1 SRAM block and PSOS as RTOS. We tried a full software implementation and one with a number of functions mapped in hardware resulting in a system with 4 busses, 14 application-specific co-processors and 17 software tasks running on the microprocessor [16]. The all-software solution had an estimate running time using the methods described above that was 160 times slower than the mixed hardware-software solution. The most relevant hardware blocks were identified by performance analysis. Top of the list was the Frame-Buffer Memory-Manager (T-memory). Some additional mappings are under study now to see whether a different micro-architecture with fewer co-processors could be safely used. Rather surprisingly given the complexity of the system, the estimation results obtained with VCC were within 5% of a cycle accurate simulator developed at Philips, TSS, thus demonstrating that estimation is feasible and a real help in quick architectural evaluations. The modeling of the micro-architecture and of the functional part of the design kept the communication part of the design separate from the computation part [17]. This resulted in an accurate analysis of the communication performance of the system. A major design effort in Philips is to apply the principles of Platform-based design and rapid prototyping to their most important new generation products.

#### Magneti-Marelli Automotive Engine Control

This design [18] had many different features with respect to the previous one: it has a strong control component and tight safety constraints. In addition, the application had a

---

<sup>2</sup> The number of lines of code in a functional specification corresponds in general to MANY more lines of actual implementation code. This is a sizable example!

large part of legacy design. The functionality of the engine-control automotive electronic subsystem consists of the following components:

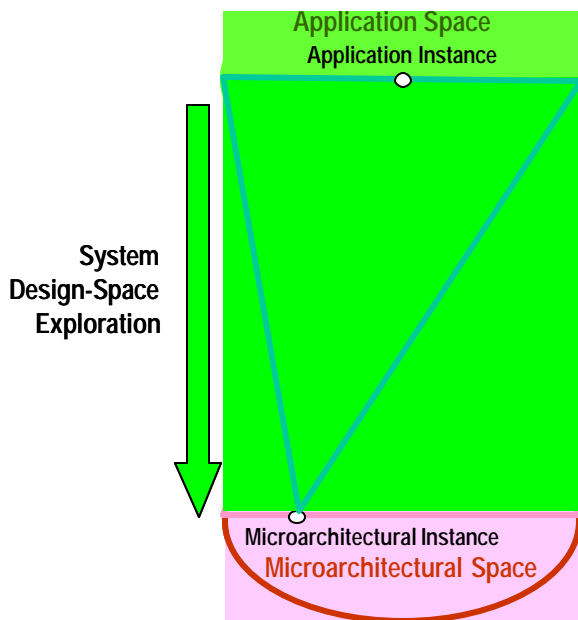
- Failure detection and recovery of input sensors
- Computation of engine phase, status and angle, crankshaft revolution speed and acceleration
- Injection and ignition control law
- Injection and ignition actuation drivers

The existing implementation had *135,000 line of source C code without comments*. The first task was to extract the precise functionality from the implementation. This was done by using a CFSM-based representation resulting in 89 CFSMs and 56 timers. The behavior of the actuators and of part of the sensors was completely re-written in the formal model. For the ignition and injection control law, we encapsulated the legacy C code into 18 CFSMs representing concurrent processes. The software was re-designed using the architecture described in the next section so that the mapping into different micro-architecture could be done with relative ease. In particular, we were able to test three different CPUs and, for each, two different software partitions to verify functionality and real-time behavior. In addition, we explored three different architectures for the I/O subsystem: one with a full software implementation, one with the use of a particular peripheral for timing functions (provided by the CPU vendor) and one with a highly optimized full hardware peripheral of new design.

The performance estimation was performed with VCC and resulted in an error with respect to a prototype board with real hardware of only 11%. The implementation of the functionality on the three platforms is under way. For two of them it is almost completed resulting in software re-usability of more than 86%. We also used the functionality captured in semi-formal terms to design a new dual processor architecture that is under design at ST Microelectronics.

## **PLATFORM-BASED DESIGN**

When mapping the functionality of the system to an integrated circuit, the economics of chip design and manufacturing are essential to determine the quality and the cost of the system. Since the mask set and design cost for Deep Sub-Micron implementations is predicted to be overwhelming, it is important to find common architectures that can support a variety of applications as well as the future evolutions of a given application. To reduce design costs, re-use is a must. In particular, since system designers will use more and more frequently software to implement their products, there is a need for design methodologies that allow the substantial re-use of software. This implies that the basic micro-architecture of the implementation is essentially "fixed", i.e., the principal components should remain the same within a certain



**Figure 2. Top-down and Bottom-up Approaches to Platform Specification**

degree of parameterization. For embedded systems, which we believe are going to be the dominant share of the electronics market, the “basic” micro-architecture consists of programmable cores, I/O subsystem and memories. A family of micro-architectures that allow substantial re-use of software is what we call a **hardware platform**. We believe that hardware platforms will take the lion’s share of the IC market. However, the concept of hardware platform by itself is not enough to achieve the level of application software re-use we are looking for. To be useful, the hardware platform has to be abstracted at a level where the application software sees a high-level interface to the hardware that we call **Application Program Interface or API**. There is a software layer that is used to perform this abstraction. This layer wraps the different parts of the hardware platform: the programmable cores and the memory subsystem via a Real-Time Operating System (RTOS), the I/O subsystem via the Device Drivers, and the network connection via the network communication subsystem. This layer is called the **software platform**. The combination of the hardware and the software platforms is called the **system platform**.

## HARDWARE PLATFORMS

Seen from the application domain, the constraints that determine the hardware platform are often given in terms of performance and “size”. To sustain a set of functions for a particular application, a CPU should be able to run at least at a given speed and the memory system should be of at least a given number of bytes. Since each product is characterized by a different set of functions, the constraints identify different hardware platforms where applications that are more complex yield stronger architectural constraints. Coming from the hardware space, production and design costs imply adding hardware platform constraints and consequently reducing

the number of choices. The intersection of the two sets of constraints defines the hardware platforms that can be used for the final product. Note that, as a result of this process, we may have a hardware platform instance that is over-designed for a given product, that is, some of the power of the micro-architecture is not used to implement the functionality of that product. Over-design is very common for the PC platform. In several applications, the over-designed micro-architecture has been a perfect vehicle to deliver new software products and extend the application space. We believe that some degree of over-design will be soon accepted in the embedded system community to improve design costs and time-to-market. Hence, the “design” of a hardware platform is the result of a trade-off in a complex space that includes:

- The size of the application space that can be supported by the micro-architectures belonging to the hardware platform. This represents the flexibility of the hardware platform;
- The size of the micro-architecture space that satisfies the constraints embodied in the hardware platform definition. This represents the degrees of freedom that micro-architecture providers have in designing their hardware platform instances.

Once a hardware platform has been selected, then the design process consists of exploring the remaining design space with the constraints set by the hardware platform. These constraints cannot only be on the components themselves but also on their communication mechanism. When we march towards implementation by selecting components that satisfy the architectural constraints defining a platform, we perform a successive refinement process where details are added in a disciplined way to produce a hardware platform instance.

Ideally, the design process in this framework starts with the determination of the set of constraints that defines the hardware platform for a given application. In the case of a particular product, we advocate to start the design process before splitting the market into high-end and low-end products. The platform thus identified can then be refined towards implementation by adding the missing information about components and communication schemes. If indeed we keep the platform unique at all levels, we may find that the cost for the low-end market is too high. At this point then, we may decide to introduce two platform instances differentiated in terms of peripherals, memory size and CPU power for the two market segments. On the other hand, by defining the necessary constraints in view of our approach, we may find that a platform exists that covers both the low-end and the high-end market with great design cost and time-to-market improvements.

Hardware platform-based design optimizes globally the various design parameters including, as a measure of optimality, NRE costs in both production and design. Hardware platform-based design is neither a top-down nor

a bottom-up design methodology. Rather, it is a “meet-in-the-middle” approach. In a pure top-down design process, application specification is the starting point for the design process. The sequence of design decisions drives the designer toward a solution that minimizes the cost of the micro-architecture. Figure 2 shows the single application approach, the bottom of the figure shows the set of micro-architectures that could implement that application. The design process selects the most attractive solution as defined by a cost function. In a bottom-up approach, a given micro-architecture (instance of the architectural space) is designed to support a set of different applications that are often vaguely defined and is in general much based on designer intuition and marketing inputs. In general, this is the approach taken by IC companies that try to maximize the number of applications (hence, the production volume) of their platforms.

### SOFTWARE PLATFORM

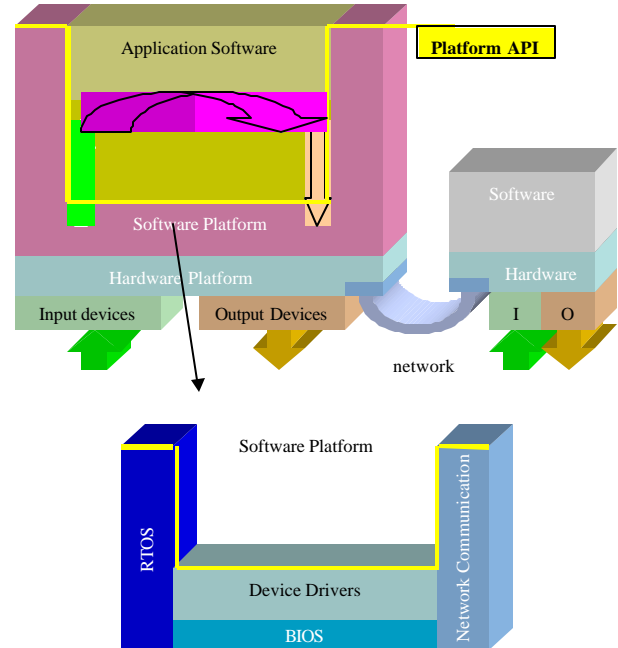
The concept of hardware platform by itself is not enough to achieve the level of application software re-use we are looking for. To be useful, the hardware platform has to be abstracted at a level where the application software “sees” a high-level interface to the hardware that we call Application Program Interface (API) or Programmers Model. There is a software layer that is used to perform this abstraction (Figure 3). This layer wraps the essential parts of the hardware platform:

- The programmable cores and the memory subsystem via a Real Time Operating System (RTOS),
- The I/O subsystem via the Device Drivers, and
- The network connection via the network communication subsystem<sup>3</sup>.

This layer is called the software platform. In our conceptual framework, the programming language is the abstraction of the ISA, while the API is the abstraction of a multiplicity of computational resources (concurrency model provided by the RTOS) and available peripherals (Device Drivers).<sup>4</sup> There are different efforts that try to standardize the API or Programmers Model. In our framework, the API or Programmers Model is a unique abstract representation of the hardware platform. With an API so defined, the application software can be re-used for every platform instance.

Of course, the higher the abstraction layer at which a platform is defined, the more instances it contains. For example, to share source code, we need to have the same operating system but not necessarily the same instruction set, while to share binary code, we need to

add the architectural constraints that force to use the same ISA, thus greatly restricting the range of architectural choices.



**Figure 3: Layered software structure**

In our framework, the RTOS is responsible for the scheduling of the available computing resources and of the communication between them and the memory subsystem. Note that in most of the embedded system applications, the available computing resources consist of a single microprocessor. However, in general, we can imagine a multiple core hardware platform where the RTOS schedules software processes across different computing engines.

There is a battle that is taking place in this domain to establish a standard RTOS for embedded applications. For example, traditional embedded software vendors such as ISI and WindRiver are now competing with Microsoft that is trying to enter this domain by offering Windows CE, a stripped down version of the API of its Windows operating system. In our opinion, if the conceptual framework we offer here is accepted, the precise definition of the hardware platform and of the API should allow to synthesize automatically and in an optimal way most of the software layer, a radical departure from the standard models borrowed from the PC world. Software re-use, i.e. *platform re-targetability*, can be extended to these layers (middle-ware) hopefully resulting more effective than binary compatibility.

<sup>3</sup> In some cases, the entire software layer, including the Device Drivers and the network communication subsystem is called RTOS.

<sup>4</sup> There are several languages that abstract or embed the concurrency model directly, avoiding the RTOS abstraction.

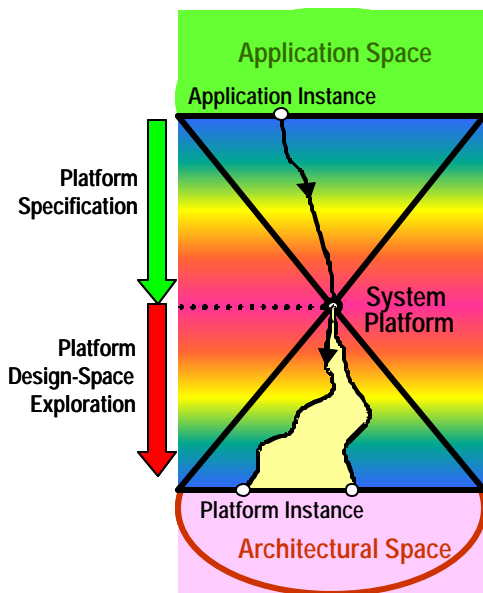


Figure 4. Platform Abstraction and Design Flow

## SYSTEM PLATFORMS

In summary, the development of programmable solution involves understanding the application domain, developing an architecture and micro-architecture that is specialized to that application domain, as well as the development of software development tools to program the specialized architecture.

One of the major challenges in this regard is developing an understanding what it means to program a complex system efficiently. The central question to be addressed here is: "What is the programmers' model?" Or "How should the programmer view the underlying hardware and input/output systems?" On the one hand, we want to hide as many of the details of the underlying implementation as possible, while on the other we want to make visible a sufficient level of control that the application programmer can develop an efficient solution—in terms of performance, power, and reliable functionality.

The basic ideas of system platform and platform-based design are captured in Figure . The vertex of the two cones represents the API or Programmers' Model, i.e., the abstraction of the hardware platform. A system designer maps its application into the abstract representation that "includes" a family of micro-architectures that can be chosen to optimize cost, efficiency, energy consumption and flexibility. The mapping of the application into the actual architecture in the family specified by the Programmers' Model or API can be carried out, at least in part, automatically if a set of appropriate software tools (e.g., software synthesis, RTOS synthesis, device-driver synthesis) is available. It is

clear that the synthesis tools have to be aware of the architecture features as well as of the API.

In the design space, there is an obvious trade-off between the level of abstraction of the Programmers' Model and the number and diversity of the platform instances covered. The more abstract the Programmers' Model the richer is the set of platform instances but the more difficult it is to choose the "optimal" platform instance and map automatically into it. Hence, we envision a number of system platforms that will be handled with somewhat different abstractions and tools. For example, more traditional platforms that include a small number of standard components such as microprocessors and DSPs will have an API that is simpler to handle than reconfigurable architectures. In the next section, we will show examples of application of the concepts exposed so far. One is related to the design of next generation wireless systems, the other on a class of reconfigurable architectures, their Programmers' Model and tools to map applications onto an architecture.

## CONCLUSION

We have presented some considerations about the future of electronic devices and infrastructures as they affect the world of car manufacturing and design. In particular, I stressed the issues related to the choice and design of integrated platforms. I argued that a rigorous design methodology based on separation of concerns is essential. I presented a novel concept of hardware, software and system platforms that could be used to reduce substantially design time and cost. The challenges ahead of us are great, but the opportunities are enormous. We are on the edge of a revolution in the way a car is conceived and designed.

## ACKNOWLEDGMENTS

Several people were instrumental in building the vision presented in this paper: the initial inspiration came from Dr. Pecchini, General Manager of the Power-train Division of Magneti-Marelli; the first environment built on these ideas was developed in close collaboration with Prof. Luciano Lavagno, University of Udine and Cadence Laboratories; the mathematical formalism came from the joint work with Max Chiodo of Cadence Design Systems, Inc. and Luciano Lavagno; the VCC architecture was built with Dr. Jim Rowson. Dr. Alberto Ferrari shared his work on platform-based design and was the main actor in the application of the methodology to the engine control problem. Dr. Wolfgang Reitzle formerly with BMW and now with Ford was a very strong supporter of the work presented here. Dr. Patrick Popp of BMW Technology Center in Palo Alto provided resources for the project and shared its vision. Giovanni Gaviani of Magneti-Marelli Power-train Division gave his inputs and important views into the future of engines and cars. This research has been supported by the Giga-scale Silicon Research

1. M. Pinto, CTO Lucent Microelectronics, private communication, June 1999.
2. F. Balarin et al., *Hardware-Software Co-Design of Embedded Systems: The POLIS Approach*, Kluwer Publishing Co., 1998.
3. J. Rowson and A. Sangiovanni-Vincentelli, *System Level Design*, *EE Times*, 1996.
4. J. Rowson and A. Sangiovanni-Vincentelli, *Interface-based Design*, *Proceedings of the 34th Design Automation Conference (DAC-97)*. pp. 178-183, Las Vegas, June 1997.
5. C. Szyperski, *Component Software: Beyond Object-Oriented Software*, ACM Press, Addison-Wesley 1998
6. A. Sangiovanni-Vincentelli, R. McGeer and A. Saldanha, *Verification of Integrated Circuits and Systems*, *Proc. Of 1996 Design Automation Conference*, June 1996.
7. E. Lee and A. Sangiovanni-Vincentelli, *A Unified Framework for Comparing Models of Computation*, *IEEE Trans. on Computer Aided Design of Integrated Circuits and Systems*, Vol. 17, N. 12, pp. 1217-1229, December 1998.
8. M.Sgroi, L.Lavagno, A.Sangiovanni-Vincentelli, *Formal Models for Embedded System Design*, *To appear in IEEE Design & Test of Computers. Special Issue on System Design*, 2000.
9. J.L. da Silva Jr., M. Sgroi, F. De Bernardinis, S.F. Li, A. Sangiovanni-Vincentelli, J. Rabaey *Wireless Protocols Design: Challenges and Opportunities*, *8th International Workshop on Hardware/Software Co-Design Codes/CASHE '00*, Diego, CA May 2000.
10. C. G. Bell and A. Newell, "Computer Structures: Readings and Examples," McGraw-Hill, New York, 1971.
11. A. Ferrari and A. Sangiovanni-Vincentelli, *System Design: Traditional Concepts and New Paradigms*, *Proceedings of the 1999 Int. Conf. On Comp. Des.*, Austin, Oct. 1999.
12. H. Chang et al., *Surviving the SOC Revolution: Platform-based Design*, Kluwer Academic Publishing, 1999.
13. J-Y. Brunel, A. Sangiovanni-Vincentelli and Rainer Kress, «COSY: a methodology for system design based on reusable hardware & software IP's,» *in: J-Y. Roger (ed.), Technologies for the Information Society*, IOS Press, pp. 709-716, June 1998
14. J. Davis II, M. Goel, C. Hylands, B. Kienhuis, E. A. Lee, Jie Liu, X. Liu, L. Muliadi, S. Neuendorffer, J. Reekie, N. Smyth, J. Tsay and Y. Xiong, "An Overview of the Ptolemy Project", ERL Technical Report UCB/ERL No. M99/37, Dept. EECS, University of California, Berkeley, CA 94720, July 1999.
15. E.A. de Kock, G. Essink, W. Smits, van der Wolf, JY. Brunel, W. Kruijtzter, P. Lieveise and K. Vissers, «YAPI: Application Modeling for Signal Processing Systems,» *DAC'2000*, Los Angeles, June 2000
16. H. Kenter, C. Passerone, W. Smits, Y. Watanabe and A. Sangiovanni-Vincentelli, "Designing Digital Video Systems: Modeling and Scheduling," *CODES'99*, Rome, May 1999
17. J-Y. Brunel, W. Kruijtzter, H. Kenter, F. Pétrot, L. Pasquier, E. de Kock and W. Smits, «COSY Communication IP's,» *DAC'2000*, Los Angeles, June 2000
18. M. Baleani, A. Ferrari, A. Sangiovanni-Vincentelli and C. Turchetti, "Hardware-Software Co-design of an Engine Management System", *Proc. of DATE 2K*, Paris, March 2000.