INVITED PAPER

Quo Vadis, SLD? Reasoning About the Trends and Challenges of System Level Design

Recognizing common requirements for co-design of hardware and software in diverse systems may lead to productivity gains, lower costs and first-pass design success.

By Alberto Sangiovanni-Vincentelli, *Fellow IEEE*

ABSTRACT | System-level design (SLD) is considered by many as the next frontier in electronic design automation (EDA). SLD means many things to different people since there is no wide agreement on a definition of the term. Academia, designers, and EDA experts have taken different avenues to attack the problem, for the most part springing from the basis of traditional EDA and trying to raise the level of abstraction at which integrated circuit designs are captured, analyzed, and synthesized from. However, my opinion is that this is just the tip of the iceberg of a much bigger problem that is common to all *system* industry. In particular, I believe that notwithstanding the obvious differences in the vertical industrial segments (for example, consumer, automotive, computing, and communication), there is a common underlying basis that can be explored. This basis may yield a novel EDA industry and even a novel engineering field that could bring substantial productivity gains not only to the semiconductor industry but to all system industries including industrial and automotive, communication and computing, avionics and building automation, space and agriculture, and health and security, in short, a real technical renaissance.

In this paper, I present the challenges faced by industry in system level design. Then, I propose a design methodology, platform-based design (PBD), that has the potential of addressing these challenges in a unified way. Further, I place methodology and tools available today in the PBD framework and present a tool environment, Metropolis, that supports PBD and that can be used to integrate available tools and methods together with two examples of its application to separate industrial domains.

KEYWORDS | Embedded software; embedded systems; models and tools; platform-based design (PBD); system-level design (SLD); system-level design environments

I. INTRODUCTION

Electronic design automation (EDA) has played a pivotal role in the past 25 years in making it possible to develop a new generation of electronic systems and circuits. However, innovation in design tools has slowed down significantly as we approach a limit in the complexity of systems we can design today satisfying increasing constraints on time-to-market and correctness. The EDA community has not succeeded as of today in establishing a new layer of abstraction universally agreed upon that could provide productivity gains similar to the ones of the traditional design flow (Register Transfer Level (RTL) to GDSII) when it was first introduced. Nor has it been able to expand significantly into new adjacent markets to increase its total available market. Among the adjacencies of interest, I believe the *electronics system market* has great potential since system companies that are now facing significant

Manuscript received May 13, 2006; revised December 7, 2006. This work was supported in part by the Gigascale System Research Center, the Center for Hybrid and Embedded Software Systems (CHESS) at the University of California, Berkeley, which receives support from the National Science Foundation (NSF award CCR-0225610), the State of California Micro Program, and the following companies: Agilent, Bosch, DGIST, General Motors, Hewlett Packard, Infineon, Microsoft, National Instruments, and Toyota. This work was also supported by the European Project SPEEDS and the Networks of Excellence Artist2 and Hycon.
The author is with the Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA 94720 USA (e-mail: alberto@eecs.berkeley.edu).

Digital Object Identifier: 10.1109/JPROC.2006.890107

difficulties due to an exponentially raising complexity and to increased demands on functionality, correctness, and time-to-market are contemplating a radical change in the way they bring new products to market.

In this paper, I will discuss the following two aspects.

1) *Raising the level of abstraction when designing chips*. In this framework, the term system-level design for an integrated circuit relates to any level of abstraction that is "above" RTL. Transaction level modeling (TLM), behavioral, algorithmic, and functional modeling are terms often used to indicate higher levels of abstraction in hardware design. The International Technology Roadmap for Semiconductors (ITRS) in 2004 places SLD "a level above RTL including both HW and SW design." SLD is defined to "consist of a behavioral (before HW/SW partitioning) and architectural level (after)" and is claimed to increase productivity by 200K gates/designer-year. The ITRS states that SLD will produce an estimated 60% productivity improvement over what they call the "intelligent testbench" approach (the previously proposed electronic system design improvement) [178]. While these claims cannot be verified as yet and do look quite aggressive, most agree that the overarching benefits that ESL should bring to the table are to:
   a) raise the level of abstraction at which designers express systems;
   b) enable new levels of design reuse.
2) *Dealing with electronics system design and, in particular, embedded system design*. In the context of this paper, the term embedded systems refers to the electronic components (which almost always include one or more software programmable parts) of a wide variety of personal or broad-use devices, e.g., a mechanical system such as an automobile, a train, a plane, an electrical system such as an electrical motor or generator, a chemical system such as a distillation plant, or health-care equipment such as a pacemaker. Hence, an embedded system is a special-purpose system in which the computing element is completely encapsulated by the device it controls. Unlike a general-purpose computer, an embedded system performs one or a few predefined tasks, usually with very specific requirements [204]. In technical terms, an embedded system interacts with the surrounding environment in a controlled way satisfying a set of requirements on responsiveness in terms of quality and timeliness. Typically, it has to satisfy implementation requirements such as cost, power consumed, and use of limited physical resources. Ideally, its interaction with the environment should be continuously available for the entire life of the artifact.

To deal with system-level problems, let them be at the chip or embedded system level, *the issue to address is not developing new tools, albeit they are essential to advance the state of the art in design, rather, it is the understanding of the principles of system design, the necessary change to design methodologies, and the dynamics of the supply chain*. Developing this understanding is necessary to define a sound approach to the needs of the system and IC companies as they try to serve their customers better, to develop their products faster and with higher quality. It is no wonder that EDA experts have stayed away from system-level design; in fact, EDA experts are essentially tied to the semiconductor industry needs in the *implementation flow* with little or no expertise in the intricacies of embedded systems that include a large amount of software and system integration concerns. The motivation for EDA experts to learn system design has not been there as yet since:

1) IC companies are still struggling with the understanding of higher levels of abstraction;
2) system companies have not perceived as yet design methodology or tools to be on their critical path, hence they have not been willing to invest in "expensive" tools.

Clearly, as we are hitting a wall in the development of the next generation systems, this situation is rapidly changing. Major productivity gains are needed and better verification and validation is a necessity as the safety and reliability requirements of embedded systems become more stringent and the complexity of chips is hitting an all-time high. Several approaches have emerged in the design community to improve the situation and some EDA companies have invested in the area but a broad EDA industry support for these approaches is still missing.

I share with a number of colleagues [111], [112], [131], [135], [176], [188], [208] (this list also provides an excellent set of references for the state-of-the-art and directions for embedded system design) the strong belief that a new *design science* must be developed to address the challenges listed above where the physical is married to the abstract, where the world of analog signals is coupled with the one of digital processors, and where ubiquitous sensing and actuation make our entire environment safer and more responsive to our needs. SLD should be based on the new design science to address our needs in a fundamental way. However, the present directions are not completely clear as the new paradigm has not yet fully emerged in the design community with the strength necessary to change the EDA and design technology landscape, albeit the papers quoted in this paragraph have chartered the field with increasing clarity.

Support for the development of this design science is given in the U.S. by the traditional research funding organizations in collaboration with industrial associations. The Gigascale System Research Center (GSRC) [101] of the MARCO program [a joint initiative of the Defense

Advanced Research Project Agency (DARPA) and the Semiconductor Industry Association (SIA)] and the National Science Foundation (NSF) with the Center for Hybrid and Embedded Software Systems (CHESS) [112] are two examples of this effort. However, a much stronger effort is afoot in Europe, where the European community has been supporting embedded system research and novel methodologies for chip design for years with large integrated projects (e.g., SPEEDS) and networks of excellence (e.g., Artist 2 [2] and HYCON [119]) and is planning an increased effort for the VII Framework. In addition, a technology platform, Artemis [51], was formed three years ago by the leading European industrial companies (the initial founding group included Nokia, Ericsson, ST, ABB, Airbus, Infineon, British Telecom, Siemens, Bosch, Contiteves, Daimler-Chrysler, Thales, FIAT, Finmeccanica, Philips, COMAU, Symbian, Telenor, and PARADES with the support of research organizations such as IMEC, Verimag, and the Technical University of Vienna, a healthy combination of Academia, service providers, software companies, system, subsystem and semiconductor manufacturers). The companies have recently formed the Artemis Industrial Association (ARTEMISIA), while the European community is deciding to make it a joint technology initiative, an instrument to funnel substantial resources of the member states. In the last meeting of the European Community Prime Ministers on October 2006, Artemis was quoted by some of the participants (in particular, the Finnish Prime Minister) as an example of agenda setting initiative for the industrial future of Europe.

This paper is not intended to review exhaustively the various approaches and tools that have been proposed over the years (a reader interested in tools and environments available today is referred to [63] for a taxonomy), albeit I will review the most relevant contributions in the perspective presented here.

The paper is organized as follows: in Section II, I focus on the pressing concerns of system level design together with the strategic and business concerns in the supply chains of the mobile terminal and automotive vertical domains as examples of the issues to be taken into consideration when we think about expanding the reach of design methodology and tools. I then present my view on how to form a unified approach to embedded system design, platform-based design (PBD), that could provide a solution to the challenges presented in the previous sections (Section III). In Section IV, I describe some of the most promising approaches to embedded system design using the PBD context as a guide. Then, I present a brief overview of a system design framework that supports this methodology (Section V) and that could form the basis for a unified environment for system level design and integration. In this section, I also present two test cases of the application of the methodology and of the framework from widely different industrial domains: the design of a JPEG encoder on a heterogeneous single chip computing platform and the design of a distributed architecture for supervisory control in automotive. In Section VI, I draw conclusions and indicate future directions for research and industrial developments.

Notably missing from this paper is testing. The topic is extremely important for SLD but to do justice to it, an entire new paper would be needed.

II. SETTING THE STAGE: CHALLENGES OF SYSTEM LEVEL DESIGN

In the present technology environment and industrial structure, SLD has to address concerns of individual players in the industrial domain that are facing serious problems in bringing their products to market in time and with the required functionality. I do believe that SLD also needs to be concerned about the entire industrial supply chain that spans from customer-facing companies to subsystem and component suppliers, since the health of an industrial sector depends on the smooth interaction among the players of the chain as if they were part of the same company. In this section, I present a view on both challenges that underline commonalities that allow a unified approach to SLD.

A. Managing Complexity and Integration

The ability to integrate an exponentially rising number of transistors within a chip, the ever-expanding use of electronic embedded systems to control increasingly many aspects of the "real world," and the trend to interconnect more and more such systems (often from different manufacturers) into a global network are creating a challenging scenario for embedded system designers. Complexity and scope are exploding into the three interrelated but independently growing directions, while teams are even shrinking in size to further reduce costs. In this scenario, the three challenges that are taking center stage are as follows.

1) Heterogeneity and Complexity of the Hardware Platform: The trends mentioned above result in exponential complexity growth of the features that can be implemented in hardware. The integration capabilities make it possible to build a real complex system on a chip including analog and RF components, general purpose processors (GPP) and Application-Specific Instruction-set Processors (ASIP). The decision of what goes on a chip is no longer dictated by the amount of circuitry that can be placed there, but by reliability, yield, power consumption, performance, and ultimately cost (it is well known that analog and RF components force the use of more conservative manufacturing lines with more processing steps than pure digital ICs). Even if manufacturing concerns suggest to implement hardware in separate chips, the resulting package may still be very small given

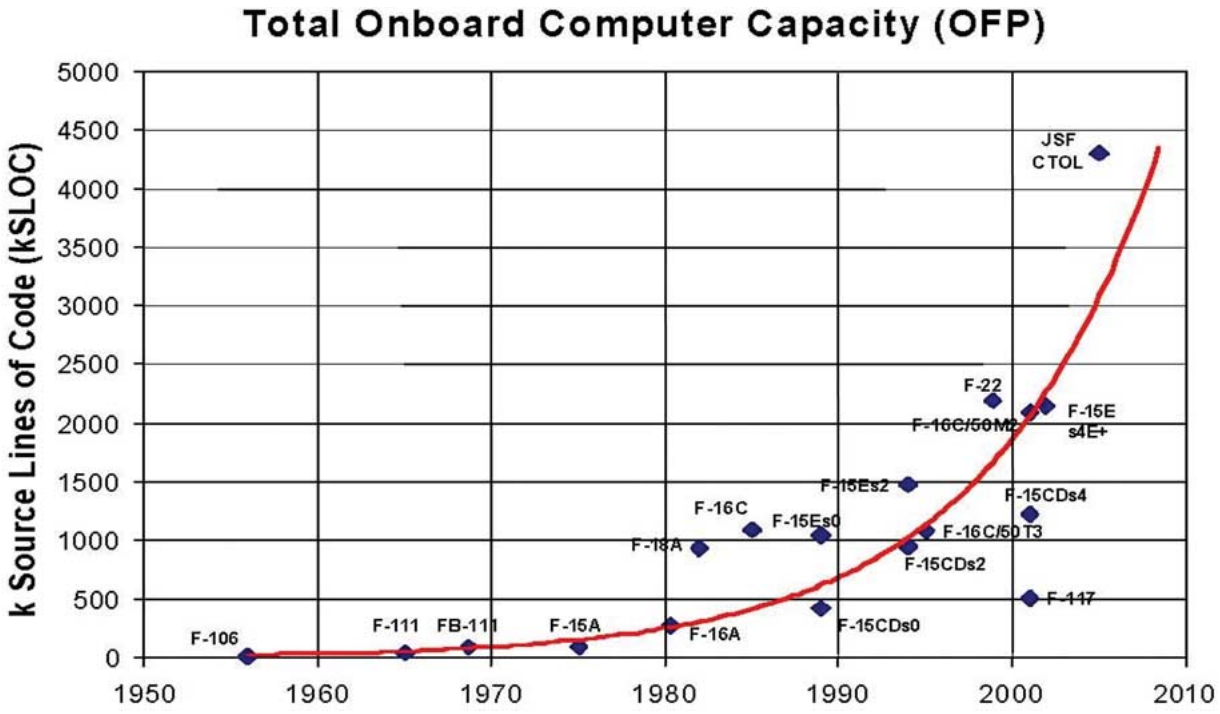


Fig. 1. ***Software growth in avionics.***

the advances in packaging technology yielding the concept of system-in-package (SiP). Pure digital chips are also featuring an increasing number of components. Design time, cost, and manufacturing unpredictability for deep submicron technology make the use of custom hardware implementations appealing only for products that are addressing a very large market and for experienced and financially rich companies. Even for these companies, the present design methodologies are not yielding the necessary productivity forcing them to increase beyond reason the size of design and verification teams. These IC companies (for example Intel, Freescale, ST, and TI) are looking increasingly to system design methods to allow them to assemble large chips out of predesigned components and to reduce validation costs (design reuse). In this context, the adoption of design models above RTL and of communication mechanism among components with guaranteed properties and standard interfaces is only a matter of time.

2) Embedded Software Complexity: Given the cost and risks associated to developing hardware solutions, an increasing number of companies is selecting hardware platforms that can be customized by reconfiguration and/or by software programmability. In particular, software is taking the lion's share of the implementation budgets and cost. In cell phones, more than 1 million lines of code is standard today, while in automobiles the estimated number of lines by 2010 is in the order of hundreds of millions [199]. The number of lines of source code of embedded software required for defense avionics systems is also growing exponentially as reported in Fig. 1 [made available by Robert Gold Associate Director, Software and Embedded Systems, Office of the Deputy Under Secretary of Defense (Science and Technology)]. However, as this happens, the complexity explosion of the software component causes serious concerns for the final quality of the products and the productivity of the engineering teams. In transportation, the productivity of embedded software writers using the traditional methods of software development ranges in the few tens of lines per day. The reasons for such a low productivity are in the time needed for verification of the system and long redesign cycles that come from the need for developing full system prototypes for the lack of appropriate virtual engineering methods and tools for embedded software. Embedded software is substantially different from traditional software for commercial and corporate applications. By virtue of being embedded in a surrounding system, the software must be able to continuously react to stimuli in the desired way, i.e., within bounds on timing, power consumed, and cost. Verifying the correctness of the system requires that the model of the software be transformed to include information that involves physical quantities to retain only what is relevant to the task at hand. In traditional software systems, the abstraction process leaves out *all* the physical aspects of the systems as only the functional aspects of the code matter.

Given the impact that embedded software has on the safety of embedded system devices and on the quality of the final artifact, there is an increasingly strong interest in having high assurance that embedded software is correct. *Software certification* demonstrates the reliability and safety of software systems in such a way that it can be checked by an independent authority with minimal trust in the techniques and tools used in the certification process itself. It builds on existing software assurance, validation, and verification techniques but introduces the notion of explicit software certificates, which contain all the information necessary for an independent assessment of the properties. Software certification has been required by military applications for years and has been recently extended to the U.S. aviation sector. The FAA accepted the DO-178B regulations as the means of certifying all new aviation software. A joint committee with the European authorities has been recently empowered to "promote safe implementation of aeronautical software, to provide clear and consistent ties with the systems and safety processes, to address emerging software trends and technologies, and to implement an approach that can change with the technology" [69], [70]. I believe that certification will expand into new safety-critical domains and will create an additional, serious burden on the embedded software design process not only for the aviation industry but for an increasingly large number of companies worldwide. Note that today, the main scope of the certification process relates to the process followed to develop software. I believe it will be of extreme importance to link the certification process with the *content* of the software and not only with the development process. This approach will have to include formal verification techniques as I believe this is the only way to increase the confidence in the correctness of the software.

3) Integration Complexity: A standard technique to deal with complexity is decomposing "top-down" the system into subsystems. This approach, which has been customarily adopted by the semiconductor industry for years, has a limitation as a designer or a group of designers has to fully comprehend the entire system and to partition appropriately its various parts, a difficult task given the enormous complexity of today's systems. Hence, the future is one of developing systems by *composing* pieces that all or in part have already been predesigned or designed independently by other design groups or even companies. This has been done routinely in vertical design chains for example in the avionics and automotive verticals, albeit in a heuristic and *ad hoc* way. The resulting lack of an overall understanding of the interplay of the subsystems and of the difficulties encountered in integrating very complex parts causes system integration to become a nightmare in the system industry. For example, Jurgen Hubbert, then in charge of the Mercedes-Benz passenger car division, publicly stated in 2003: "*The industry is fighting to solve problems that are coming from electronics. Companies that introduce new technologies face additional risks. We have experienced blackouts on our cockpit management and navigation command system and there have been problems with telephone connections and seat heating.*"

I believe that in today's environment this state is the rule rather than the exception for the leading system original equipment manufacturers (OEMs)[1] in all industrial sectors [51]. The source of these problems is clearly the increased complexity but also the difficulty of the OEMs in managing the integration and maintenance process with subsystems that come from different suppliers who use different design methods, different software architecture, different hardware platforms, and different (and often proprietary) real-time operating systems. Therefore, there is a need for standards in the software and hardware domains that will allow plug-and-play of subsystems and their implementation. The ability to integrate subsystems will then become a commodity item, available to all OEMs. The competitive advantage of an OEM will increasingly reside on novel and compelling functionalities.

There is also the need for improving the interaction among all the players in the supply chain to improve the integration challenges in a substantial way as I argue in the next section.

B. Industrial Supply Chain Landscape

The design and supply chains are the backbone for any industrial sector. Their health and efficiency are essential for economic viability. While tools for supply chain management have been around for quite some time, support for the design chain has not been pursued nearly as vigorously. There are great opportunities for improving the situation substantially at least in the safety-driven industrial sector, which includes the transportation as well as industrial automation domain, with a combination of tools and methodologies. We are just at the beginning.

Integration of electronic and mechanical design tools and frameworks will be essential in the near future. Integration of chemical, electronic, and biology tools will also be essential in the further future for nanosystems. Data integration and information flow among the companies forming the chain have to be supported. In other words, it is essential that the fundamental steps of system design (functional partitioning, allocation on computational resources, integration, and verification) be supported across the entire design development cycle. Thus, whether the integrations pertains to SW-SW integration on a distributed network, HW-SW integration on a single electronic control unit (ECU), or electronics and mechanical integration for a subsystem, tools and models have to

[1]In this paper, OEM is used to refer to the companies that acquire a product or component and reuse or incorporate it into a new product with their own brand names. Examples are Mercedes, GM, and Toyota, as well as Boeing and Airbus.

be integrated seamlessly from a static point of view (e.g., data dictionaries and off-line model transformations) and dynamic point of view (e.g., cosimulation, HW-in-the-loop simulations and emulation).

Assuming the design methodology and the infrastructure for design chain integration are all in place, what will be the implication on the industrial structure?

The dynamics in the system industry is *similar across the vertical domains but certainly there are important differences*. For example, for embedded controllers in industrial engineering applications, automotive, avionics, energy production, and health related equipment, safety considerations, and hence hard real-time constraints, are of paramount importance. In the case of consumer electronics, including entertainment subsystems in cars and airplanes, cell phones, cameras, and games, the concerns are on sound, video, and recording quality and on the look and feel of the devices in presence of severe cost constraints. I will briefly discuss the cell phone design chain and the automotive design chain as the representatives of the embedded system market dynamics.

1) Mobile Communication Design Chain: The cell phone industrial segment is a complex ecosystem in continuous evolution with the following actors.

a) ***Application developers*** such as gaming, ring tones, and video. These companies sell their products directly to the end customer except in cases where these applications come bundled either with standard services like voice offered by service providers such as Cingular, Verizon, or Telecom Italia, or with the device itself offered by makers such as Nokia, Motorola, Samsung, or Ericsson. Their designs are, in general, implemented in software running on the platform provided by the device manufacturers who choose also the OS.

b) ***Service providers*** who offer the access to the network infrastructure for voice and data. These providers also offer other services such as news, weather information, and traffic. The GSM standard introduced (and mandated) the use of the subscriber identity module (SIM), a smart card that securely stores the key identifying a mobile phone service subscriber, as well as subscription information, saved telephone numbers, preferences, text messages, and other information. The use of the SIM card is important in the dynamics of the vertical segment as it is under control of the service providers. The service provider technology relates to the management of the infrastructure and of the service delivery. They worry, for example, about communication handoffs when cell boundaries are traversed and base-station location.

c) ***Device makers*** who manufacture the mobile terminal, e.g., the cell phone. Device makers must master a number of different technologies as they manufacture systems with significant software content (more than 1 million lines of code today) and hardware content including computing and communication circuitry involving analog and RF. In most cases, the IC content is obtained by chip manufacturers such as Qualcomm, TI, Freescale, and ST, but it may also be designed by captive teams. One of the many challenges of a mobile terminal manufacturer is integrating heterogeneous semiconductors manufactured by different companies (for example, DSPs and microcontrollers for the digital part, base-band, and RF circuitry) whose interaction must be accurately predicted and controlled to provide the functionality with no errors. There is a significant IP content acquired by middleware software providers such as the Symbian OS, an operating system designed for mobile devices, with associated libraries, user interface frameworks, and reference implementations of common tools, produced by Symbian Ltd. In addition, styling, ergonomics, and user-friendliness are major attractions for the end customer.

d) ***IC providers*** who offer semiconductors and other IPs that implement a variety of a mobile terminal functions. Semiconductor technology has had a major impact in the diffusion of mobile terminals as it is responsible for the dimension, power consumption, performance, functionality, and cost of the terminal. Because of the complexity of the design and of the need of interfacing with other vendors, IC manufacturers have turned to a particular design style that is the major content of this paper, platform-based design. The TI OMAP [55] platform together with the Nexperia Philips platform for digital video are the first examples of complex semiconductors designed in this style. Given the sale volumes of mobile terminals, IC manufacturers are competing fiercely and to provide the features needed by the device manufacturers, they had to enter into system-level design and into the development of significant software components including device drivers and other middleware. The semiconductor manufacturers are themselves integrating third party IPs.

e) ***IP providers*** who provide components to the rest of the design chain. Symbian (with its OS for cell phones), Microsoft (with Windows CE), and ARM (with its processors) are examples of IP providers. These components are integrated in the semiconductors or in the terminal to perform an important function. They are instrumental to the functioning of the devices but cannot be sold to the end customer *per se*.

f) ***Outsourcing companies*** who provide design and manufacturing services to the rest of the chain. For example, Flextronics provides manufacturing services to a large variety of companies in the system domain including mobile terminal manufacturers. E-silicon [73] in U.S., Accent [6] in Europe and Faraday [87] in Taiwan offer design services to semiconductor and device manufacturers for part or entire chips as well as brokerage services to manage the interactions with silicon foundries. Finally, semiconductor foundries such as TSMC [200], IBM [153] and UMC [201] provide IC manufacturing services.

Today, there is a great deal of competition and turf battles to determine where the added value is inserted. For example, the boundary between service providers and device makers as well as the one between device and IC makers is under stress. Service providers favored the SIM card as a way of capturing value in their products and defend it against the device makers. The standard that limits the communication bandwidth between SIM cards and the cell phone electronics defends the device makers turf against the intrusion of the service providers. The device makers defend their added value against IC manufacturers by avoiding being locked into a single provider situation, farming out different components to different companies. In addition, they force, whenever possible, the IC providers to use standards that favor the possibility of using different IPs as they see fit. The use of the Open Core Protocol [157] standard in the TI OMAP [55] platform is a case where the interest of the device makers and the one of the IC provider aligned since it was also the interest of the IC provider to be able to incorporate quickly external and internal IPs. My opinion is that providing a unified methodology and framework, we will favor the balance of the chain where everyone reaches an equilibrium point that maximizes the welfare of the system.

2) Automotive Design Chain: The need for integrating widely different subsystems such as safety, propulsion, communication, and entertainment makes this vertical very interesting for our purposes. Today, the roles of car makers and their suppliers are relatively stable but they are undergoing a period of stress due to the increased importance of electronics and its added value. The Automotive supply chain includes:

a) ***Car manufacturers (OEMs)*** such as GM, Ford, Daimler-Chrysler, and Toyota, who provide the final product to the consumer market.

b) ***Tier 1 suppliers*** such as Bosch, Contiteves, Siemens, Nippon Denso, Delphi, and Magneti-Marelli, who provide subsystems such as power-train management, suspension control, and brake-by-wire devices to OEMs.

c) ***Tier 2 suppliers***, e.g., chip manufacturers such as Freescale, Infineon, ST, and Renesas, IP providers e.g., ARM and RTOS suppliers such as WindRiver and ETAS, who serve OEMs and more likely Tier 1 suppliers.

d) ***Manufacturing suppliers*** such as Flextronics and TSMC who provide manufacturing services. Opposite to verticals that are not safety critical, liability issues make the recourse to outside manufacturing not as common. However, there are signs that manufacturing for Tier 1 suppliers is increasingly considered for outsourcing.

Car makers express the desire of gaining a stronger grip on the integration process and on the critical parts of the electronics subsystems. At the same time, there is evidence that sharing IPs among car makers and Tier 1 suppliers could improve substantially time-to-market, development, and maintenance costs. The essential technical problem to solve for this vision is the establishment of standards for interoperability among IPs and tools. AUTOSAR [107], a world-wide consortium of almost all players in the automotive domain electronics supply chain, has this goal very clear in mind. However, there are technical and business challenges to overcome. In particular, from the technical point of view, while sharing algorithms and functional designs seems feasible at this time, the sharing of *hard real-time* software is difficult even assuming substantial improvements in design methods and technology, if run-time efficiency has to be retained. The issues are related to the interplay that different tasks can have at the RTOS level. The timing of the software tasks depend on the presence or absence of other tasks. A scheduling policy that could prevent timing variability in the presence of dynamical changing task characteristics can be conceived (for example, timing isolation or resource reservation policies) but it will carry overhead, albeit potentially not prohibitive; further, this kind of policy is not supported by any of the commercially available RTOS. This situation is the standard tradeoff between efficiency and reliability but it has more important business implications than usual. In fact, if software from different sources has to be integrated on a common hardware platform, in the absence of composition rules and formal verification of the properties of composed systems, who will be responsible for the correct functioning of the final product?

Whoever will take on this responsibility would need a very strong methodology and an iron fist to make suppliers and partners comply with it. This may not be enough, in the sense that software characteristics are hard to pin down and with the best intentions of this world, one may not be able to guarantee functional *and* timing behavior in the presence of foreign components. The constant growth of complexity of the embedded systems designed today makes manual analysis and design impractical and error prone. The ideal approach would be a tool that could map automatically the set of tasks onto the platform guaranteeing the correct functionality and timing with optimal resource utilization [160]. This

tool should take the design description at the pure functional level with performance and other constraints and the architecture of the platform and produce correct settings for the RTOS and optimized code. We are still far from this ideal situation. It is likely, then, that the responsibility for subsystem integration will still rest with the car manufacturers but the responsibility for integrating software components onto ECUs will be assigned to Tier 1 suppliers. In this case, the burden of Tier 1 suppliers will be increased at a possibly reduced premium because of the perceived reduction in added value. This is likely to be an unstable model and major attention should be devoted to find a common ground where both car makers and suppliers find their economic return.

If the strategy followed by car makers in AUTOSAR succeeds, then it is likely that a global restructuring of the industry will take place by creating an environment where Tier 1 players with small market share will find themselves in a difficult position unless they find a way of competing on a more leveled ground with the major stake holders. In this scenario, Tier 2 suppliers including IP providers may find themselves in a better position to entertain business relations directly with the car manufacturer. Tool providers will be in a more strategic position as providers of mapping tools that make the business model feasible. Hence, it is likely that a shift of recognized value will take place from Tier 1 suppliers towards tool providers and Tier 2 suppliers. The redistribution of wealth in the design chain may or may not be a positive outcome for the health of the industrial sector. If the discontinuities are sharp, then there may be a period of instability where much effort will be required to keep the products coming out with quality and reliability problems that may be larger than the ones observed lately. However, if it is well managed, then a natural shake-up with stronger players emerging will have a double positive: more quality in the products at lower cost. An additional benefit from a real plug-and-play environment will be the acceleration of the rate of innovation. Today, the automotive sector is considered conservative and the innovations in design methods and electronic components are slow to come. For example, if a well-oiled mechanism existed to migrate from one hardware platform to another, the "optimal" solutions would be selected instead of the ones that have been traditionally used. In this case, the Tier 2 market place will also be rationalized and the rate of innovation will likely be increased.

As a final consequence, the introduction of new functionalities will be a matter of algorithm and architecture rather than detailed software and hardware selection. The trend in electronics for the automotive industry (but for other verticals as well) is clear: less customization, more standardization. For a subsystem supplier, the choice will be richer in terms of platforms but it will not require heavy investment in IC design or RTOS development. For car manufacturers, the granularity of the choices will be also richer because of interoperability. They will have the choice of selecting entire macro systems or components that could be integrated in a large automotive platform. The choice will be guided by cost, quality, and product innovation.

The final goal of the strategy is rather clear. The way of getting there is not as clear and the road has many bumps and turns that are difficult to negotiate. A positive outcome will have to come from a process of deep business and technical cooperation among all players in the design chain as well as the research community. It is a unique opportunity and a great challenge.

3) Remarks on the Needs of the Supply Chains: The design chains should connect seamlessly to minimize design errors and time-to-market delays. Yet, the boundaries among companies are often not as clean as needed and design specs move from one company to the next in nonexecutable and often imprecise forms, thus yielding misinterpretations and consequent design errors. In addition, errors are often caught only at the final integration step as the specifications were not complete and imprecise; further, nonfunctional specifications (e.g., timing, power consumption, size) are difficult to trace. I believe also that since the design process is fragmented, product optimization is rarely carried out across more than one company boundary. If the design process were carried out as in a unique "virtual" company including all the players shown above, the overall ecosystem would greatly benefit. We have seen that many of the design chain problems are typical of two very diverse verticals, the difference between the two being in the importance given to time-to-market and to the customer appeal of the products versus safety and hard-time constraints. Similar considerations could be drawn also for the consumer electronic market at large that shares many of its characteristics with the wireless communication market. This consideration motivates the view that the unified methodology and framework could be used in several (if not all) industrial vertical domains.

III. PRINCIPLES OF A UNIFIED DESIGN APPROACH: PBD

As I will discuss in Section V, most of the present approaches to SLD used in industry have the drawback of primarily addressing either hardware or software but not both. Hardware/software codesign has been a topic of interest for years, but the proposed methodologies have still treated the two aspects essentially in a segregated way. Software approaches miss time and concurrency in their semantics making it pretty much impossible to describe, synthesize, and verify hardware. Hardware approaches are too specific to the hardware semantics to work well for software designers. I also believe that the levels of abstraction available in these approaches are not rich

enough to allow the supply chain to exchange design data in a seamless fashion.

These drawbacks cause the presently available approaches to address some of the challenges presented in Section II but not all, failing especially in the integration complexity realm and in the supply chain support domain. A more powerful approach would be to use an all-encompassing methodology and the supporting tools that:

a) would include both hardware and embedded-software design as two faces of the same coin;
b) favor the use of high levels of abstraction for the initial design description;
c) offer effective architectural design exploration;
d) achieve detailed implementation by synthesis or manual refinement.

In this section I present the PBD methodology and argue that it meets these requirements.

The concept of "platform" has been around for years. The main idea of a platform is one of reuse and of facilitating the work of adapting a common design to a variety of different applications. Several papers and books have appeared in the literature discussing platforms and their use in embedded system design (see for example, [48], [95], [129], [170], [171], [173], [175], and [176].

In this section, I first introduce the use of the platform concept in industry, then I present a distilled way of considering platforms as the building blocks for a general design methodology that could be used across different boundaries. I then present the application of this methodology to an emerging system domain application, wireless sensor networks, to illustrate the unification power of platform-based design.

A. Conventional Use of the Platform Concept

There are many definitions of "platform" that depend on the domain of application.

IC Domain: a platform is considered a flexible integrated circuit where customization for a particular application is achieved by programming one or more of the components of the chip. Programming may imply metal customization (gate arrays), electrical modification (FPGA personalization), or software to run on a microprocessor or a DSP. For example, a platform may be based on a fixed micro-architecture to minimize mask-making costs but flexible enough to warrant its use for a set of applications so that production volume will be high over an extended chip lifetime. Microcontrollers designed for automotive applications such as the Freescale PowerPC are examples of this approach. The problem with this approach is the potential lack of optimization that in some applications may make performance too low and size too large.

An extension of this concept is a "family" of similar chips that differ for one or more components but that are based on the same microprocessor(s). Freescale developed the Oak Family [92] of PowerPC-based microcontrollers that cover the market more efficiently by differing in flash memory size and peripherals. The TI OMAP platform [55] for wireless communication[2] was indeed developed with the platform concept well in mind. J.-M. Chateau of ST Microelectronics commenting on its division commitment to platform-based design defines it "as the creation of a stable microprocessor-based architecture that can be rapidly extended, customized for a range of applications, and delivered to customers for quick deployment."

The use of the platform-based design concept actually started with the Phillips Nexperia Digital Video Platform (DVP). The concept of PBD for IC design has not been without its critics. G. Smith, the former Gartner Data Quest Analyst for CAD, pointed out a number of shortcomings [185] that make, in his words, PBD work well in an embedded software development context as advocated in [176] but not so for chip design. However, not a month later, in an interview [62], [213], McGregor, former CEO of Philips semiconductors was quoted: "... we redoubled the company's efforts in platform-based design. Philips embraced the idea early—in the mid'90s, The recommitment to the platform approach under my watch is among my most notable accomplishments." In another important quote: "ST's Geyres attributed ST's continued success in the set-top business to its migration from systems-on-chip to application platforms," [214]. At this time, there is little doubt that PBD has made significant inroads in any semiconductor application domain. The Xilinx Virtex II [210] family is a platform rich in flexibility offered by an extensive FPGA fabric coupled with hard software programmable IPs (up to four PowerPC cores and a variety of peripherals. The FPGA fabric is enriched by a set of "soft" library elements such as the microblaze processor and a variety of smaller granularity functional blocks such as adders and multipliers.

I believe there will be a converging path towards the platform of the future, where traditional semiconductor companies will increase the flexibility of their platforms by possibly adding FPGA-like blocks and heterogeneous programmable processors, while the FPGA-based companies will make their platforms more cost and performance efficient by adding hard macros, thus differentiating their offerings according to the markets of interest. The more heterogeneity is added to the platform, the more potential for optimizing an application at the price of a more complex design process for the application engineers who have to allocate functionality to the various components and develop code for the programmable parts. In this context, the interaction among the various components has problems similar to those faced by the system

[2]From the TI home page: "TI's OMAP Platform is comprised of market proven, high-performance, power efficient processors, a robust software infrastructure and comprehensive support network for the rapid development of differentiated internet appliances, 2.5G and 3G wireless handsets and PDAs, portable data terminals and other multimedia-enhanced devices."

companies in an inherently distributed implementation domain (e.g., cars, airplanes, industrial plants). The "right" balance among the various components is difficult to strike and the methodology I will describe later is an attempt to give the appropriate weapons to fight this battle.

PC Domain: PC makers and application software designers have been able to develop their products quickly and efficiently around a standard "platform" that emerged over the years. The "architecture" platform standards can be summarized in the following list.

1) The x86 instruction set architecture (ISA) makes it possible to reuse the operating system and the software application at the binary level3.
2) A fully specified set of busses (ISA, USB, PCI) make it possible to use the same expansion boards or IC's for different products.
3) A full specification of a set of I/O devices, such as keyboard, mouse, audio and video devices.

All PCs should satisfy this set of constraints. Both the application developers and the hardware designers benefited from the existence of a standard layer of abstraction. Software designers have long used well-defined interfaces that are largely independent from the details of the hardware architecture. IC designers could invent new micro-architectures and circuits as long as their designs satisfied the standard. If we examine carefully the structure of a PC platform, we note that it is not the detailed hardware micro-architecture that is standardized, but rather an abstraction characterized by a *set of constraints on the architecture*. The platform is an abstraction of a "family" of micro-architectures. In this case, IC design time is certainly minimized since the essential components of the architecture are fixed and the remaining degrees of freedom allow some optimization of performance and cost. Software can also be developed independently of the new hardware availability, thus offering a real hardware–software codesign approach.

System Domain: The definition of a platform is very loose. This quote from an Ericsson press release is a good example: "Ericsson's Internet Services Platform is a new tool for helping CDMA operators and service providers deploy Mobile Internet applications rapidly, efficiently and cost-effectively." The essential concept outlined here is the aspect of the capabilities a platform offers to develop quickly new applications. It is similar to the application software view of a PC platform, but it is clearly at a higher level of abstraction. The term platform has been also used by car makers to indicate the common features shared between different models. For automobiles, platforms are characterized by common mechanical features such as engines, chassis, and entire powertrains. It is not infrequent to see a number of different models even across brands share many mechanical parts, addressing different markets with optimized interior and styling. Here, the focus on subsystem commonality allows for faster time-to-market and less expensive development.

There are clearly common elements in the platform approaches across industrial domains. To make platforms a general framework for system design, a distillation of the principles is needed so that a rigorous methodology can be developed and profitably used across different design domains.

B. Platform-Based Design Methodology

The principles at the basis of platform-based design consist of starting at the highest level of abstraction, hiding unnecessary details of an implementation, summarizing the important parameters of the implementation in an abstract model, limiting the design space exploration to a set of available components, and carrying out the design as a sequence of "refinement" steps that go from the initial specification towards the final implementation using platforms at various level of abstraction [44], [129], [174].

1) Platform Definition: A *platform* is defined to be a *library of components* that can be assembled to generate a design at that level of abstraction.

This library not only contains *computational* blocks that carry out the appropriate computation but also *communication* components that are used to interconnect the computational components.

It is important to keep communication and computation elements well separated as we may want to use different methods for representing and refining these blocks. For example, communication plays a fundamental role in determining the properties of models of computation. In addition, designing by aggregation of components requires a great care in defining the communication mechanisms as they may help or hurt design reuse. In design methodologies based on IP assembly, communication is the most important aspect. Unexpected behavior of the composition is often due to negligence in defining the interfaces and the communication among the components.

Each element of the library has a characterization in terms of performance parameters together with the functionality it can support.

The library is in some sense a parameterization of the space of possible solutions. Not all elements in the library are pre-existing components. Some may be "place holders" to indicate the flexibility of "customizing," a part of the design that is offered to the designer. For example, in a Virtex II platform, part of the design may be mapped to a set of virtual gates using logic synthesis and place-and-route tools. For this part, we do not have a complete characterization of the element since its performance parameters depend upon a lower level of abstraction.

A *platform instance* is a set of components that is selected from the library (the platform) and whose parameters are set. In the case of a virtual component, the parameters are set by the requirements rather than by the implementation. In this case, they have to be considered as constraints for the next level of refinement.

This concept of platform encapsulates the notion of reuse as a family of solutions that share a set of common features (the elements of the platform). Since we associate the notion of platform to a set of potential solutions to a design problem, we need to capture the process of mapping a functionality (*what* the system is supposed to do) with the platform elements that will be used to build a platform instance or an "architecture" (*how* the system does what is supposed to do). This process is the essential step for refinement and provides a mechanism to proceed towards implementation in a structured way.

I strongly believe that function and architecture should be kept separate as functionality and architectures are often defined independently, by different groups (e.g., video encoding and decoding experts versus hardware/software designers in multimedia applications). Too often I have seen designs being difficult to understand and to debug because the two aspects are intermingled at the design capture stage. If the functional aspects are indistinguishable from the implementation aspects, then it is very difficult to evolve the design over multiple hardware generations.

2) Design Process: The PBD design process is not a fully top-down nor a fully bottom-up approach in the traditional sense; rather, it is a meet-in-the-middle process (see Fig. 2) as it can be seen as the combination of two efforts.

1) **Top-down**: Map an instance of the functionality of the design into an instance of the platform and propagate constraints.
2) **Bottom-up**: Build a platform by choosing the components of the *library* that characterizes it and an associated performance abstraction (e.g., timing of the execution of the instruction set for a processor, power consumed in performing an atomic action, number of literals for technology independent optimization at the logic synthesis level, area and propagation delay for a cell in a standard cell library).

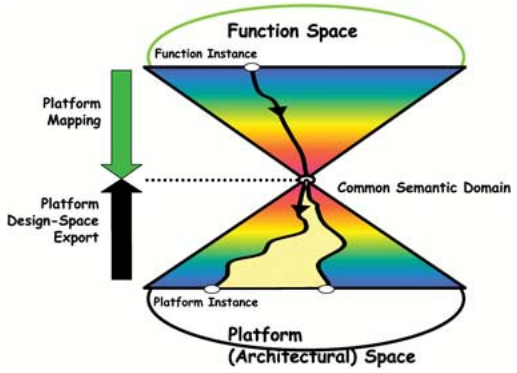


Fig. 2. ***PBD triangles.***

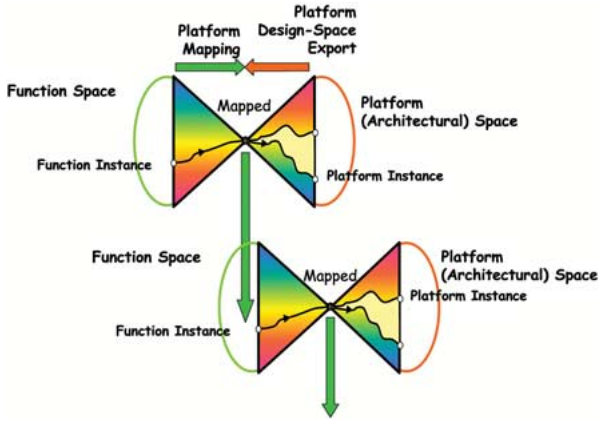


Fig. 3. ***PBD process.***

The "middle" is where functionality meets the platform. Given the original semantic difference between the two, the meeting place must be described with a common semantic domain so that the "mapping" of functionality to elements of the platform to yield an implementation can be formalized and automated.

To represent better the refinement process and to stress that platforms may pre-exist the functionality of the system to be designed, we turn the triangles on the side and represent the "middle" as the mapped functionality. Then, the refinement process takes place on the mapped functionality that becomes the "function" at the lower level of the refinement. Another platform is then considered side-by-side with the mapped instance and the process is iterated until all the components are implemented in their final form. This process is applied at all levels of abstraction, thus exposing what I call the "fractal nature of design." Note that some of the components may have reached their final implementation early in the refinement stage if these elements are fully detailed in the platform.

The resulting Fig. 3 exemplifies this aspect of the methodology. It is reminiscent of the Y-chart of Gajski, albeit it has a different meaning since for us architecture and functionality are peers and architecture is not necessarily derived from functionality but may exist independently.[3] It was used as the basis for the development of Polis [17] and of VCC [123]. The concept of architecture is well captured by the platform concept presented above.

The result of the mapping process from functionality to architecture can be interpreted as functionality at a lower level of abstraction where a new set of components, interconnects, and composition rules are identified. To progress in the design, we have to map the new

[3]This diagram together with its associated design methodology was presented independently by Bart Kienhuis and colleagues (see e.g., [130]).

functionality to the new set of architectural components. In case, the previous step used an architectural component that was fully instantiated, then that part of the design is considered concluded and the mapping process involves only the parts that have not been fully specified as yet.

While it is rather easy to grasp the notion of a programmable hardware platform, the concept is completely general and should be exploited through the entire design flow to solve the design problem. For example, the functional side of Fig. 3 can be as high level as a denotational specification (find x such that $f(x) = 0$) and the corresponding platform can be a set of algorithms for operationalizing the specification (e.g., a Newton–Raphson algorithm and a nonlinear successive over relaxation scheme) together with their performance (e.g., quadratic or linear convergence). The choice of a platform instance would be in this case the selection of the algorithm to use together with the constraints that this choice requires (e.g., differentiability of f, nonsingularity of the Jacobian at the solution for Newton–Raphson). Assuming Newton–Raphson to be the choice, then this platform instance becomes the functional specification for the next layer. In this case, a library of linear equation solvers to be used in the Newton–Raphson algorithm is then the next layer platform. We can continue along this line of reasoning until we decide to use a particular computing platform for the implementation of the fully specified algorithm that is available.

3) Considerations on Use of PBD: In PBD, the partitioning of the design into hardware and software is not the essence of system design as many think, rather it is a consequence of decisions taken at a higher level of abstraction. Critical decisions are about the architecture of the system, e.g., processors, buses, hardware accelerators, and memories, that will carry on the computation and communication tasks associated with the overall specification of the design.

In the PBD refinement-based design process, platforms should be defined to eliminate large loop iterations for affordable designs. They should restrict the design space via new forms of regularity and structure that surrender some design potential for lower cost and first-pass success. The library of functional and communication components is the design space that we are allowed to explore at the appropriate level of abstraction.

Establishing the number, location, and components of intermediate "platforms" is the essence of PBD. In fact, designs with different requirements and specifications may use different intermediate platforms, hence different layers of regularity and design-space constraints. The tradeoffs involved in the selection of the number and characteristics of platforms relate to the size of the design space to be explored and the accuracy of the estimation of the characteristics of the solution adopted. Naturally, the larger the step across platforms, the more difficult is predicting performance, optimizing at the higher levels of abstraction, and providing a tight lower bound. In fact, the design space for this approach may actually be smaller than the one obtained with smaller steps because it becomes harder to explore meaningful design alternatives and the restriction on search impedes complete design-space exploration. Ultimately, predictions/abstractions may be so inaccurate that design optimizations are misguided and the lower bounds are incorrect.

The identification of precisely defined layers where the mapping processes take place is an important design decision and should be agreed upon at the top design management level. Each layer supports a design stage where the performance indexes that characterize the architectural components provide an opaque abstraction of lower layers that allows accurate performance estimations used to guide the mapping process.

This approach results in better reuse, *because it decouples independent aspects, that would otherwise be tied, e.g., a given functional specification to low-level implementation details, or to a specific communication paradigm, or to a scheduling algorithm*. It is very important to define only as many aspects as needed at every level of abstraction, in the interest of flexibility and rapid design-space exploration.

C. Application of PBD: Wireless Sensor Network Design

In this section, I demonstrate that PBD is applicable not only to digital designs and hardware/software codesign approaches (the most obvious applications) but also to design problems as different as wireless sensor networks. In these examples, I emphasize the levels of abstraction and their relative positions, as well as the way they relate to the overall design flow. I invite the interested readers to see [45], [61], and [168], for a set of different applications that include hardware/software co-design, analog design, automotive electronic system design, and communication design both on-chip and at the system level.

The application of Wireless Sensor Networks technology [54] to the design of field-area networks for industrial communication and control systems has the potential to provide major benefits in terms of flexible installation and maintenance of field devices, support for monitoring the operations of mobile robots, and reduction in costs and problems due to wire cabling [205], [217].

The software for control applications within industrial plants is usually written by process or mechanical engineers that are expert in process control technology, but know little of the communication and sensing infrastructure that has to be deployed to support these algorithms. On the other side, the communication infrastructure is designed by communication engineers that know little about process control technology. Moreover, the adoption of wireless technology further complicates the design of these networks. Being able to satisfy high requirements on communication performance over

an unreliable communication channel is a difficult task. Consequently, the gap between the control algorithm designers and the network designers will inevitably increase. This phenomenon might delay the adoption of wireless sensor networks technology.

The platform-based methodology can contribute to the solution of these problems focusing the effort on the definition of a *clear set of abstraction layers* across which the design progresses. For a more detailed overview of the methodology, the interested readers is referred to [28]–[30] and [180]. The different abstraction layers presented below and their relationship are shown in Fig. 4.

The first layer is an application interface called *sensor network service platform (SNSP)*. The SNSP defines a set of services available to the application engineer to specify the target application formally without dealing with the details of a particular network implementation. The SNSP offers a *query service (QS)* used by controllers to get information from other components, a *command service (CS)* used by controllers to set the state of other components, a *timing/synchronization service (TSS)* used by components to agree on a common time, a *location service (LS)* used by components to learn their location, and a *concept repository service (CRS)* which maintains a map of the capabilities of the deployed system and it is used by all the components to maintain a common consistent definition of the concepts that they agreed upon during the network operation. While the SNSP description suffices to capture the interaction between controllers, sensors, and actuators, it is a purely functional description, which does not prescribe how and where each of these functions will be implemented. Hence, information such as communication protocols, energy, delay, cost, and memory size are not included.

The second abstraction layer is called *Sensor Network Ad hoc Protocol Platform (SNAPP)* [29]. The SNAPP defines a library of communication protocols and the interfaces that these protocols offer to the SNSP. In Fig. 4, RAND and SERAN are two examples of protocols that populate the SNAPP.

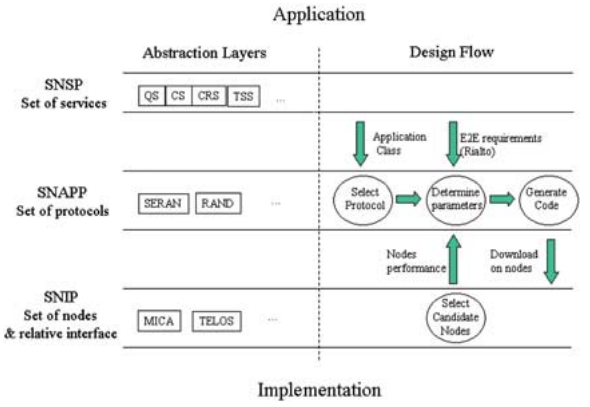


Fig. 4. *PBD for wireless sensor networks.*

Once the communication protocol is selected, it must be implemented on a set of physical nodes. A description of the actual hardware platform is given by the *sensor network implementation platform (SNIP)* [180]. In Fig. 4, MICA [155] and TELOS [156] are two commonly available hardware platforms.

The process of mapping the SNSP description to a SNAPP instance and eventually to a SNIP instance goes through a set of steps. First, the selected topology and communication protocol must be ensured to be capable of supporting the sensing, actuation and communication requirements implied by the application.

Once the constraints on sensing, actuation, and communication have been derived, the methodology requires an abstraction of the physical layer properties of the proposed hardware platform (candidate SNIP instance) and selects an adequate topology and communication protocol among the ones available in the SNAPP. Finally, the parameters of the protocol are synthesized so that the given constraints are satisfied and energy consumption optimized.

Tools that help bridging between two different layers of abstraction have been developed for particular application domains [28].

Summarizing, the methodology:

1) allows the control algorithm designer to specify the application using a clear interface that abstracts the drudgeries of the network implementation;
2) derives a set of constraints on the end-to-end (E2E) latency and packet error rate that the network has to satisfy starting from the application description;
3) derives a solution for MAC and Routing that satisfies requirements and optimizes for energy consumption using the E2E requirements and an abstraction of the hardware platform;
4) maps the communication protocol to the hardware nodes and the PLC.

The introduction of the levels of abstraction and of the tools allows a seamless path to implementation from high-level application-driven specifications where various supply chain players can optimize their contributions. The layers of abstraction define the boundaries across which the design transition occurs. Note that application designers when using this approach can quickly adapt to a new implementation platform (even an heterogeneous one where different nodes may be provided by different vendors) exploiting the advantages of the technology without having to pay the price of redesigning their applications.

The definition of the platform levels allows us also to develop synthesis and verification tools that would have been impossible otherwise. This layered approach is

reminiscent of the standard ISO-OSI layering but it has important differences as the layers are not necessarily predefined and standardized and the potential of optimization is much greater. In some sense, PBD retains the favorable aspects of the layered approach to communication design while improving performance and design freedom.

D. Platform-Based Design and Model-Driven Development

The paradigm that most closely resembles PBD is model-driven (software) development (MDD). MDD is a subject of intense research and investigation in the software development community as it bears much promise to improve the quality of software. For an excellent review of the state of the art and of challenges that MDD poses to the software community, I recommend the March 2006 issue of the *IBM Systems Journal* [66] and, in particular, the paper "Model-Driven Development: The good, the bad and the ugly" by B. Hailpern and P. Tarr, for a deep analysis of the pros and cons of the approach.

MDD is based on the concept of model-driven architecture. The OMG defines the term model-driven architecture (MDA) to be as follows: "MDA is based on a Platform-Independent Model (PIM) of the application or specification's business functionality and behavior. A complete MDA specification consists of a definitive platform-independent base model, plus one or more Platform-Specific Models (PSMs) and sets of interface definitions, each describing how the base model is implemented on a different middleware platform. A complete MDA application consists of a definitive PIM, plus one or more PSMs and complete implementations, one on each platform that the application developer decides to support. MDA begins with a model concerned with the (business-level) functionality of the system, independent of the underlying technologies (processors, protocols, etc.). MDA tools then support the mapping of the PIM to the PSMs as new technologies become available or implementation decisions change," [98].

The concept of separation of concerns between function and platform is clearly stated. The use of the mapping of functionality to platforms as a mean to move towards the final implementation is also expressed. However, the similarities between the two approaches end here as the definition of platform is not fully described nor are the semantics to be used for embedded software design.

The Vanderbilt University group [127] has evolved an embedded software design methodology and a set of tools based on MDD. In their approach, models explicitly represent the embedded software and the environment it operates in and capture the requirements and the design of the application, simultaneously. Models are descriptive, in the sense that they allow the formal analysis, verification, and validation of the embedded system at design time. Models are also generative, in the sense that they carry enough information for automatically generating embedded software using the techniques of program generators. Because of the widely varying nature of embedded systems, the Vanderbilt researchers emphasize that *a single modeling language may not be suitable for all domains*; thus, modeling languages should be domain-specific (DSL). These languages have a significant impact on the design process [118] for complex software systems. In embedded systems, where computation and communication interact with the physical world, DSLs offer an effective way to structure information about the system to be designed along the "natural dimensions" of the application [86]. *I take the position that DSLs for embedded systems should have a mathematically manipulable representation*.

This view goes against the use of a general language for embedded systems and favors customization to obtain better optimization and easier adoption. However, customization carries obvious drawbacks in terms of development costs and support efforts. To decrease the cost of defining and integrating domain-specific modeling languages and corresponding analysis and synthesis tools, the model-integrated computing (MIC) [127] approach is applied in an architecture, where formal models of domain-specific modeling languages-called metamodels play a key role in customizing and connecting components of tool chains. The generic modeling environment (GME) [127] provides a framework for model transformations enabling easy exchange of models between tools and offers sophisticated ways to support syntactic (but not semantic) heterogeneity. The KerMeta metamodeling workbench [128], [159] is similar in scope.

In synthesis, MDD emphasizes design by (whenever possible automatic) model transformations. Model-based approaches have been applied for years in the hardware domain where one can argue that since the introduction of logic synthesis, this approach has had great success. Most of the formal approaches to hardware design are indeed model driven in the sense that a design model is successively transformed into hardware. In embedded software, the approach still has to be fully exploited as using a model-driven method requires the description of the software with mathematical models, a step that for most software designers is not easy. DSLs will probably help in pushing for the adoption of MDD in the embedded software community since it is possible to design these languages to meet the specific needs of a homogeneous group of designers thus allowing them to be more effective in expressing their designs. However, if indeed each design group is going to have its specific language, the problem will be how to interface the various parts of the design so that the composition can be analyzed and verified. I believe that this issue can be resolved only if the semantics of the languages are well understood and the interaction among parts described with different languages is mathematically well characterized. The Vanderbilt

group is addressing some of these issues with semantic anchoring of DSLs using abstract semantics based on abstract state machines [32], [103]. In addition, the MILAN framework [126] offers a number of simulation, analysis, and synthesis tools that leverage the MIC framework. A recent approach to "gluing" parts described by different languages consists of using higher level programming models and languages for distributed programming, called *coordination* models and languages [46], [50]. In the coordination model approach, one can build a complete programming model out of two separate pieces—the *computation model and the coordination model*. The computation model allows programmers to build a single computational activity, a single-threaded step-at-a-time computation. The coordination model is the glue that binds separate activities into an ensemble. The similarity with the separation between computation and communication in PBD is strong.

A coordination language is "the linguistic embodiment of a coordination model" [46]. The most famous example of a coordination model is the Tuple Space in *Linda*, a language introduced in the mid 1980s, that was the first commercial product to implement a virtual shared memory (VSM), now popularly known as tuples-pace technology for supercomputers and large workstation clusters. It is used at hundreds of sites worldwide [16]. Linda can be seen as a sort of assembly level coordination language since it offers:

1) very simple coordination entities, namely, active and passive tuples, which represent processes and messages, respectively;
2) a unique coordination medium, the Tuple Space, in which all tuples reside;
3) a small number of coordination laws embedded in four primitives only.

Coordination languages can be built on Linda to offer a higher level of abstraction construct to simplify the synchronization and message passing among the components. Many coordination languages have been built over the years. An excellent review of Linda derivatives and coordination languages such as *Laura* and *Shade* can be found in [161].

Once more, I advocate the add a strong mathematically sound semantics to the linguistic approach to composition. This is indeed the contribution of some of the environments for heterogeneous models of computation such as Ptolemy II and Metropolis.

E. Concluding Remarks on PBD

The notion of PBD presented in this section is being adopted rather widely by the EDA companies who are active in the system space or that are eyeing that market. CoWare [170] and Mentor Graphics [52] use platforms in their architectural design and design-space exploration tools pretty much in the sense I introduced here. Cadence and National Instruments use the concepts of platforms in the description of their tools and approaches using diagrams similar to Fig. 2.

I believe PBD serves well the purpose of the supply chain as the layers of abstraction represented by the platforms can be used to define the hand-off points of complex designs. In addition, the performance and cost characteristics associated to the platforms represent a "contract" between two players of the design chain. If the platform has been fully specified with performance and cost given by the supplier, then the client can design at his/her level of abstraction with the assumption that the "contract" will be satisfied [82], [134]. If the supplier has done his/her homework well, the design cycles are considerably shortened. If one or more of the components of the platform instance chosen by the client is not made available by the supplier, but it has to be designed anew, the performance assumed by the client can serve as a specification for the supplier. In both cases, the "contract" is expressed in executable form and prevents misunderstandings and long design cycles.

The platform concept is also ideal to raise the level of abstraction since it does not distinguish between hardware and software but between functionality and architecture. Hence, the design-space exploration can take place with a more degrees of freedom than in the traditional flows. In addition, the partitioning between hardware and software components can be done in an intelligent and optimized way.

On the other hand, PBD does require a specific training of designers to guide them in the definition of the "right" levels of abstraction and of the relationships among them. It does benefit from the presence of supporting tools for analysis, simulation, and synthesis organized in a well-structured design flow that reflects the relationships among the platforms at the different layers of abstraction. Designers have to be careful in extracting implementation aspects they want to analyze from behavior of their design. In my experience of interaction with industry on importing PBD, this has possibly been the most difficult step to implement. However, once it is well understood, it gave strong benefits not only in terms of design time and quality, but also in terms of documentation of the design.

IV. STATE-OF-THE-ART IN EMBEDDED SYSTEM DESIGN REVIEW USING THE PBD PARADIGM

As mentioned several times, the methodology, framework, and tools presented above can serve as an integration framework to leverage the many years of important work of several researchers. As done in [63], I use the diagram of Fig. 5, a simplification of Fig. 3, to place in context system-level design approaches reported in literature. This classification is not only for taxonomy purposes. It also shows how to combine existing approaches into the unified

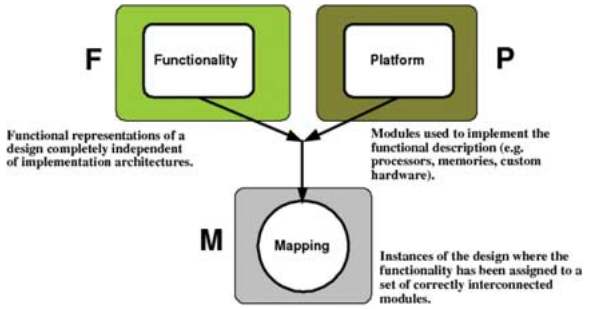


Fig. 5. ***Function architecture mapping.***

view offered by PBD to build optimized flows that can be customized for particular applications.

A. Representing the Functional Aspects of an Embedded System

We argued that for true system level design, we must be able to capture the functionality of the design at the highest possible level of abstraction without using implicit assumptions about an implementation choice. Since a most appealing feature of capturing the functionality of the design is to be able to execute it on a computer for verification and analysis, it is natural that designers and researchers cast the problem in terms of design languages.

1) Languages for Hardware Design: Because of the popularity and the efficiency of C, several approaches for raising the levels of abstraction for hardware design are based on C and its variants.[4] C has been used successfully to represent the high-level functional behavior of hardware systems and to simulate its behavior. A simulation amounts to running the compiled C code and hence is very fast, limited by the speed of the simulation host and by the quality of the compiler. *The main problem with this approach is the lack of concurrency and of the concept of time in C*. In fact, hardware is inherently concurrent and time is essential to represent its behavior accurately. In addition, C has been designed with standard programming application in mind and in its constructs, it relies upon standard communication mechanisms through memory that are inadequate, to say the least, for hardware representation. For these reasons, a number of derivative languages have been introduced, some with more success than others. The pioneering work in this field was done by De Micheli and students [81], [152] who discussed the main problems of using C as a hardware description language from which a register transfer level (RTL) description could be synthesized. Commercial offerings such as Mentor CatapultC, Celoxica Handel-C [47], C2Verilog, and Bach [85] defined a subset of ANSI C to do either synthesis or verification.

More recently, there has been a strong interest in languages that are derived from C or C++ and that *explicitly capture the particular aspects of hardware*. In particular, SystemC [97], [196] and SpecC [68] stand out.

SystemC is a class library of the C++ language while SpecC is a super set of ANSI C. Both have special constructs to represent hardware concepts, to support concurrency and a rich set of communication primitives. The resemblance to C of these languages is then mainly syntactical while their semantics are quite different.

The usefulness to a designer of a language is not only the capability of representing his/her intent but also the support given in terms of verification, formal analysis, and synthesis. Both SystemC and SpecC are not directly synthesizable nor formally verifiable. To verify formally or synthesize a design expressed in System C or SpecC, we need either to subset the language (SystemC) or to go through a set of manual or automatic transformations to yield a synthesizable representation (SpecC).

SystemC is used mainly for simulation. Several SystemC simulation engines are available (one is open source). Of course, the performance of the simulation and the leverage in design representation comes from the level of abstraction of the models described in these languages. There are a few synthesis tools that generate RTL from SystemC-like languages. Companies like Mentor, Synopsys, and Forte Design offer tools in this domain. The jury is still out regarding the degree of acceptance of hardware designers for this kind of tool as the quality of the results is mixed. Sometimes the quality is comparable and even better than the one of human designs, sometimes it is definitely worse.

An alternative approach to raising the level of abstraction is to extend existing RTLs to cover constructs that help in describing and verifying higher levels of abstraction. In this context, SystemVerilog [90], [193] has been accepted with interest by the hardware design community as it builds upon the widely used Hardware Description Language (HDL) Verilog. While SystemVerilog can be used almost for everything SystemC can do, the opposers of this approach list as drawbacks the difficulty of using SystemVerilog for system designers with software background and the difficulty in expressing some of the concepts important to system design.

An interesting approach to hardware synthesis and verification is offered by BlueSpec [27]. BlueSpec takes as input a SystemVerilog or a SystemC subset and manipulates it with technology derived from term rewriting systems (TRS) [15] initially developed at MIT by Arvind. The idea of term rewriting was developed in computer science and is the basis of several compiler techniques. It offers a nice environment to capture successive refinements to an initial high-level design that are guaranteed correct by the system. The appeal of the approach is that

[4]We recommend to the interested reader the excellent survey paper by Edwards for a critical review of C-derived languages [74].

designers can maintain their intent throughout their design process and control the synthesis steps. This is a significant deviation from the work on high-level synthesis pioneered by the CMU school [67] where from a high-level design representation, both architecture and micro-architecture implementations were automatically generated. The general consensus today is that the chasm between an algorithmic description and an implementation is just too wide to be able to obtain a good implementation.

2) Languages for Embedded Software Design: Traditionally, abstract analysis and design have been divorced from implementation concerns, helped by the Turing abstraction that has simplified the task of programming by decoupling functionality and the physical world. Because of this dichotomy, today, embedded software designers use low-level facilities of a real-time operating system (RTOS), tweaking parameters such as priorities until the system seems to work. The result is, of course, quite brittle.

The main difficulty faced by embedded software designers are programmer productivity and design correctness. This is particularly true when the software is to be distributed over a network, as in the case of avionics and automotive applications. In this case, side effects due to the interaction of different tasks implemented on different elements of the design make the verification of the correct behavior of the system very difficult since the traditional paradigm of adjusting priorities does not apply.

Most of the design methodologies in use in the embedded system industry have been borrowed by standard software design practices, and emphasis is placed on development processes rather than on the content of the design. However, in the industrial sectors where safety is a primary concern, there has been an interesting shift towards the use of languages that have intrinsic correctness guarantees and for which powerful analysis and synthesis methods are possible. Ironically, while in the case of hardware system design the state-of-the-art is the attempt at adapting languages like C and C++ typically used for software, the most advanced paradigm in embedded software design is borrowed from hardware design! In particular, the most elegant approach is the extension of the synchronous design paradigm to software design.

Synchronous Languages: The goal of synchronous languages is to offer strong formal semantics that make the verification and the code generation problem easier *by construction*. The work of the French school on synchronous languages [25], and in particular, Esterel [26], Lustre [104], and Signal [102] with their industrial derivatives (e.g., Esterel Studio and Lustre-SCADE from Esterel Technology and Signal-RT Builder from TN-Software Division of Valiosys), has been at the forefront of a novel way of thinking about embedded software. The synchronous design languages approach has made some significant in-roads in the safety critical domain, especially in avionics.

The synchronous languages adopt the synchronous hardware design paradigm in the sense that they assume that computation occurs in two separate phases, computation and communication, that do not overlap. Often this concept is described as communication and computation taking zero time, while a better way would be to say that the actual time taken for communication and computation does not matter from a functional point of view as they do not overlap. The notion of time is then one of *logical time* that relates to phases of behavior and sequencing. In this model, behavior is predictable and independent of the actual implementation platform. This is similar to synchronous hardware where as long as the critical path of the combinational hardware between latches takes less time than the clock cycle, the behavior of the hardware is independent of the actual delays of the combinational logic.

The adoption of these languages in the embedded software community has been limited to safety-critical domains such as aviation and automotive, but I believe that they could and should have a larger application.

3) Models of Computation: When we look at the approaches used for software design and hardware design, we see that the assumptions about the execution platform are embedded in the formulation. Take a language like C. The notion of sequential execution tied to the Von Neumann computing architecture is deeply embedded in its semantics. Our paradigm of separating functionality from implementation is orthogonal to this. We insist that the formulation must maintain maximum flexibility and capture the functionality of the design using the least constrained formalism. The design process then is originated in a "pure" way that allows a designer complete freedom in choosing an implementation. In addition, formal verification and implementation by synthesis and refinement is much better defined and more powerful using this approach. Hence, the attention on design capture must be shifted to mathematically sound representations [75] from which a design process can generate (possibly in an automatic fashion) an implementation. These representations are expressed in an appropriate language of course, but the syntactical aspects that make the language usable and appealing to the designers are secondary. The decision on the mathematical properties of the representation is the critical one. We have already seen hardware and software description languages that carry a rigorous semantics (e.g., synchronous languages), but they lack the generality needed to face the challenges set forth in Section II. There is an interesting tradeoff to play: on one hand, having a very expressive mathematical formalism, such as *discrete time*, allows us to use one model for a large number of designs; on the other, having a mathematical model with strong properties, such as *finite-state machines* or *data flow*, reduces the space of expressible designs but makes formal analysis and

synthesis much easier. How to strike a balance between the two requirements is a part of the art of system design formulation. The debate between one universal model versus a set of *ad hoc* models has been raging for years. The direction taken by the debate seems to favor the use of models to represent application-specific designs but with the possibility of mixing these models in a more general framework, thus trying to leverage the strong properties of each of the models and the generality offered by their composition.

Heterogeneous Models of Computation: The issue of mixing different models is nontrivial. Much has been published on this topic (e.g., see [36], [75], [76], [136], [146], [147], and [172]). In my view, making heterogeneous models of computation interact is a mathematically ill-posed proposition. There is no precise meaning that can be associated to this communication: information must be added to the semantics of the original models to specify separately the meaning of the interaction. In all the publications about this issue, the interaction model is chosen either by the user or by the people who built the environment. I think it is preferable for the interaction to be specified by the user since he/she is the one to connect the different models and, hence, must know the semantics of that construct. The tool builder's responsibility is to take into account the wishes of the user and point out potential inconsistencies and contradictions. How the models are made to communicate has strong impact on the analyzability of the overall design. I believe that a reasonable approach for the user is to find a common semantic domain to embed the different models so that the interaction can be specified in a controlled environment. Then, the implications of the interconnection of the models in the common semantics domain should be projected back into the respective native domains of the different models for analysis [163].

Another approach to analyze models of computation is represented by the so-called LSV model [136]. In this denotational model, a system is represented by a set of behaviors. Behaviors are sets of events. Each event is characterized by a data value and a tag that may be used to represent partial orders, total orders, logical and physical time. Complex systems are derived through the parallel composition of simpler sub-systems, by taking the conjunction (intersection) of their corresponding sets of behaviors. This is the lowest level common semantic domain where models of computation and their heterogeneous composition can be embedded. This model has been used in a number of approaches to analyze the refinement of "single-clock" synchronous models into less constrained implementations [4], [42], [57] as well as design environments [19].

Yet another approach is about analyzing the behavior of interfaces to find whether two models can be composed in an appropriate semantic domain. *Interface automata* have been proposed in [59] and used in [137] for heterogeneous modeling within Ptolemy [76]. Interface automata are low-level finite-state machines equipped with their usual synchronous product. They model in a generic way (an abstraction of) the microstep interface behavior of different Ptolemy II domains. Interface automata are used as a typing mechanism between domains. Informally, if P and Q are models of respective domains $dom_P \neq dom_Q$ and respective interface automata $\mathcal{P}$ and $\mathcal{Q}$, then the these two domains can be composed if the automaton $\mathcal{P} \times \mathcal{Q}$ is deadlock free. In case the two domains cannot be composed, there is a synthesis procedure that can be developed if certain conditions are satisfied [164], [165].

Environments for Heterogeneous Models of Computation: Environments for capturing designs based on heterogeneous models of computation have been around for a few years both in academia and in industry. We review, in this section, the most relevant examples.

Ptolemy: Among the environments for heterogeneous models of computation, Ptolemy II [176] is perhaps the most well-known. Ptolemy II is the successor of Ptolemy Classic, started jointly by Lee and Messerschmitt in 1990 [35], which was the first modeling environment to systematically support multiple models of computation, hierarchically combined. Ptolemy II introduced the notion of domain polymorphism (where components could be designed to be able to operate in multiple domains, for example, in the discrete time and in the continuous-time domain) and modal models (where finite-state machines are combined hierarchically with other models of computation; in this case, the finite-state machine represents the mode switching operation of the system). Ptolemy II incorporates a continuous-time domain that, when combined with the modal modeling capability, yields hybrid system modeling. Ptolemy II has a sophisticated type system with type inference and data polymorphism (where components can be designed to operate on multiple data types) and a rich expression language. The concept of behavioral types emerged (where components and domains could have interface definitions that describe not just static structure, as with traditional type systems, but also dynamic behavior).

The approach of Ptolemy to modeling is actor oriented. This model is shared by other environments such as Metropolis to be presented in details further on [21]. The term actor was introduced in the 1970s by Hewitt of MIT to describe the concept of autonomous reasoning agents [114]. The term evolved through the work of Agha and others to describe a formalized model of concurrency [8], [9]. Agha's actors each have an independent thread of control and communicate via asynchronous message passing. The Ptolemy actors are still conceptually concurrent; and although communication is still through some form of message passing, it need not be strictly asynchronous.

The Ptolemy framework is an ideal playground for researchers to experiment with novel models of computation and to analyze their properties. It is essentially

geared towards embedded software as hardware architectures do not have a particular place in the framework.

ForSyDe and SML-Sys: The formal system design (ForSyDe) [172] provides a set of formal design-transformation methods for a transparent refinement process of a system model into an implementation model that is optimized for synthesis. Similarly to other modeling approaches, in ForSyDe the design process starts with a *functional specification*. As opposed to the generality of the Ptolemy paradigm, ForSyDe models of computation are based on the synchronous paradigm, that is on the assumption that the components of the design interact according to precise rules that forbid computation and communication to overlap. This system specification, which must be expressed in the Haskell language [7], a functional language as opposed to Ptolemy that uses Java, an imperative language, is then refined through the stepwise application of formally defined design transformations. Designers can either select among a predefined set of basic transformation rules from a given library or define new transformation rules to capture the characteristics of target execution platform.

SML-Sys is based on the same approach as ForSyDe but it uses instead the functional language Standard-ML [142]; more importantly, it aims at enabling the use of heterogeneous models of computation at the specification phase as in Ptolemy [146]. Noticing that functional languages are not widely used in the industry due mainly to issues of efficiency and reusability, the authors of ForSyDe and SML-Sys have recently proposed a type-safe framework for the *untimed model of computation* (UMoC) that uses C++ while supporting a higher order functional programming style [147].

Behavior-Interaction-Priority (BIP) framework: The BIP [96] framework for modeling heterogeneous real-time components, integrates results developed at Verimag over the past five years. BIP supports a component construction methodology based on the thesis that components are obtained as the superposition of three layers. The lower layer describes behavior. The intermediate layer includes a set of connectors describing the interactions between transitions of the behavior. The upper layer is a set of priority rules describing scheduling policies for interactions. Layering implies a clear separation between behavior and structure (connectors and priority rules).

BIP uses a parameterized binary composition operator on components. The product of two components consists of composing their corresponding layers separately. Parameters are used to define new interactions as well as new priority rules between the composed components. BIP provides a mechanism for structuring interactions involving strong synchronization (rendezvous) or weak synchronization (broadcast). Synchronous execution is characterized as a combination of properties of the three layers. Finally, timed components can be obtained from untimed components by applying a structure preserving transformation of the three layers. BIP allows us to consider the system construction process as a sequence of transformations in the three-dimensional space it supports: Behavior $\times$ Interaction $\times$ Priority. A transformation is the result of the superposition of elementary transformations for each dimension. This provides a basis for the study of property preserving transformations or transformations between subclasses of systems such as untimed/timed, asynchronous/synchronous, and event-triggered/data triggered.

Industrial Frameworks: There are several examples of actor-oriented industrial frameworks with, albeit limited, support for heterogeneous model of computation.

1) *Signal Processing Worksystem (SPW)* [182] was derived from the first approach to actor-oriented design even earlier than Ptolemy. SPW deals with data flow designs and uses discrete-event semantics to simulate designs. SPW was developed by Comdisco in the 1980s. It was later acquired by Cadence Design Systems in the early 1990s and placed in the Alta System division. In 2003, SPW was acquired by CoWare. Similar tools were Cossap [133] later acquired by Synopsys and the Mentor Design Workbench based on the Berkeley language Silage [115] and originated by the IMEC Cathedral system [169] (no longer available commercially).
2) *Simulink from The MathWorks* [183] has a functionality similar to SPW but can simulate mixed data flow-continuous time designs. The composition of the two domains is taken in discrete time as the common semantic domain. In fact, continuous time is discretized by the integration formulae embedded in the simulation engine before being composed with the data-flow components that are converted in discrete time as well. In concert with StateFlow [189], a program that analyzes designs described as finite-state machines with the semantics of StateCharts [105], Simulink forms the most popular approach to functional design today.
3) *LabVIEW from National Instruments* [125] is making significant inroads in functional design especially as a virtual measurement platform the company offers.
4) *Cocentric System studio from Synopsys* [192] uses, not surprisingly given the role of J. Buck in the development of the tool [34], some of the Ptolemy concepts, in particular, the hierarchical composition of models of computation. It can capture data flow diagrams and finite-state machines. Both modeling styles work together and can be nested to any level of depth. For example, a designer can model a state machine in which each state is best modeled as a dataflow, and the algorithm contained within such a dataflow block can contain further dataflow or control models.

Among these languages, Modelica [154], an object-oriented, domain-specific modeling language, stands out. Modelica was designed by H. Elmqvist and colleagues at Lund to allow convenient, component-oriented modeling of complex systems, e.g., systems containing mechanical, electrical, electronic, hydraulic, thermal, control, electric power, or process-oriented subcomponents. The free Modelica language, free Modelica libraries, and Modelica simulation tools are available, ready-to-use, and have been utilized in demanding industrial applications, including hardware-in-the-loop simulations. Dymola offered by Dynasim [72] (very recently acquired by Dassault) is a modeling environment and a simulation engine based on the Modelica language.

B. Representing Architectures

In the PBD framework, the design proceeds from the functional representation towards the implementation by mapping the components of the functionality onto platform instances (architectures). To implement this design methodology, architectures have to be appropriately represented. In Fig. 5, the right branch corresponds to this representation.

Architecture is a broad term used in very liberal ways. In the computing systems literature, the instruction set architecture (ISA) represents the highest level of abstraction that specifies what the processor is supposed to implement. The micro-architecture is the set of components and their interconnect that realizes the ISA.

In my view, the concept has been used mostly to refer to the *structure* of a design. Hence, *an architecture is a netlist that establishes how a set of components is connected*. It is then possible to refer to an architecture as the basic structure of an algorithm as well as a set of ECUs connected by a CAN bus.

I also include the *capabilities* of the components as an element of the architecture especially when thought in the PBD domain. For example, the set of functions that a block can compute are elements of the architecture. A Xilinx FPGA block implements a look-up table (LUT) with four logical inputs; thus, it can compute any of 32 logic functions that can be formed with four Boolean input variables. This information is part of the platform (architecture) functional characterization.

In the PBD framework, an architectural block is decorated with its "cost," i.e., the time and power it takes to compute. The communication infrastructure with its components and performance information is an essential part of the architecture. The "cost" is either a static evaluation for the platform component or is computed dynamically by executing a model of the operation of the component where the basic steps of the execution have a "cost" model. For example, the performance of a processor when executing a program can be dynamically evaluated having the time consumed by each basic operation of the processor model; if the model is at the ISA level, the performance of the processor when executing each instruction has to be provided. In this case, a *method for computing the cost* is associated with the component and the actual cost can only be computed when a functionality is mapped to the component.

1) Software Architecture Description: In literature, we find descriptions of software systems involving terms such as "client-server organization" and "layered system." These descriptions are typically expressed informally and accompanied by diagram drawings indicating the global organization of computational entities and the interactions among them. While these descriptions may provide useful documentation, the current level of informality limits their usefulness. In the software domain, it is generally not clear precisely what is meant by these architectural descriptions. Hence, it may be impossible to analyze an architecture or determine nontrivial properties.

Unified Modeling Language (UML): UML [31] is an approach to software development that stresses a successive refinement approach to software design. UML is a standard supported by the Object Management Group (OMG) [158]. The OMG is an industrial consortium which produces and maintains computer industry standards for inter-operable enterprise applications. It is by now the leading standardization body for modeling technologies and processes. UML is the result of fusing the concept of visual languages with the one of object oriented languages. The language has broad scope and a structure based on diagrams (Use Case, Interaction, Collaboration, State, and Physical) that allows one to customize and extend it to particular domains of application. The semantics of the language is left unspecified except for the state diagrams that have the semantics of state charts [105], a formalism for the description of hierarchical state machines.

The original focus was on the description of software architecture, i.e., of the causal, communication, and hierarchy relationships. UML has nice syntactical features including a graphical representation of the diagrams. Several companies are standardizing on this language for their software development.

The generality of UML is paid by the difficulty in expressing constructs that are common to standard programming languages such as one-dimensional and sequential operations (assignments, loops, branches), the difficulty in performing code generation efficiently and the difficulty in having tools that support it in full. Its graphical nature makes it a favorite for software developers to describe the structure of their programs but is in the way in the case textual descriptions are more compact.

UML initially targeted software development, and as such it missed the concept of time, a very important concept for hardware and real-time designs, and has difficulty expressing concurrency. In UML 2.0, the latest release of the language, the "Schedulability, Performance and Time" (SPT) UML profile definition is intended to

address time following the pioneering work of Selic [177] with real-time object-oriented modeling (ROOM). Yet, the historical background of middleware software engineering can be felt, with paradigms drawn straight from the world of software components and asynchronous communicating agents or real-time OS underlying many modeling notions.

Profiles refine UML for specific applications. This is at the same time a strength and a weakness of the language; today, there are more than 300 UML profiles! The profiles may be inconsistent and overlap in ways that are not fully documented.

Among this plethora of profiles, a novel UML profile, *SysML* [99], is of great potential interest for embedded system design. SysML represents a subset of UML 2 with extensions to support the specification, analysis, design, verification, and validation of systems that include hardware, software, data, personnel, procedures, and facilities. It has been approved by OMG in May 2006. The four pillars of SysML are the capability of modeling of requirements, behavior, structure, and parameters. It is a complex language that is still in its infancy. Strong support from vendors and application engineers is an obvious prerequisite for its wide adoption. It does have interesting aspects that may make its use quite powerful in the development of embedded systems with real-time requirements.

There has been a substantial rationalization of the UML-based tool market as two players have the dominant position after targeted acquisition: IBM with its Rational tools [120][5] and Telelogic with Rhapsody [197][6] and Tau [198] are the most visible providers of UML-based software development tools. Rhapsody and Rational Rose are especially targeted to embedded system designers. On November 2006, a new Tau release (3.0) pulls out the requirements profile that was embedded in the systems modeling language (SysML) and makes it available to anyone who does UML modeling.

With UML, we are still not able to check that the overall description is consistent in the sense that the parts fit together appropriately. Architectural description languages (ADL) are a step toward a complete theory of architectural description [88] that allows us to reason about the behavior of the system as a whole. For example, Wright [10] provides a formal basis for specifying the interactions among architectural components by providing a notation and underlying theory that gives architectural connection explicit semantic status.

Eclipse: Eclipse [91] is an open source platform comprising a plug-in Java-based environment for building software. It is the basic framework upon which the IBM software development tools are built [120]. The Eclipse platform has been gaining increased popularity and is by now the preferred integrated development environment (IDE) for many software development companies. The Eclipse Modeling Framework (EMF), built as an Eclipse plug-in, includes a metamodeling language, called Ecore, and a framework to facilitate loading, access, editing, and generation of Java code from Ecore models.

Recent software tools for system level design use Eclipse (for example, BIP, and Metropolis) as a development environment and for the definition of their user interface. The concepts of architectural description not only apply to hardware but to software as well.

2) Hardware Architecture Description: An architecture description is useful especially when providing a model for its "execution" so that the performance and the properties can be dynamically and statically analyzed. Some of the critical questions to answer when analyzing an architecture are: how long does it take this platform instance to compute a particular function? How much power does it consume? How large is its physical footprint? How available are its components? What is the worst case performance? What are the performance variations if we change one or more elements of the platform instance?

Transaction Level Modeling: Currently, the levels of abstraction above RTL are not precisely defined. They are often generically referred to as *transaction level models*. Gajski defines six levels of abstraction (PE-assembly model, bus-arbitration model, time-accurate communication model, and cycle-accurate computation model) within the TLM framework related to each other in terms of accuracy and abstraction power. The SPIRIT (Structure for Packaging, Integrating and Re-using IP within Tool-flows) [187] consortium[7] of EDA vendors and users, has set course to define more precisely what is intended for TLM so that it can be come a standard. TLM 2.0 is the most recent version of the standard. F. Ghenassia of ST Microelectronics, one of the most vocal and competent proposers of this style of design capture and analysis [93] estimates by using measurements on actual ST designs that high-level TLM can provide a factor of three orders of magnitude improvement in terms of simulation speed and of one order of magnitude in modeling effort versus RTL.

Assembly tools: For now, let us assume that we have the models at the level of accuracy we need for the components and that we are interested in the performance of the entire (virtual) platform instance. To answer these questions, a simulation engine is almost always required. For this reason, the most successful approaches are based on SystemC descriptions that facilitate simulation. The library of available components can be fully explicit or encrypted for proprietary IPs. Hence, for the virtual

[5]The tools were developed by Rational Design, a company that was acquired in 2002 by IBM.

[6]Rhapsody was developed by iLogix, acquired by Telelogic in 2006.

[7]The SPIRIT consortium has generated the IEEE P1685 SPIRIT Standardization group whose scope is to define an XML Schema for metadata that documents the characteristics of Intellectual Property (IP) required for the automation of the configuration and integration of IP blocks and to define an Application Programming Interface (API) to make this metadata directly accessible to automation tools.

platform environment, we need a library of models of the available components, a library of models of the components for the interconnections needed to build a platform instance, a set of different "cost" functions, and a simulation environment. The cost functions can be explicit or the result of a computation that may involve models of the components at lower levels of abstraction. Environments for platform description are generally application specific since the models of the components are complex to build and to maintain. Because the concept of platform as defined in the previous sections is based on a limited number of available components, application specific environments can be very effective and allow efficient architecture exploration and analysis.

CoWare Platform Architect [124], Synopsys Cocentric System Studio [192], Mentor Platform Express [52], and ARM MaxSim [141][8] are all addressing the issue of model creation, integration, simulation, and analysis of platform instances created by an assembly of components. Among the academic efforts, MPARM [12], [140] is the most complete approach. The differences among these tools are mostly related to the level of abstraction supported and to the claimed accuracy. It is quite interesting that only a few years ago IC designers would not even consider a model that was not guaranteed to be cycle accurate,[9] while today we are witnessing an increased interest in simulation environment where accuracy is sacrificed to obtain faster simulations. All of these systems are based on SystemC and leverage its simulation capabilities. CoWare ConvergenSC and the Cadence Incisive [122] verification platform are interfaced to provide analysis capabilities including formal verification from TLM to RTL implementations.

Communication-Based Design: To assemble (possibly foreign) IPs, it is important to establish a standard interface so that the modeling work is minimized and IP reuse maximized. There are two approaches to interconnect architectures: one is based on a bus structure to reflect the most commonly used interconnect structure, the other, Network-on-Chip (NoC), is based on a general networking infrastructure and is the focus of a great deal of research, so much so that in all EDA and design conferences there is at least a session dedicated to NoC and a Symposium on NoC [195], following a number of informal NoC workshops such as the one held at DATE [209]. So important is the design of the interconnect infrastructure and of the IP interfaces that an entire design methodology, *communication-based design*, was proposed [179] as part of the GSRC [101] agenda.

The Open Core Protocol (OCP) [157] is a standard that provides a rich modeling framework for its users, centered around bus interconnects. This approach was actually originated by the VSI Alliance (VSIA) formed in 1996 to address the integration challenges of System-on-Chip (SoC). The VSIA now endorses the OCP as a bus-interface standard. The OCP was proposed by Sonics, Inc. [186], [207], a company that specializes in optimized interconnects to ease IP integration for SoCs. IBM with the CoreConnect [121], ARM with the Amba architecture [13], as well as the French start-up Arteris [14] are also offering interconnect IPs. Several IC companies have their own "standard" interconnect IP (see, for example, [191]) that may offer better performance for their particular IP design methodology. The pros and cons of the various approaches to bus-based interconnects coupled with TLM are discussed in [162].

The semiconductor industry has been experiencing a paradigm shift from computation-bound design to *communication-bound design* [43]: the number of transistors that can be reached in a clock cycle, and not those that can be integrated on a chip, is now driving the design process. While local interconnects scale in length approximately in accordance with transistors, global wires do not because they need to span across multiple IP cores to connect distant gates [116]. Consequently, global interconnects have been replacing transistors as the dominant determiner of chip performance; they are not only becoming responsible for a larger fraction of the overall power dissipation but exacerbate also design problems such as noise coupling, routing congestion, and timing closure, thereby imposing primary limits on the design productivity for gigascale integration [151]. In this scenario, researchers have proposed to implement on-chip global communication with packet-switched micro-networks [24], [56], [109]. Based on a regular scalable structure such as a mesh or a torus, an NOC is made of carefully engineered links and represents a shared medium that could provide enough bandwidth to replace many traditional bus-based and/or point-to-point communications. On one hand, NOCs have the potential to mitigate the complexity of system-on-chip design by facilitating the assembling of different IP cores through the emergence of standards for communication protocols and network access points [143], [144]. On the other hand, it is unlikely that one particular network architecture will prevail as a single standard solution. In fact, recent experimental studies [132] suggest that the design of the interconnect network for a multicore architecture must be coupled to the design of the processing cores in a much tighter way than what is typically done, for instance, for computer clusters.

The use of standard interconnects does not completely eliminate the need for verification of the assembly of components as they can interact in unexpected ways unless a strong compositional approach is taken where the properties of the components are inherited by the assembly. As in the case of software components, compositionality is indeed the goal of any approach that aims at improving substantially design time and correctness. Yet in most of the cases, compositionality is either

[8]MaxSim was developed by AXYS that was later acquired by ARM.

[9]This comment is the result of my interactions with designers when discussing Polis [17] (and later VCC [123]) modeling philosophy that were based on less accurate but faster models for software validation.

not true or it may be computationally intractable to verify. This is particularly true when implementation concerns are considered as is typically done when performing design-space exploration.

In the analysis of component-based systems, as in design, it is usually insufficient to consider components in isolation; rather, one must consider each component together with an assumption about the environment in which the component is placed. In other words, a component typically meets its requirements only if the environment does likewise. Such reasoning is inherently circular, and not always sound. For example, the circular ("*assume-guarantee*") reasoning is sound for safety properties: if component *A* does not fail (the *guarantee*) provided *B* does not fail (*the assumption*), and *B* does not fail provided *A* does not fail, then the composition of *A* and *B* will never fail [3], [11], [150]. The same reasoning, however, is unsound for liveness properties: if *A* will respond eventually provided *B* does, and *B* will respond eventually provided *A* does, this does not ensure that the composition of *A* and *B* will ever respond. Little is known about assume-guarantee reasoning for richer properties of the kind we are interested in, such as timing, resource usage, and performance [60], [113]. Indeed, such properties are inherently noncompositional if components are viewed in isolation; for example, if two components share the same resource, then their individual worst case timing properties do not apply to the composition. Therefore, assume-guarantee reasoning, with its emphasis on environment assumptions, is essential for composing physical and computational properties.

Microprocessor Modeling in Architecture Analysis: The large majority of implementation platforms for embedded systems contains one or more software programmable processors since, as we argued earlier, these components are the key to flexibility for embedded system designers. When evaluating a platform, we need to run an application on the platform itself. In the past, embedded system designers would resort to emulation to evaluate their software and to choose their processors. However, emulation has obvious drawbacks when it comes to flexibility and design-space exploration. This leads to much work focusing on virtual platforms that simulate that entire system on one or more general purpose computers. The speed of such models is of tantamount concern, and processor modeling is the main bottleneck to achieving acceptable performance.

Thus, most of the research and industrial attention in architecture modeling has been devoted to the development of the fastest possible processor models for a given accuracy. The models we need vary according to the design problem we are addressing. If we are interested in evaluating a platform where the set of processors is part of the library, then we should not be interested in the internal workings of the processor unless it is necessary for the desired level of accuracy. On the other hand, if part of our problem is to design the architecture or the micro-architecture of the processor (or both), then we need to describe them at a level of abstraction that can give us confidence in our selection.

The approaches that have been used thus far are as follows.

Virtual Processor Model (VPM) [23]: This approach is based on a simple RISC processor model that is used to "emulate" the target processor. The idea is to run the code on the simplified processor whose model parameters are adjusted to mimic as accurately as possible the cycle count and the memory address behavior of the target processor. The source code is then back-annotated with the estimated cycle counts and memory accesses. This approach is very fast since the simplified processor takes little time to execute on the simulation machine but, of course, it may suffer from lack of accuracy. It is ideal when several processor architectures have to be quickly analyzed since adjusting the model parameters does not take too long. This approach does not take into account the optimizations that are done by the target compiler since the approximated model is at the architectural level. It works even if the target processor development chain is not available.

C-Source Back Annotation (CABA) and model calibration via target machine instruction set [23]: This approach also uses an approximate model but the parameters of the model are calibrated using the actual behavior of the code on the target machine. In this case, we do need the software development chain of the target processor. Deriving the model for a given processor requires one to two man months [23].

Interpreted Instruction-Set Simulator (I-ISS): I-ISS are generally available from processor IP providers, often integrating fast cache model and considering target compiler optimizations and real data and code addresses. They are rather slow compared to the previous approaches.

Compiled Code Instruction-Set Simulator (CC-ISS): Compiled-code ISS share all the features of the Interpreted ones but are very fast. Unfortunately, they are often not available from processor IP providers, and they are expensive to create in terms of required skills of the developers. Both require the complete software development chain. This approach had been followed by three interesting start-ups: Axys (later acquired by ARM), Vast, and Virtio (later acquired by Synopsys). All offer cycle accurate compiled-code ISSs. We already mentioned AXYS' MaxSim where the microprocessor model was detailed enough to enable designing its architecture and micro-architecture while it was fast enough to run some application code. Vast Systems [108] developed a modeling approach that produced the fastest models but would not allow the designer to break the microprocessor into its architectural components. In addition, to achieve this speed, models for a

given processor had to be developed by the company. Virtio [202] took an intermediate approach, where the internal structure of the processor could be accessed but not to the level of details offered by MaxSim and the simulation speed would not match Vast's.The AXYS modeling methodology was originated by the work at Aachen of Meyr and Zivojnovic [216] on the LISA language. A company focused on LISA, LisaTex, was founded by H. Meyr and later acquired by CoWare, which also features virtual platform capabilities similar to MaxSim. The importance of LISA resides in the capability of expressing models of microprocessors and DSPs at different layers of abstraction and of generating automatically the entire software development suite (compiler, linker, assembler, and debugger) from the model.

Worst Case Execution Time Estimation: In addition to simulation, the performance of a program running on a microprocessor or DSP can be estimated statically, i.e., performing a path analysis on the program execution with timing annotation, similar to the way static timing analysis is performed in hardware design. As in timing analysis, we are interested in worst case execution times which are in general very difficult and even impossible to compute, hence the necessity to develop methods for WCET estimation that are fast to compute and safe, i.e., are upper bounds on the real execution times. Of course, if the bounds are loose, then the static analysis would be almost useless. Unfortunately, the micro-architecture of modern microprocessors is so complex with caches, pipelines, pre-fetch queues, and branch prediction units that accurately predicting their effects in execution time is a real challenge.

Several techniques for WCET estimate have been proposed. The most mathematically elegant technique relies on *abstract interpretation* [53], a theory of sound approximation of the semantics of computer programs, based on monotonic functions over ordered sets. It can be viewed as a partial execution of a computer program that gains information about its semantics (e.g., control structure, flow of information) without performing all the calculations. Path-analysis including cache and pipeline prediction was solved by Malik and students at Princeton [138] using integer linear programming. Abstract interpretation was also the foundation of the work of Wilhelm and colleagues at the University of the Saarland, which is the most comprehensive and effective body of work in this area. The Wilhelm paper [89], [106] is an excellent review of WCET estimation techniques. Their work is also the basis for AbsInt [5], a company that offers tools related to advanced compiler technology for embedded systems.

C. Mapping

The problem of mapping a functional description to a platform instance (architecture) is the core of the PBD design-by-refinement paradigm. I recall that the PBD process begins with a description of the functionality that the system must implement, a set of constraints that must be satisfied, and a library (platform) of architectural components that the designer can use to implement the functionality. The functionality specifies what the system does by using a set of services. The architectural platform captures the cost of the same set of services. *Mapping* binds the services used with the services offered and selects the components to be used in the implementation and the assignment of functionality to each component.

The problem, in general, is the mismatch between the models of computation of the functionality and the one of the implementation. For example, when a functional design expressed with a synchronous language is mapped to an architecture that is asynchronous, the behavior of the design may be substantially changed. If we wish to maintain a formal relationship between mapped design and the original functional design, we need to guarantee that the mapped design does not introduce behaviors that are incompatible with the assumptions made at the functional level. Most of the approaches followed today try to maintain the same models of computation for functionality and platform instance but this may overly restrict the design space. For example, synchronous languages mostly map to synchronous implementation platforms.

If we allow more general implementation architectures, since mapping is mostly done manually at higher levels of abstraction for the lack of a precise mathematical formulation of the problem or of an efficient automatic procedure, there is a nonnegligible chance that the final implementation will not behave as expected. Very active research is on the mapping of synchronous descriptions into *functionally equivalent* asynchronous implementations (see e.g., [4]).

The most traditional problem to be faced is mapping a concurrent functional model into a sequential implementation platform, for example, when we map a set of concurrent processes to a single processor. In this case, concurrent tasks must be scheduled for execution. If there are timing constraints on the execution of the concurrent processes, then we are faced with an interesting mathematical problem with great practical relevance: scheduling.

1) Scheduling: The literature on scheduling and schedulability analysis for real-time systems is *very* rich and it will take many pages to do justice to it. Here, I summarize the main points and their relevance to the problem of embedded system design. References [38] and [39] are excellent books on the topic.

The RT scheduling problem can be classified along three axes:

1) *Input characteristic*: time-driven: continuous (synchronous) inputs; event-driven: discontinuous (asynchronous) inputs;

2) *Criticality of timing constraints*: Hard RT systems: response of the system within the timing constraints is crucial for correct behavior; Soft RT systems: response of the system within the timing constraints increases the value of the system;
3) *Nature of the RT load*: Static: predefined, constant and deterministic load; Dynamic: variable (nondeterministic) load.

The load is represented by a set of processes called tasks. Tasks are given priorities to indicate their relative importance. The rules according to which tasks are executed at any time on the limited resource is called a scheduling algorithm. Dynamic scheduling algorithms compute schedules at *run time* based on tasks that are executing. Static scheduling algorithms determine schedules at *compile time* for all possible tasks.

In general, dynamic scheduling yields better processor utilization and overall performance at the cost of longer run time used to decide the schedule and of lack of determinism if tasks are event triggered. Preemptive scheduling permits one task to *preempt* another one of lower priority.

Real-time scheduling theory [38] provides a number of "schedulability tests" for proving whether a set of concurrent tasks will always meet its deadlines or not. Major improvements have been made to scheduling theory in recent years. The original classic Rate Monotonic Analysis [139] and the newer Deadline Monotonic Analysis [84] have both been absorbed into the general theory of fixed-priority scheduling [83].

Real-time scheduling theories have historically found limited success in industry. However, there is a renewed interest in scheduling theory as real-time embedded software systems experienced serious problems due to scheduling failures (see for example the Mars PathFinder malfunctioning due to a problem of priority inversion in its scheduling policy). Ways of transferring scheduling theory from academia to industrial practice have been investigated for more than ten years in [78].

2) Correct-by-Construction Mapping—Giotto: The idea is to avoid solving the scheduling problem by forcing the models of computation of functionality and architecture to match. The time-triggered architecture [131] offers a view of an implementation platform that matches the view of synchronous languages. Giotto [94], [110] is a bridge towards the PBD paradigm and the more traditional embedded software approaches. Giotto's view of the typical activities of a software engineer includes decomposing the necessary computational activities obtained by the application engineer who determines the functionality of the design and captures it at a high level of abstraction with an informal mathematical description or with algorithmic languages, into periodic tasks, assigning tasks to CPUs and setting task priorities to meet the desired hard real-time constraints under the given scheduling mechanism and hardware performance. The software engineer has final authority over putting the implementation together through an often iterative process of code integration, testing, and optimization. Giotto [94], [110] provides an intermediate level of abstraction, which:

1) permits the software engineer to communicate more effectively with the control engineer;
2) keeps the implementation and its properties more closely aligned with the mathematical model of the control design.

Specifically, Giotto defines a software architecture of the implementation which specifies its functionality and timing. Functionality and timing are sufficient and necessary for ensuring that the implementation is consistent with the mathematical model. On the other hand, Giotto abstracts away from the realization of the software architecture on a specific platform and frees the software engineer from worrying about issues such as hardware performance and scheduling mechanism while communicating with the control engineer. After writing, a Giotto provides an abstract programmer's model for the implementation of embedded control systems with hard real-time constraints.

A typical control application consists of periodic software tasks together with a mode-switching logic for enabling and disabling tasks. Giotto specifies time-triggered sensor readings, task invocations, actuator updates, and mode switches independent of any implementation platform. In this respect, Giotto is related to synchronous languages since it limits the semantics of the designs to the synchronous assumption.

Giotto can be annotated with platform constraints such as task-to-host mappings and task and communication schedules. The annotations are directives for the Giotto compiler, but they do not alter the functionality and timing of a Giotto program.

The Giotto compiler is based on the *virtual machine concept*. In general-purpose computing, the virtual machine concept has been effective at permitting software developers to focus on problem-level design rather than platform-specific idiosyncrasies. The Giotto E-machine is a virtual machine that abstracts away from the idiosyncrasies of the RTOS, shifting the burden of ensuring time correctness from the programmer to the compiler.

By separating the platform-independent from the platform-dependent concerns, Giotto enables a great deal of flexibility in choosing control platforms as well as a great deal of automation in the validation and synthesis of control software. The time-triggered nature of Giotto achieves timing predictability, which makes Giotto particularly suitable for safety-critical applications.

3) Automatic Mapping With Heterogeneous Domains: As the PBD methodology solidifies and design environments such as Metropolis [21] are built to support it, there is a clear need for automatic optimized mapping processes to

increase productivity and quality. To automate the mapping process, we need to formulate the optimization problem in rigorous mathematical terms.

This is an on-going research topic that can be framed in the general algebraic theory of heterogeneous models of computation [163]. We are taking inspiration from the classic logic synthesis flow [33] viewed as an instance of the general PBD methodology. In this flow, the behavioral portion of the design is captured in terms of Boolean equations. The architecture is represented by a gate library which contains different types of logical gates. The mapping process selects a set of gates from the gate library such that the functionality remains the same. To optimize the gate selection process, both the Boolean equations and the gate library are transformed into netlists consisting of a primitive logical gate, such as a two-input NAND gate (NAND2). The mapping stage, known as technology mapping, is then reduced to a covering problem. Each covering has a certain cost, and the synthesis process reduces the overall cost according to metrics such as delay or power consumption.

This synthesis flow is based on a common primitive—the NAND2 gate—and mathematical rules for defining the behavior of a set of interconnected primitives—Boolean logic. Boolean logic is appropriate for synthesizing combinational logic between registers in a synchronous hardware design. Hence, the flow can be restated in three steps: restriction of the functional domain to synchronous circuits, choosing a common mathematical representation, and representing the functionality and platform library in terms of primitives.

In this research, we use these same three aspects at the system level. The first problem to solve is to find a common mathematical language between functionality and platform. The selection of a common mathematical language depends on critical decisions involving expressiveness, ease of use, and manipulation. In selecting the common mathematical domain, we must realize that some of the properties of the original models of computation are inevitably lost. What to leave on the table is of paramount importance and cannot be done automatically. The second step is to identify primitive elements in the common language that can be used to perform optimal mapping. The selection of the primitive elements to use involves a tradeoff between granularity and optimality: coarser granularity allows the use of exhaustive search techniques that may guarantee optimality, while finer granularity allows the possibility of exploring, albeit nonoptimally, a much larger search space.

This research is still in its infancy, but I believe it will have an important role in making PBD a mainstay of system level design.

D. Final Remarks

The approaches in this section all fit in the PBD scheme addressing the various components of the methodology: functionality and architecture capture and analysis, refinement to implementation. The PBD methodology can be implemented "manually" by linking together the various tools and models presented above to describe functionality and architectural libraries and to perform mapping to go down one level in the refinement chain. This approach has been followed in a number of industrial experiments. In particular, Magneti-Marelli Powertrain with the help of PARADES formed a proprietary PBD flow out of existing industrial tools achieving significant savings in design effort and excellent quality improvements [80]. The tools used and their relationships to PBD are shown in Fig. 6 (courtesy of A. Ferrari, PARADES). For the capture of the functionality of the design (control algorithms for power train management) Magneti-Marelli engineers used either Simulink/StateFlow or ASCET-SD by ETAS, a Bosch company providing tools for automotive design. The architecture of the implementation was captured at PARADES using VCC and UML/ADL. Mapping was performed manually via textual binding of the functionality to the architecture and with VCC. The RTDruid tool-set, developed by the Scuola Superiore di Sant'Anna, Pisa, in cooperation with Magneti Marelli Powertrain, was used to validate the scheduling properties of automotive real-time applications. The final implementation was carried out generating code with TargetLink that supported the OSEK/VDX operating system configured manually with OIL to complete the implementation of the functionality on the chosen architecture.

The benefits of PBD can be amplified if the "glue" that allows us to pick the flow that best fits a particular application is automatically generated. The glue should serve as an integration framework not only for the design flow and for the supply chain but also for the various application-specific languages, tools, and methods. This underlines the need for a different kind of framework.

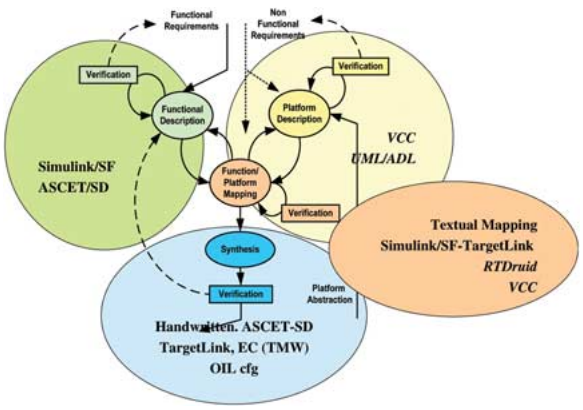


Fig. 6. ***PBD in Magneti-Marelli.***

V. METROPOLIS FRAMEWORK

Metropolis [19], [21] was designed with the requirements of Section II in mind. In its design, there is a determination to learn from previous mistakes and to include the best ideas that were previously developed. In addition, Metropolis design representation has foundations that can be shared across different models of computation and different layers of abstraction. Architectures are also represented as computation and communication "services" to the functionality. Metropolis can then analyze statically *and* dynamically *functional designs* with models that have no notion of physical quantities and *mapped designs* where the association of functionality to architectural services allows us to evaluate the characteristics (such as latency, throughput, power, and energy) of an implementation of a particular functionality with a particular platform instance. The mechanism used to derive the performance of the overall design given an estimate of the quantities of the components of the design. These are kept separate because they represent the result of implementation choices and as such, they derive from a specific architectural mapping of the behavior. Metropolis also allows us to enter constraints separately from functionality thus providing a mixed denotational and operational specification. Constraints are expressed as equalities and inequalities over the performance indexes and as formulas in a temporal logic.

A. Introduction to Metropolis

Metropolis[10] has been designed to support platform-based design in a unified framework. However, given the generality of the framework, it could be used to support other design approaches. A schematic representation of the Metropolis framework is shown in Fig. 7: the infrastructure consists of:

1) an internal representation mechanism, called the *Metropolis Meta-Model (MMM)*;
2) design methodology;
3) base tools for simulation and design imports.

The MMM is a model with formal semantics that:

1) is powerful enough, so that the most commonly used abstract models of computation (MOCs) and concrete formal languages could be translated into it, so as not to constrain the application designer to a specific language choice[11];
2) can be used to capture and analyze the desired functionality as well as to describe an architecture and the associated mapping of the functionality onto the architectural elements.

[10]Metropolis is a public domain package that can be downloaded from http://embedded.eecs.berkeley.edu/metropolis/forum/2.html.

[11]At least in principle. Of course, time may limit in practice the number of available translators.

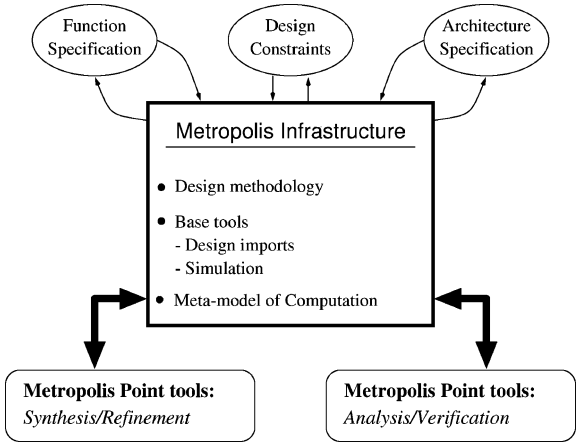


Fig. 7. ***Metropolis framework.***

Since the model has formal semantics, it could be used to support a number of synthesis and formal analysis techniques in addition to simulation.

The meta-model uses different classes of objects to represent a design.

1) Processes, communication media, and netlists describe the functional requirements, in terms of input–output relations first and of more detailed algorithms later.
2) A mechanism of refinement allows one to replace processes and media with subnetlists representing more detailed behavior.
3) Temporal and propositional logic can be used to describe constraints over quantities such as time or energy [20].

Functional Model: A functional model in the Metropolis Metamodel is represented using a functional netlist. A functional netlist consists of a network of processes, linked using media. A process communicates using ports, which have associated interfaces. Media that implement these interfaces can be connected to ports. The behavior of the network is described as a set of executions, which themselves consist of a sequence of events. Events are atomic and can be associated with the starting and ending points of actions.

The functional model utilizes services, which are just methods bundled into interfaces, to carry out computation and communication. These services can be thought of as system calls. The functional model also specifies in which order services are utilized. However, it does not represent the concurrency with which these services are carried out, nor does it specify the cost of utilizing these services. To obtain this additional information, we turn to the architectural model.

Architectural Model: An architectural model is realized using an architectural netlist in the Metropolis Metamodel. An architectural netlist is an interconnection

of computational and communication components characterized by some services and costs, i.e., quantities. An architectural model provides services to the functional model by defining mapping processes. A mapping process is similar to a device driver in that it sits between the hardware platform and the application. The services in an architecture are also annotated with costs by using quantity managers and their associated algorithms.

In Metropolis, an architecture model is in general divided into two netlists: a scheduled netlist and a scheduling netlist. The scheduled netlist contains all the architectural components, while the scheduling netlist contains the quantity managers [22]. Each process and medium in the scheduled netlist has a state-medium (e.g., CPU, bus, memory) associated with it. These state-media belong to the scheduling netlist. The scheduling netlist also contains all the quantity managers.

The distinction between the scheduled netlist and scheduling netlist is based on the implementation of a scheduling interface. Each component in the architecture has some resources or physical quantities that characterize the services it can provide, i.e., what it can do, and how much it will cost. For instance, the CPU clock cycle is a quantity associated with the CPU component. Normally, the scheduling netlist will implement two methods: *resolve()* and *postcond()*, which we use to model the scheduling of the resource or physical quantity the corresponding quantity manager will perform.

Mapping: In Metropolis, the architecture provides certain services at a particular cost and concurrency while the functional model utilizes these services. Mapping adds explicit synchronization constraints to coordinate the interface functions that characterize these services. In this manner, it eliminates some of the nondeterminism present in the functional and architectural models by intersecting their possible behaviors. After mapping, a *mapped implementation* is created.

Mapping involves the creation of a new mapping netlist. This netlist instantiates both the functional and architectural netlists without modification. Then, synchronization constraints are added to the mapping netlist that coordinate the beginning and end events of the interface function calls. These constraints force the functional model to inherit the concurrency and latency defined by the architecture while forcing the architectural model to inherit the sequence of calls specified for each functional process.

Constraints: Constraints:

1) initially describe constraints over yet undefined quantities (e.g., latency constraints when describing processes at the untimed functional level using Kahn process networks);
2) then specify properties that must be satisfied by more detailed implementations (e.g., while a Kahn process network is being scheduled on a processor);
3) finally, define assumptions that must be checked for validity (e.g., using a cycle-accurate instruction set simulator to run the scheduled processes in software).

Constraints allow different types of formal properties to be either specified or checked. Linear temporal logic (LTL) [184] constraints restrict the possible behavior of the system during execution. Behaviors which violate the constraint are not legal behaviors of the system. In most cases, to specify a set of properties is much easier than to implement a piece of code with those properties. Synchronization constraints used in mapping are an example of LTL constraints.

Logic of constraints (LOC) is the second way in which constraints can be represented in the Metropolis Metamodel [212]. LOC constraints are checked during an execution trace, they are not enforced. Metropolis currently has several backends that allow the checking of traces both online and offline. Since only the traces are checked, not the system itself, LOC checking is relatively cheap. Full-blown model checking makes much stronger claims about system properties, but its PSPACE complexity rules out evaluating industrial-size designs.

B. Tool Support

The Metropolis framework contains a front end that parses the input metamodel language and creates an abstract syntax tree. Then, the abstract syntax tree can be passed to different back-end tools for analysis. One of the most important back-end tools is the simulator [211], which preserves the metamodel semantics while translating a metamodel specification into the executable SystemC language. LTL and a set of built-in LOC constraints can be enforced during the simulation [212]. Also, based on simulation traces, the LOC checker [49] can examine whether the system satisfies the LOC properties. Another formal verification back-end uses SPIN [117] to verify LTL and a subset of LOC properties against a metamodel specification [49]. The refinement verification tool [64] compares two models based on event traces and determines whether a refinement relationship holds between them. The synthesis back-end tool is based on UCLA xPilot [1] system, and it works on a synthesizable subset of the metamodel. All the front-end and back-end tools can be invoked interactively by using the Metropolis Interactive Shell.

Remarks on Metropolis Use: Metropolis is not aimed at providing algorithms and tools for all possible design activities. Instead, it offers syntactic and semantic mechanisms to store and communicate all the relevant design information, and it can be used to "plug in" the required algorithms for a given application domain or design flow. This unified mechanism makes it easy to incorporate external tools and thus addresses the problem of design chain integration by providing a common semantic infrastructure.

Another fundamental aspect that we considered throughout the design of Metropolis is the ability to *specify* rather than implement, *execute* reasonably detailed but still fairly abstract specifications, and finally *use the best synthesis algorithms* for a given application domain and implementation architecture. For these reasons, we represented explicitly the concurrency available at the specification level, in the form of multiple communicating processes. We used an executable representation for the computation processes and the communication media, in order to allow both simulation and formal and semiformal verification techniques to be used. Finally, we restricted that representation, with respect to a full-fledged programming language such as C, C++ or Java, to improve both *analyzability* and *synthesizability*.

Unfortunately, we were not able to fully remain within the decidable domain (or, maybe even worse, within the efficiently analyzable domain), because this goal is achievable only with extreme abstraction (i.e., only at the very beginning of a design), or for very simple applications, or in very specialized domains. However, we defined at least a meta-model for which the task of identifying *sufficient conditions* that allow analysis or synthesis algorithms to be applied or speeded up, could be simple enough. Hence, our meta-model can be translated relatively easily, under checkable conditions or using formally defined abstractions, into a variety of abstract synthesis- or analysis-oriented models, such as Petri nets, or static dataflow networks, or synchronous extended finite-state machines.

C. Related Work

Metropolis is the result of the evolution of our work on system-level design that began in the late 1980s. Metropolis is strongly related to the Polis System [17] of Berkeley and to the VCC of Cadence Design Systems. The Artemis Work-bench [166], [167], Mescal [79], [149], and CoFluent Studio are the most related design systems as far as methodology and tool flow are concerned.

The most used system level design flows in industry today are based on Simulink. These flows are mostly related to design capture and rapid prototyping even though they can also be used to generate efficient code for some industrial domain of application. They are not geared towards architecture design space exploration.

The Polis, VCC, Artemis, Mescal, and CoFluent Studio approaches are the closest to the ideal of a framework for full support of PBD but none addresses all the requirements that we set forth in the previous sections. In particular, the separation of function and architecture, the capability of representing in the same framework both function and architecture as well as different layers of abstraction, of handling heterogeneous models of computation, of mixing denotational and operational specifications, of representing the design process as successive refinement via mapping of function to architecture are not all present at the same time.

1) Polis System: The Polis System was developed in the early 1990s at Berkeley [17]. It focused on a particular application domain, automotive, since the industrial supporters of the approach were mostly in the automotive supply chain, from car manufacturers (BMW and Mercedes) to Tier 1 suppliers (Magneti-Marelli) and Tier 2 suppliers (ST). While developing Polis, the foundations for PBD were laid as the separation of concern principle was set for architecture and function, communication, and computation.

Polis supported *one model of computation*, Co-design finite-state machines (CFSMs), a Globally Asynchronous Locally Synchronous (GALS) model where concurrent FSMs communicated via asynchronous channels with size 1 buffers. The architecture supported were based on a single microprocessor and peripherals including FPGAs and other custom hardware blocks.

The supported tools were simulation, architectural exploration with accurate and very fast code execution time evaluation. This was possible because the software was automatically generated from the CFSMs model. The code generation technique was novel as its optimization approach was based on restructuring of the model as it was done in hardware logic synthesis [18]. Esterel compilers were built based on the same principle as the synchronously communicating finite-state machines, corresponding to the semantics of the synchronous languages were mapped into C instructions using MIS [33], a logic synthesis tool. The estimation was based on a simplified model of the architecture of the processor similar to the VPM method presented above [194].

The hardware–software interfaces were automatically generated and the software tasks scheduled according to a scheduling policy decided by the user. The design was partitioned in software and hardware parts by assigning CFSMs to the microprocessor to execute or to the hardware blocks. The back end was targeted for an Aptix rapid prototyping board that was used by some of the industrial partners to verify the control algorithms on the car early in the design cycle.

Polis drawbacks were in the limitations of the targeted architecture and of the model of computation used that were constraining the application domain. However, it did have several strengths as the separation of concerns in the supported methodology allowed some of the industrial partners to eliminate errors, shorten design time, and reduce the number of embedded software designers maintaining intact the overall productivity of the team.

2) VCC: The VCC of Cadence Design Systems [145] (no longer commercially available) inherited several of the Polis principles including the methodology, the separation of concerns, the processor modeling approach for execution time estimation, and the model of computation, but it took them to another level. In particular, the idea of architecture as service to the functionality via the

architecture service concept and the communication refinement mechanisms were real advances to bring the basic concepts of PBD and design-space exploration to the forefront.

However, VCC shared also some of the drawbacks of Polis: it used a fixed computation model and an architecture can be constructed from a predefined set of components, but it cannot handle a recursive layering of platform models. In addition, VCC used C and C++ to define the behavior of processes, which ruled out formal process analysis or optimization techniques beyond those that standard software compilers use.

The VisualSim [65] offered by the start-up Mirabilis Design resembles in many respects VCC and promises a similar functionality.

3) Artemis Workbench: The Artemis workbench [166], [167] was designed in the early 2000s following a design methodology that espoused the PBD paradigm in a similar way to Polis as it emphasized one particular domain of application, multimedia, and it exploited the characteristics of the domain to determine the model of computation and the supported architecture.

The supported design flow is patterned after Fig. 5 and begins with a Simulink representation of the functionality of the design that is converted in a language based on a mathematically rigorous model, Kahn Process Networks. This translation is carried out with the Compaan tool [190]. The architecture is based on a set of virtual processors. This is also captured using Kahn Process Networks so that functionality can be mapped easily to the architecture [190]. The mapping generates actual VHDL code that can be synthesized to obtain an FPGA implementation. The workbench includes modeling and verification tools collected in Sesame [77].

4) Mescal: The Modern Embedded Systems, Compilers, Architectures, and Languages (Mescal) project introduced a disciplined approach to the production of reusable architectural platforms that can be easily programmed to handle various applications within a domain. In particular, Mescal's focus is on thorough design-space exploration for network-processing applications.

The targeted architectures are application-specific instruction processor networks. Mescal allows the representation of behavior via Ptolemy II. The architecture description allows multiple levels of abstraction and mapping is performed from behavior to architecture generating code for the ASIPs. The mapping process consists of matching components of the behavior to the ASIPs and communication among the components of the behavior to the communication resources in the architecture.

5) CoFluent Studio: The environment supports the MCSE methodology (Méthodologie de Conception des Systémes Electroniques, also known as CoMES—Co-design Methodology for Electronic Systems) [40] that is based on a top-down design process. At the functional level, CoMES supports Model Driven Design. Going towards implementation, it follows the mapping of function to architecture paradigm in the Y-chart organization that was also a cornerstone of Polis, VCC, Artemis, Mescal, and Metropolis. As for the other tools, there is a dual use of the approach: one for the system developer to capture his requirements and functional specifications and one for the architecture designer to to develop multi-board, single board, or SoC architectures to support the functionality of the system. The system targets multimedia applications in the consumer and telecommunication domains.

6) Simulink-Based Flows: The role of Simulink (and Stateflow) in advancing industrial model-based design cannot be overemphasized. In several industrial domains, Simulink has become a *de facto* standard. I argued that one of the strengths of model-based design is the transformation aspects and the corresponding potential of automatic generation of production code. Simulink semantics is based on its execution engine. However, it is possible to generate automatically efficient code. Two tool sets have been used in industry for this purpose.

1) Real-Time Workshop (RTW) of the Mathworks [148]: Real-Time Workshop generates and executes stand-alone C code for developing and testing algorithms modeled in Simulink. The generated code using Simulink blocks and built-in analysis capabilities can be interactively tuned and monitored, or run and interact with code outside the MATLAB and Simulink environment. In addition to standard compiler optimization techniques such as code reuse, expression folding, dead path elimination, and parameters inlining that are used to optimize code generation, RTW offers
   a) *Single Tasking*: In single-tasking mode, a simple scheduler invokes the generated code as a single thread of execution, preventing preemption between rates.
   b) *Multitasking*: In multitasking mode, a deterministic rate monotonic scheduler invokes the generated code, enabling preemption between rates. In a bare-board environment, you preempt the code with nested interrupts. In an RTOS environment, you use task priorities and task preemption.
   c) *Asynchronous*: In asynchronous mode, nonperiodic, or asynchronous, rates are specified using Simulink S-functions. Real-Time Workshop translates these rates into target-specific code based on the execution environment. Code for events, such as hardware

interrupts, are modeled and generated. Subsystems can be triggered as independent tasks. An asynchronous block library is provided for the VxWorks RTOS offered by WindRiver [206] that can be used as a template for creating a library appropriate for a particular environment.

2) TargetLink of dSpace [71]: TargetLink's code generation strength lies in its target awareness. Target awareness means generating code for a specific compiler/processor combination and running it on an evaluation board during simulation. Most processors for embedded applications, especially in the automotive field, are supported by TargetLink.

Simulink and Stateflow can also be transformed in Verilog and VHDL to generate hardware. However, optimization in this case is still lacking and this feature is mainly useful to generate quickly an input for an FPGA prototype implementation.

D. Design Examples Using Metropolis

In this section, the applications of Metropolis and of PBD to two relevant industrial examples are presented. The examples have been selected to express the generality of the approach and come from two of the industrial sponsors of this approach: Intel and General Motors. We are fortunate to have a number of other interested industrial sponsors such as United Technologies Corporation, Pirelli, Telecom Italia, ST, and Cadence who have supported this activity with funding and design examples.

The first example is about mapping an application to a heterogeneous single-chip computing platform. The second is about the design of a distributed architecture for supervisory control in a car. These two examples cannot expose all potential use cases of Metropolis. In particular, the goal of the exercise is to show how the paradigms exposed above can be applied at the micro system level as well as to the macro system level to perform architectural design exploration and design analysis. The examples do not address the use of formal analysis nor of the support for the design chain. However, the fact that the two examples are so different in terms of the level of abstraction should demonstrate that the supply chain could be fairly easily accommodated.

1) JPEG Encoder Design: A JPEG encoder [203] is required in many types of multimedia systems, from digital cameras to high-end scanners. A JPEG encoder compresses raw image data and emits a compressed bitstream. A block diagram of this application is shown in Fig. 8.

The goal of the design exercise is to map this algorithm efficiently to a heterogeneous architecture, the Intel MXP5800. For more details on the design see [58].

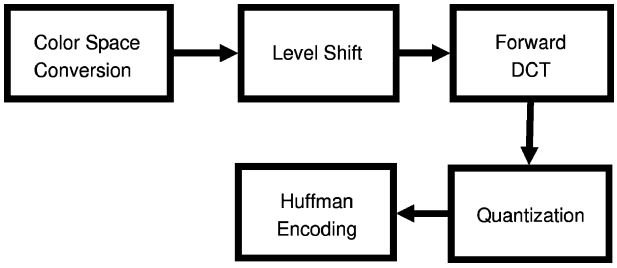


Fig. 8. *JPEG encoder block diagram.*

Intel MXP5800 Platform: The Intel MXP5800 digital media processor is a heterogeneous, programmable processor optimized for document image processing applications. It implements a data-driven, shared register architecture with a 16-bit data path and a core running frequency of 266 MHz. The MXP5800 provides specialized hardware to accelerate frequently repeated image processing functions along with a large number of customized programmable processing elements.

The basic MXP5800 architecture, shown in Fig. 9, consists of eight Image Signal Processors (ISP_1 to ISP_8) connected with a toroidal mesh.

Each ISP consists of five programmable processing elements (PEs), instruction/data memory, 16 16-bit general purpose registers (GPRs) for passing data between PEs, and up to two hardware accelerators for key image processing functions. The major characteristic of this architecture platform is the extremely high degree of parallelism and heterogeneity. Harnessing the flexibility of the PEs to extract high performance is the main design challenge.

Modeling and Design Space Exploration: Starting from both a sequential C++ implementation and the concurrent assembly language implementation provided in the Intel MXP5800 development kit, we assembled an architecture-independent model of the JPEG encoder in Metropolis expressed in a statically schedulable dataflow model. A total of 20 FIFO channels and 18 separate processes are used in the application model. Characteristics of the original C++, assembly, and Metamodel designs are provided in Table 1.

The MXP5800 architecture platform can be modeled in Metropolis by using processes, media, and quantity managers in the Metropolis Metamodel. A single ISP is modeled as shown in Fig. 10. The rectangles in the diagram represent tasks, the ovals represent media, while the diamonds are the quantity managers.

To model running time, a global time quantity manager is used. Every PE, every global register, and the local memory are connected to it. Both computation and communication costs can be modeled by sending requests to this global time quantity manager and obtaining time annotations.

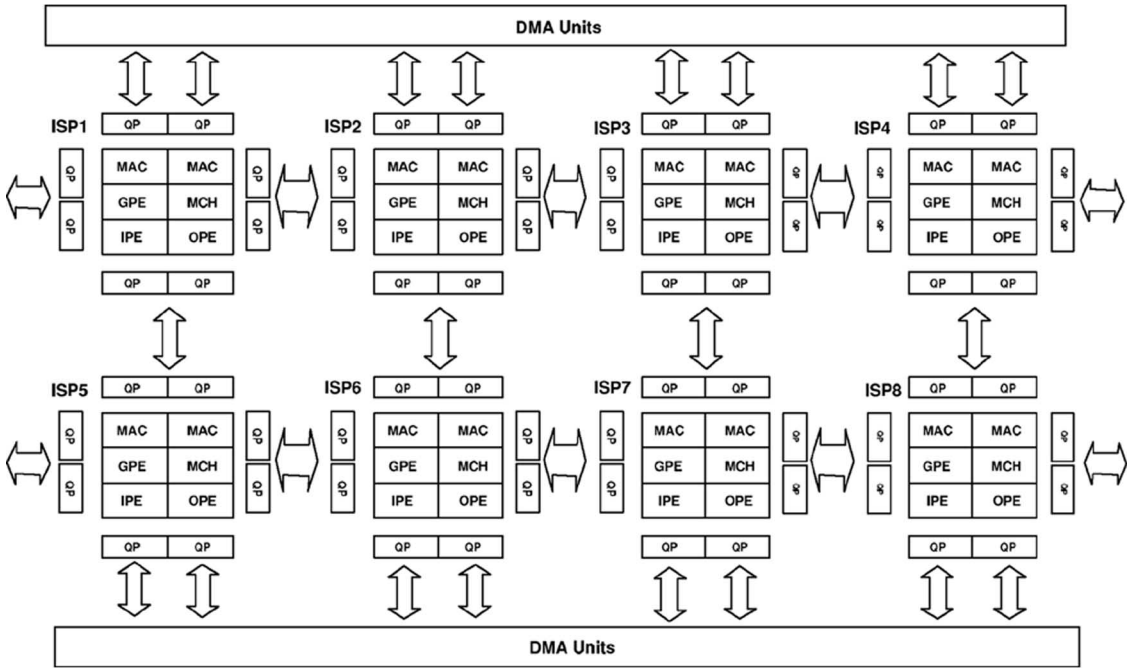


Fig. 9. ***Block diagram of MXP5800.***

Design Space Exploration and Results: Given the application and architectural models in Metropolis, the design space can be explored by attempting different mappings. Each mapping scenario is specified in Metropolis with two types of information. The first is a specific set of synchronization constraints between the events in both models corresponding to the services that constitute the MoC. Along with these events—which represent the read, write, and execution services defined in our MoC—the parameters such as register location or local memory address can also be configured. The second is the set of schedules for the PEs that determine the execution order between the tasks. Both of these are relatively compact, meaning that new mapping scenarios can be created quickly and without modifying either the application or the architectural models.

The application is a total of 2695 lines as shown previously in Table 1. The architectural model is 2804 lines, while the mapping code is 633 lines. Each additional mapping scenario can be described with approximately 100 lines of additional code and without modifying any of the other code.

To show the fidelity of our modeling methodology and mapping framework, we initially abstracted two mapping scenarios from the implementations provided in Intel MXP5800 algorithm library and carried out simulation in the Metropolis environment. We also tried an additional two scenarios which did not have a corresponding assembly language implementation. For all of the scenarios, only the mapping of the fine granularity 1D-DCT processes was varied.

For each scenario, the number of clock cycles required to encode an 8×8 sub-block of a test image was recorded through simulation in Metropolis. For the first two scenarios, implementations from the MXP5800 library are available and were compared by running the code on a development board. The results are shown in Fig. 11. The cycle counts reported with the Metropolis simulation are approximately 1% higher than the actual implementation since we did not support the entire instruction set for the

Table 1 JPEG Encoder Models

Implementation	Language	Concurrency	Lines of Code
IJG	C++	Sequential	18,000
MXP5800 Library	ASM	Concurrent	915
Metropolis	MMM	Concurrent	2,695

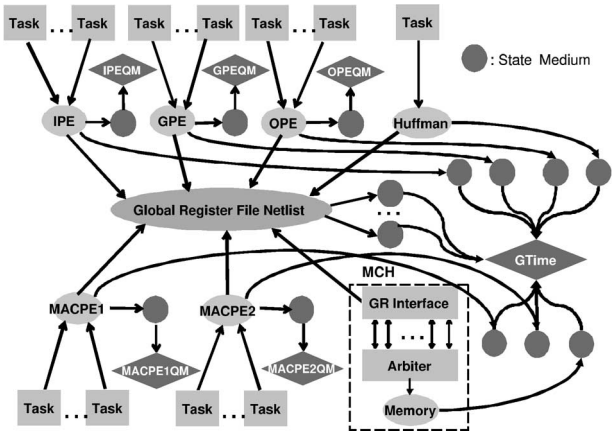


Fig. 10. *MXP5800 ISP modeling in metropolis.*

processing elements. The latter two scenarios provide reasonable relative performance, but assembly implementations were not available for comparison.

2) Distributed Automotive Design: The distributed system under consideration is a limited-by-wire system that implements a supervisory control layer over the steering, braking, and suspension system. The objective is to integrate active vehicle control subsystems to provide enhanced riding and vehicle performance capabilities.

The high-level view of the functional architecture of this control system is defined in Fig. 12. Using sensors to collect data on the environment, the supervisor plays a command augmentation role on braking, suspension and steering. This supervisory two-tier control architecture enables a flexible and scalable design where new chassis control features could be easily added into the system by only changing the supervisory logic.

The goal of the design exercise is to show how a virtual integration and exploration environment based upon formal semantics (Metropolis) can be effectively used to uncover design errors and correct them. In particular, the error discovered was a priority inversion issue in the present implementation of the software and hardware stack for the supervisory control. This problem was addressed by exploring different CAN Controller hardware setups. The Metropolis simulation environment enabled us to trade off buffer size and access policy with the amount of latency incurred by the different signals. For more details on this design see [215].

Automotive CAN Platform: For this case study, the typical baseline physical-architecture consists of 6 ECUs including two CAN-based smart sensors connected over a high-speed CAN communication bus, one ECU per subsystem in Fig. 12. Within each ECU, the platform instance consists of software tasks, a Middleware, a CPU/RTOS, an interrupt handler, a CAN Driver, and a CAN Controller, as shown in Fig. 13.

The middleware layer handles the transfer of data between tasks. The CPU/RTOS implements a priority-based preemptive scheduling algorithm as specified in the OSEK specification for BCC1 type tasks [100]. The CAN Driver transfers the messages between the middleware and the CAN Controller. The CAN Controller implements the CAN communication protocol as specified in [41].

Tradeoff Analysis: The problem considered here is the priority inversion problem [181] that may occur between messages of different priority originating from the same ECU, when transmit buffers are shared among them and the message holding the buffer cannot be aborted. In the fairly common case, a single transmit buffer is shared in mutual exclusion by all transmitted messages from the same ECU.

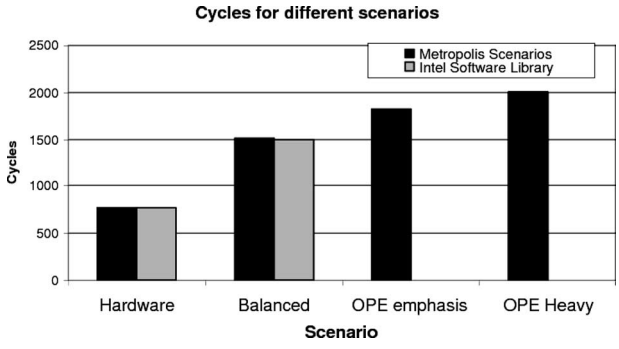


Fig. 11. *Performance comparisons.*

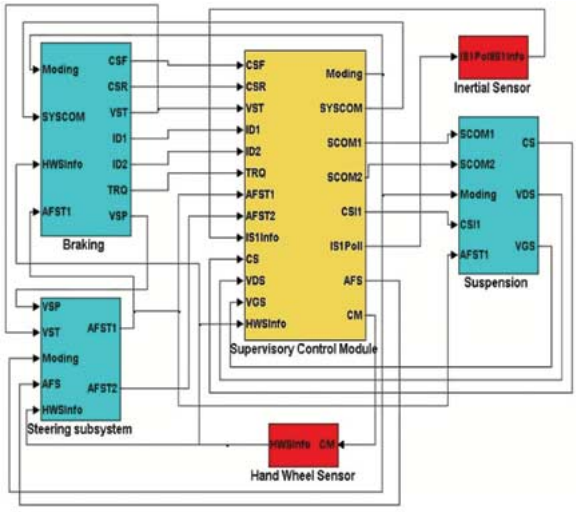


Fig. 12. *Functional architecture of supervisory control system.*

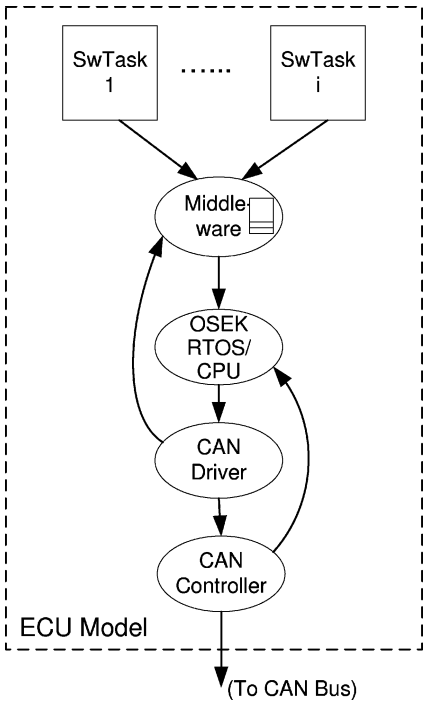


Fig. 13. *ECU model.*

We analyzed three different configurations of the CAN Controller and CAN Driver in the Supervisor module. In this ECU there are 11 messages to be transmitted in total, and five of them have message ID $\leq 0 \times 400$:

1) one transmit buffer, shared;
2) two transmit buffers, shared;
3) two transmit buffers, one of which is shared and the other is dedicated to high priority messages with message ID $\leq 0 \times 400$.

In the third configuration, the first buffer can transmit any message, including high-priority ones, but the second buffer cannot transmit the six lowest priority messages from the Supervisor.

The architectural model was annotated with the following numbers:

1) The ECU system clock operates at 40 MHz.
2) The CAN Controller operates at 20 MHz.
3) The WCET of a functional process is estimated to be 200 microseconds.
4) The CAN bus transmits data at the rate of 500 kb/s.

We will focus our attention on the following Supervisor signals and on the messages that contain them:

1) "e_r_StrgRatOverrdF_orw", in message $0 \times 0D0$;
2) "e_w_VehYawRateCntrd_orw", in message 0×700;
3) "e_transmit_id_isg", in message $0 \times 7C0$.

Due to the specific offsets, messages 0×700 and $0 \times 7C0$ are enabled before message $0 \times 0D0$, respectively, at times 1 ms, 1 ms, and 2 ms.

Fig. 14 shows the GANTT charts for the three messages above (columns) for each of the three CAN configurations (rows).

Each chart illustrates the times when the signal is posted to the middleware buffer, to the CAN Controller transmit buffer, to the CAN Controller receive buffer (on the receiver node), and to the middleware buffer (on the receiver node).

Since the CAN Driver receives messages based on interrupt, signals stay in the CAN Controller receive buffer for a very short time.

In this exploration we are interested in highlighting priority inversion, so the latency of the first (highest priority) signal, within message $0 \times 0D0$ is the most

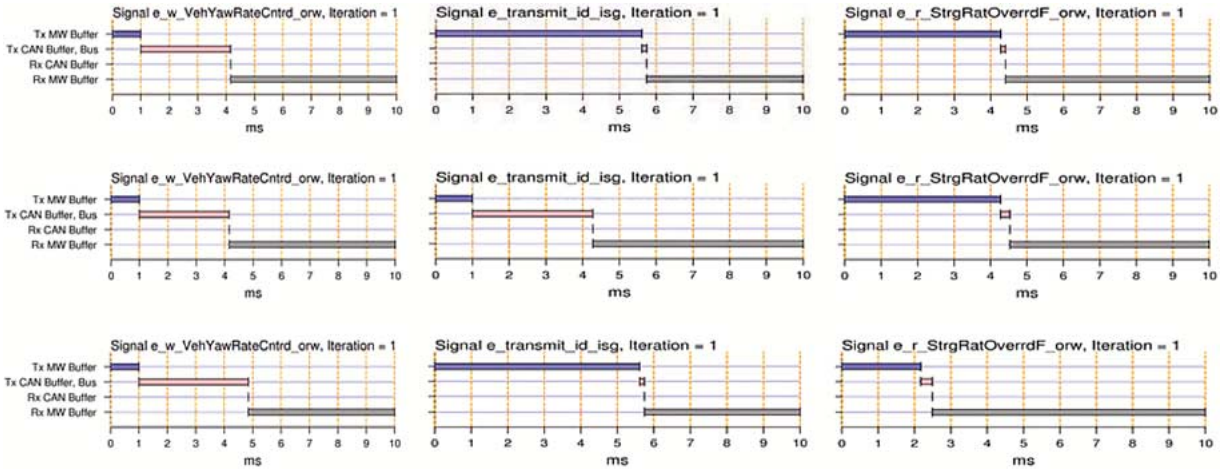


Fig. 14. *Rows correspond to three CAN configurations: one transmit buffer, two transmit buffers, and two transmit buffers with priority threshold, respectively.*

interesting metric. In other explorations the designer may be interested in other metrics, like the timing of other events or the loss of signals due to overwrites.

In the first and second CAN configurations, the two low priority messages 0×700 and $0 \times 7C0$ "lock" the transmit buffer(s) for a long stretch of time, and, although enabled at time 2 ms, the high-priority message $0 \times 0D0$ needs to wait in the middleware due to priority inversion.

It is interesting to note that in the second CAN configuration the additional transmit buffer does not mitigate the priority inversion problem. In fact, in the second rows of Fig. 14 message $0 \times 0D0$ is transmitted after message $0 \times 7C0$. More precisely, the start-of-transmission times in the second row of Fig. 14 are 4.2801 ms for message $0 \times 7C0$ and 4.5321 ms for message $0 \times 0D0$.

In principle, after message 0×700 is transmitted, there is a transmit buffer available for message $0 \times 0D0$, but in practice it takes time for the CAN Driver to copy the message into the transmit buffer, and message $0 \times 7C0$ starts the arbitration and the following transmission before then.

In the third CAN configuration, because one of the buffers is dedicated to messages with ID $\leq 0 \times 400$, although both message 0×700 and $0 \times 7C0$) are enabled at time 1 ms only message 0×700 is allowed to occupy a transmit buffer.

Clearly, when at time 2 ms message $0 \times 0D0$ is enabled, it will be queued quickly in the vacant transmit buffer, thus avoiding priority inversion.

VI. CONCLUSION

I discussed the trends and challenges of system design from a broad perspective that covers both semiconductor and industrial segments that use embedded systems. In particular, I presented the dynamics of the mobile terminal and automotive segments and how the different players interact. Existing approaches to hardware design above RTL and embedded system design have been presented. I then argued about the need for a unified way of thinking about system design as the basis for a novel system science. One approach was presented, PBD, that aims at achieving that unifying role. I discussed some of the most promising approaches for chip and embedded system design in the PBD perspective. The Metropolis framework and some examples of its application to different industrial domains were then described.

While I believe we are making significant in-roads, much work remains to be done to transfer the ideas and approaches that are flourishing today in research and in advanced companies to the generality of IC and embedded system designers. To be able to do so, we need the following.

1) We need to advance further the understanding of the relationships among parts of an heterogeneous design and its interaction with the physical environment.
2) The efficiency of the algorithms and of the tools must be improved to offer a solid foundation to the users.
3) Models and use cases have to be developed.
4) The scope of system level design must be extended to include fault-tolerance, security, and resiliency.
5) The EDA industry has to embrace the new paradigms and venture into uncharted waters to grow beyond where it is today. It must create the necessary tools to help engineers to apply the new paradigms.
6) Academia must develop new curricula (for example, see the special issue on education of the *ACM Transactions on Embedded Systems* [37]) that favor a broader approach to engineering while stressing the importance of foundational disciplines such as mathematics and physics; embedded system designers require a broad view and the capability of mastering heterogeneous technologies.
7) The system and semiconductor industry must recognize the importance of investing in training and tools for their engineers to be able to bring to market the new products and services that are described in exciting scenarios.

I believe that in the next few years we will witness a new renaissance in design technology and science. The winners in this arena are unclear as several players are eyeing system level design and design chain support as a promising area for them to engage. It is an exciting moment for the people involved! ■

Acknowledgment

This paper is dedicated to the memory of my long-time friend and colleague, Richard Newton, who passed away in January. He had been a great companion in research, vision, industrial interaction, and life in general. He will be sorely missed. I wish to acknowledge the support of the Gigascale System Research Center and of its research team, in particular J. Rabaey and the colleagues of the Heterogeneous System Theme (J. Cong, L. Carloni, L. Daniel, W. Hwu, R. Marculescu, and J. Roychowdhuri), the support of the NSF-sponsored Center for Hybrid and Embedded Software Systems (H. Gill, Program Director) and of its research team (T. Henzinger, E. Lee, S. Sastry, J. Sztipanovits, and C. Tomlin), the discussions with K. Keutzer, the great working relationship with A. Benveniste, B. Caillaud, L. Carloni, and P. Caspi, the support of the EU networks of excellence ARTIST and HYCON (Directors: J. Sifakis and F. Lamnabhi), and of the European community projects Columbus, Computation

and Control, SPEEDS, and RIMACS. I wish to thank most of all my present and former students and my colleagues and collaborators of PARADES (in particular, A. Ferrari who contributed to the first implementation and definition of Platform-Based Design and A. Balluchi who contributed substantially to the research described here). The past and present support of Cadence (in particular, the contributions of F. Balarin, L. Lavagno, C. Pinello, Y. Watanabe, and T. Vucurevich), Ferrari, General Motors (in particular, the support and contributions of P. Popp, M. di Natale, T. Fuhrman, P. Giusto, and S. Kanayan), Infineon (in particular, U. Ramacher), Intel (in particular, J. Moondanos, G. Frostig, Z. Hanna, A. Gupta, and D. Schmidt) Pirelli (in particular, G. Audisio and E. Albizzati), ST, Telecom Italia (in particular, F. Bellifemine e Fulvio Faraci), UMC, and United Technologies Corporation (in particular, the strong interaction with C. Jacobson and J. F. Cassidy, Jr.) is also gratefully acknowledged. Particular thanks go to C. Jacobson, M. Sgroi, L. Carloni, M. di Natale, and C. Pinello for a very thorough review of earlier versions of the manuscript.

REFERENCES

[1] The Xpilot System. [Online]. Available: http://cadlab.cs.ucla.edu/soc

[2] Artist 2. [Online]. Available: http://www.artist.org

[3] M. Abadi and L. Lamport, "Conjoining specifications," *ACM Trans. Programming Languages Syst.*, vol. 17, pp. 507–534, 1995.

[4] A. Benveniste, B. Caillaud, L. Carloni, P. Caspi, and A. Sangiovanni-Vincentelli, "Heterogeneous reactive systems modeling: Capturing causality and the correctness of loosely time-triggered architectures (ltta)," in *Proc. EMSOFT'2004*, G. Buttazzo and S. Edwards, Eds., Sep. 2004.

[5] AbsInt. [Online]. Available: http://www.absint.com/wcet.htm

[6] Accent. [Online]. Available: http://www.accent.it

[7] A. Davie, *Introduction to Functional Programming System Using Haskell*. Cambridge, U.K.: Cambridge Univ. Press, 1992.

[8] G. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott, "A foundation for actor computation," *J. Functional Programming*, vol. 7, no. 1, p. 172, 1997.

[9] G. A. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*. Cambridge, MA: MIT Press, 1986.

[10] R. Allen and D. Garlan, "Formalizing architectural connection," in *Proc. 16th Int. Conf. Software Engineering (ICSE 94)*, May 1994, pp. 71–80.

[11] R. Alur and T. A. Henzinger, *Reactive Modules*, vol. 15, pp. 7–48, 1999.

[12] F. Angiolini, J. Ceng, R. Leupers, F. Ferrari, C. Ferri, and L. Benini, "An integrated open framework for heterogeneous MPSOC design space exploration," in *Proc. Design, Automation Test Eur. Conf. (DATE'06)*, 2006, pp. 1145–1150.

[13] ARM. *Amba Specification*. [Online]. Available: http://www.arm.com/products/solutions/

[14] Arteris. [Online]. Available: http://www.arteris.com/

[15] Arvind and X. Shien, "Using term-rewriting systems to design and verify processors," *IEEE Micro Special Issue on Modeling and Validation of Microprocessors*, May 1999.

[16] Scientific Computing Associates. *Linda*. [Online]. Available: http://www.lindaspaces.com/

[17] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, E. Sentovich, K. Suzuki, and B. Tabbara, *Hardware-Software Co-Design of Embedded Systems—The Polis Approach*. Boston, MA: Kluwer, 1997.

[18] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, A. Sangiovanni-Vincentelli, E. Sentovich, and K. Suzuki, "Synthesis of software programs for embedded control applications," *IEEE Trans. Computer-Aided Design Integrated Circuits Syst.*, vol. 18, no. 6, p. 834, 1999.

[19] F. Balarin, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, M. Sgroi, and Y. Watanabe, "Modeling and designing heterogenous systems," in *Concurrency and Hardware Design*, J. Cortadella, A. Yakovlev, and G. Rozenberg, Eds. New York: Springer, 2002, vol. LNCS2549, pp. 228–273.

[20] F. Balarin, Y. Watanabe, J. Burch, L. Lavagno, R. Passerone, and A. Sangiovanni-Vincentelli, "Constraints specification at higher levels of abstraction," in *Proc. Int. Workshop High Level Design Validation and Test*, Nov. 2001.

[21] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli, "Metropolis: An integrated electronic system design environment," *IEEE Comput.*, vol. 36, no. 4, pp. 45–52, Apr. 2003.

[22] F. Balarin, L. Lavagno *et al.*, "Concurrent execution semantics and sequential simulation algorithms for the metropolis metamodel," in *Proc. 10th Int. Symp. Hardware/Software Codesign*, 2002, pp. 13–18.

[23] J. R. Bammi, W. Kruijtzer, L. Lavagno, E. Harcourt, and M. T. Lazarescu, "Software performance estimation strategies in a system-level design tool," in *CODES '00: Proc. 8th Int. Workshop Hardware/Software Codesign*, ACM Press, 2000, pp. 82–86.

[24] L. Benini and G. De Micheli, "Networks on chip: A new SoC paradigm," *IEEE Comput.*, vol. 35, no. 1, pp. 70–79, Jan. 2002.

[25] A. Benveniste and G. Berry, "The synchronous approach to reactive and real-time systems," *Proc. IEEE*, vol. 79, no. 9, pp. 1270–1282, Sep. 1991.

[26] G. Berry, S. Moisan, and J.-P. Rigault, "Esterel: Toward a synchronous and semantically sound high level language for real time applications," in *Proc. IEEE Real Time Systems Symp.*, 1983.

[27] BlueSpec. [Online]. Available: http://www.bluespec.com

[28] A. Bonivento, L. P. Carloni, and A. Sangiovanni-Vincentelli, "Rialto: A bridge between description and implementation of control algorithms for wireless sensor networks," in *Proc. EMSOFT*, Jersey City, NJ, Sep. 2005.

[29] ——, "Platform-based design for wireless sensor networks," *Monet, J. Mobile Networks Applic.*, 2006.

[30] A. Bonivento, C. Fischione, A. Sangiovanni-Vincentelli, F. Graziosi, and F. Santucci, "Seran: A semi random protocol solution for clustered wireless sensor networks," in *Proc. MASS*, Washington, DC, Nov. 2005.

[31] G. Booch, J. Rambaugh, and J. Jacobson, *The Unified Modeling Language User Guide*, ser. Object Technology Series. New York: Addison-Wesley, 1999.

[32] E. Börger, "Logic programming: The evolving algebra approach," in *Proc. IFIP 13th World Computer Congr.*, vol. I, *Technology/Foundations*, B. Pehrson and I. Simon, Eds., Elsevier, Amsterdam, The Netherlands, 1994, pp. 391–395.

[33] R. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang, "MIS: A multiple-level logic optimization system," *IEEE Trans. Computer-Aided Design*, vol. CAD-6, no. 6, pp. 1062–1082, Nov. 1987.

[34] J. Buck. (2001). "Scheduling and code generation in CoCentric system studio," in *Fourth Ptolemy Miniconference*. [Online]. Available: http://ptolemy.berkeley.edu/conferences/01/src/miniconf/

[35] J. Buck, S. Ha, E. A. Lee, and D. G. Masserschmitt, "Ptolemy: A framework for simulating and prototyping heterogeneous systems," *Int. J. Computer Simulation, Special Issue on Simulation Software Development*, Jan. 1990.

[36] L. Burch, R. Passerone, and A. Sangiovanni-Vincentelli, "Overcoming heterophobia: Modeling concurrency in heterogeneous systems," in *Proc. Conf. Application of Concurrency to System Design*, Jun. 2001.

[37] A. Burns and A. Sangiovanni-Vincentelli, "Editorial," *ACM Trans. Embedded Computing Systems, Special Issue on Education*, vol. 4, no. 3, pp. 472–499, Aug. 2005.

[38] Buttazzo, *Hard Real-Time Computing Systems*. Boston, MA: Kluwer, 1997.

[39] ——, *Hard Real-Time Computing Systems*. New York: Springer-Verlag, 2004.

[40] J. P. Calvez, *Embedded Real-Time Systems. A Specification and Design Methodology*. New York: Wiley, 1993.

[41] CAN-Specification. Road Vehicles—Interchange of Digital Information—Controller—Area Network (CAN) for

High-Speed Communication—ISO 11898.

[42] L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli, "Theory of latency-insensitive design," *IEEE Trans. Computer-Aided Design*, vol. 20, no. 9, pp. 1059–1076, Sep. 2001.

[43] L. P. Carloni and A. L. Sangiovanni-Vincentelli, "Coping with latency in SOC design," *IEEE Micro*, vol. 22, no. 5, pp. 24–35, Sep.–Oct. 2002.

[44] L. P. Carloni, F. De Bernardinis, C. Pinello, A. Sangiovanni-Vincentelli, and M. Sgroi, "Platform-based design for embedded systems," in *The Embedded Systems Handbook*, Boca Raton, FL: CRC, 2005.

[45] L. P. Carloni, F. De Bernardinis, A. L. Sangiovanni-Vincentelli, and M. Sgroi, "The art and science of integrated systems design," in *Proc. Eur. Solid-State Circuits Conf.*, Sep. 2002.

[46] N. Carriero and D. Gelernter, "Coordination languages and their significance," *Commun. ACM*, vol. 35, no. 2, pp. 97–107, 1992.

[47] Celoxica, *Handel-c Language Reference Manual*, Tech. Rep. RM-1003-4.0, 2003.

[48] H. Chang, L. Cooke, M. Hunt, G. Martin, A. McNelly, and L. Todd, *Surviving the SOC Revolution: A Guide to Platform-Based Design*. Boston, MA: Kluwer, 1999.

[49] X. Chen, H. Hsieh, F. Balarin, and Y. Watanabe, "Logic of constraints: A quantitative performance and functional constraint formalism," *IEEE Trans. Computer-Aided Design Integrated Circuits*, vol. 23, no. 8, Aug. 2004.

[50] P. Ciancarini, "Coordination models and languages as software integrators," *ACM Computing Surveys*, vol. 28, no. 2, Jun. 1996.

[51] European Community. Building Artemis (Advanced Research and Technology for Embedded Intelligence and Systems), 2004, Rep. High Level Group on Embedded Systems.

[52] Mentor Graphics Corporation. *Platform Express*. [Online]. Available: http://www.mentor.com/products/

[53] P. Cousot and R. Cousot, *Verification of Embedded Software: Problems and Perspectives*, 2001.

[54] D. Culler, D. Estrin, and M. Srivastava, "Overview of sensor networks," *IEEE Comput.*, vol. 37, no. 8, pp. 41–49, Aug. 2004.

[55] P. Cumming, "The TI OMAP platform approach to SOCs," in *Surviving the SOC Revolution: A Guide to Platform-Based Design*, H. Chang *et al.*, Eds. Boston, MA: Kluwer, 1999, ch. 5, pp. 1–29.

[56] W. J. Dally and B. Towles, "Route packets, not wires: Onhip interconnection networks," in *Proc. Design Automation Conf.*, Jun. 2001, pp. 684–689.

[57] A. Davare, K. Lwin, A. Kondratyev, and A. Sangiovanni-Vincentelli, "The best of both worlds: The efficient asynchronous implementation of synchronous specifications," in *Proc. Design Automation Conf. (DAC)*, Jun. 2004.

[58] A. Davare, Q. Zhu, J. Moondanos, and A. L. Sangiovanni-Vincentelli, "JPEG encoding on the Intel MXP5800: A platform-based design case study," in *Proc. 3rd Workshop Embedded Systems for Real-Time Multimedia*, 2005, pp. 89–94.

[59] L. de Alfaro and T. A. Henzinger, "Interface automata," in *Proc. Joint 8th Eur. Software Eng. Conf. 9th ACM SIGSOFT Int. Symp. Foundations of Software Eng.*, 2001.

[60] L. de Alfaro, T. A. Henzinger, and R. Jhala, *Compositional Methods for Probabilistic Systems*, vol. 2154, pp. 351–365, 2001.

[61] F. De Bernardinis, P. Nuzzo, and A. Sangiovanni Vincentelli, "Mixed signal design space exploration through analog platforms," in *Proc. IEEE/ACM DAC*, 2005.

[62] J. A. de Oliveira and H. van Antwerpen, "The Philips Nexperia digital video platforms," in *Winning the SoC Revolution*, G. Martin and H. Chang, Eds. Boston, MA: Kluwer, 2003, pp. 67–96.

[63] D. Densmore, R. Passerone, and A. Sangiovanni-Vincentelli, "A platform-based taxonomy for ESL design," *IEEE Design Test*, vol. 23, no. 5, pp. 359–374, Oct. 2006.

[64] D. Densmore, S. Rekhi, and A. Sangiovanni-Vincentelli, "Microarchitecture development via metropolis successive platform refinement," in *Proc. Design Automation Test Eur. (DATE)*, Feb. 2004.

[65] Mirabilis Design, *Visualsim*. [Online]. Available: http://www.mirabilisdesign.com/WebPages/presentation/

[66] "Model-driven software development," *IBM Syst. J.*, vol. 45, no. 3, 2006.

[67] S. Director, A. Parker, D. Siewiorek, and D. Thomas, Jr., "A design methodology and computer aids for digital VLSI systems," *IEEE Trans. Circuits Syst.*, vol. 28, no. 7, pp. 634–645, 1981.

[68] R. Dömer, A. Gerstlauer, J. Zhu, S. Zhao, and D. D. Gajski, *Specc: Design Methodology*. Boston, MA: Kluwer, 2000.

[69] *Arp 4754—Certification Considerations for Highly-Integrated or Complex Aircraft Systems*, Nov. 1996, DO178B/C.

[70] *Arp 4761—Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment*, Dec. 1996, DO178B/C.

[71] dSpace. *Targetlink*. [Online]. Available: http://www.dspaceinc.com/ww/en/inc/home/

[72] Dynasim. [Online]. Available: http://www.dynasim.se

[73] e Silicon. [Online]. Available: http://www.esilicon.com

[74] S. Edwards, "The challenges of synthesizing hardware from c-like languages," *IEEE Design Test Comput.*, pp. 375–386, Oct. 2006.

[75] S. Edwards, L. Lavagno, E. A. Lee, and A. Sangiovanni-Vincentelli, "Design of embedded systems: Formal models, validation, and synthesis," *Proc. IEEE*, vol. 85, no. 3, pp. 366–390, Mar. 1997.

[76] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, J. Ludwig, S. Neuendorffer, S. Sachs, and Y. Xiong, "Taming heterogeneity—The Ptolemy approach," *Proc. IEEE*, vol. 91, no. 1, pp. 127–144, Jan. 2003.

[77] C. Erbas, S. C. Erbas, and A. D. Pimentel, "A multiobjective optimization model for exploring multiprocessor mappings of process networks," in *Proc. First IEEE/ACM/IFIP Int. Conf. Hardware/Software Codesign and System Synthesis*, ACM Press, 2003, pp. 182–187.

[78] A. Burns *et al.*, "Engineering a hard real-time system: From theory to practice," *Software: Practice Experience*, vol. 25, no. 7, p. 705, 1995.

[79] A. Mihal *et al.*, "Developing architectural platforms: A disciplined approach," *IEEE Des. Test*, vol. 19, no. 6, pp. 6–16, 2002.

[80] A. Ferrari *et al.*, *Automatic Code Generation and Platform Based Design Methodology: An Engine Management System Design Case Study*, 2004.

[81] G. De Micheli *et al.*, "The Olympus synthesis system," *IEEE Design Test Comput.*, vol. 7, no. 5, pp. 37–53, 1990.

[82] J.-Y. Brunel *et al.*, *Softcontract: An Assertion-Based Software Development Process That Enables Design-by-Contract*, 2004.

[83] N. Audsley *et al.*, "Fixed priority pre-emptive scheduling: An historical perspective," *Real-Time Systems*, vol. 8, no. 2–3, p. 173, 1995.

[84] N. C. Audsley *et al.*, "Deadline monotonic scheduling theory and application," *Control Engineering Practice*, vol. 1, no. 1, p. 71, 1993.

[85] T. Kambe *et al.*, "A C-based synthesis system, Bach, and its application," in *Proc. Asia South Pacific Design Automation Conf. (ASP-DAC 01)*, 2001, pp. 151–155.

[86] F. P. Brooks, Jr., "No silver bullet: Essence and accidents of software engineering," *Comput. Mag.*, Apr. 1987.

[87] Faraday. [Online]. Available: http://www.faraday-tech.com

[88] P. H. Feiler, B. Lewis, and S. Vestal, *The Sae Architecture Analysis and Design Language (aadl) Standard: A Basis for Model-Based Architecture-Driven Embedded Systems Engineering*, p. 110, 2003.

[89] C. Ferdinand, F. Martin, R. Wilhelm, and M. Alt, "Cache behavior prediction by abstract interpretation," *Sci. Comput. Programming*, 1998.

[90] T. Fitzpatrick, A. Salz, D. Rich, and S. Sutherland, *System Verilog for Verification*. New York: Springer, 2006.

[91] The Eclipse Foundation, *Eclipse Platform Technical Overview*, 2003.

[92] Freescale. [Online]. Available: http://www.freescale.com

[93] F. Ghenassia, *Transaction-Level Modeling With SystemC*. New York: Springer, 2005.

[94] Giotto. [Online]. Available: http://embedded.eecs.berkeley.edu/giotto/

[95] R. Goering, "Platform-based design: A choice, not a panacea," *The Art of Change: Technologies for Designing our Future, Yearbook, EE Times*, Sep. 2002.

[96] G. Goessler and J. Sifakis, "Composition for component-based modeling," *Sci. Comput. Programming*, vol. 55, pp. 161–183, Mar. 2005.

[97] T. Grotker, *System Design With SystemC*. Boston, MA: Kluwer, 2002.

[98] Object Management Group. *Model Driven Architecture (mda) Faq*. [Online]. Available: http://www.omg.org/mda/

[99] Object Management Group, *Systems Modeling Language (sysml) Specification*, ver. 1.0, May 2006.

[100] OSEK Group, *Osek/vdx Operating System Specification 2.0*, Jun. 1997.

[101] GSRC. *The Gigascale System Research Center*. [Online]. Available: http://www.gigascale.org

[102] P. Le Guernic, A. Benveniste, P. Bournai, and T. Gautier, "Signal: A data-flow oriented language for signal processing," *IEEE Trans. Acoust., Speech, Signal Processing,* vol. 34, pp. 362–374, 1986.

[103] Y. Gurevich, "A new thesis," in *Abstracts, American Mathematical Society,* Aug. 1985, p. 317.

[104] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous data flow programming language LUSTRE," *Proc. IEEE,* vol. 79, no. 9, pp. 1305–1320, Sep. 1991.

[105] D. Harel, "Statecharts: A visual formalism for complex systems," *Sci. Comput. Programming,* vol. 8, no. 3, pp. 231–274, Jun. 1987.

[106] R. Heckman, M. Langenbach, S. Thesing, and R. Wilhelm, "The influence of processor architecture on the design and the results of WCET tools," *Proc. IEEE,* vol. 91, no. 7, Jul. 2003.

[107] H. Heinecke, K.-P. Schnelle, H. Fennel, J. Bortolazzi, L. Lundh, J. Leflour, J.-L. Maté, K. Nishikawa, and T. Scharnhorst, "Automotive open system architecture an industry-wide initiative to manage the complexity of emerging automotive e/e-architectures," in *Proc. Convergence 2004*, Detroit, MI, Oct. 2004.

[108] G. Hellestrand, "The engineering of supersystems," *Computer,* vol. 38, no. 1, pp. 103–105, Jan. 2005.

[109] A. Hemani, A. Jantsch, S. Kumar, A. Postula, J. Vberg, M. Millberg, and D. Lindqvist, "Network on chip: An architecture for billion transistor era," in *Proc. IEEE NorChip Conf.*, Nov. 2000.

[110] T. A. Henzinger, B. Horowitz, and C. M. Kirsch, "Giotto: A time-triggered language for embedded programming," *Proc. IEEE,* vol. 91, pp. 84–99, 2003.

[111] T. A. Henzinger and J. Sifakis, "The embedded systems design challenge," in *Lecture Notes in Computer Science,* New York: Springer, 2006.

[112] T. A. Henzinger, E. A. Lee, A. L. Sangiovanni-Vincentelli, S. S. Sastry, and J. Sztipanovits. (2002). Mission Statement: Center for Hybrid and Embedded Software Systems. [Online]. Available: http://chess.eecs.berkeley.edu

[113] T. A. Henzinger, M. Minea, and V. Prabhu, *Assume-Guarantee Reasoning for Hierarchical Hybrid Systems*. New York: Springer-Verlag, 2001, vol. 2034, pp. 275–290.

[114] C. Hewitt, "Viewing control structures as patterns of passing messages," *J. Artificial Intelligence,* vol. 8, no. 3, pp. 323–363, Jun. 1977.

[115] P. Hilfinger, "A high-level language and silicon compiler for digital signal processing," in *Proc. Custom Integrated Circuits Conf.*, May 1985, pp. 213–216.

[116] R. Ho, K. W. Mai, and M. A. Horowitz, "The future of wires," *Proc. IEEE*, pp. 490–504, Apr. 2001.

[117] G. J. Holzmann, "The model checker SPIN," *IEEE Trans. Software Eng.,* vol. 23, no. 5, pp. 258–279, May 1997.

[118] P. Hudak. (1997). *Keynote Address at the Usenix dsl Conf.* [Online]. Available: http://www.cs.yale.edu/HTML/YALE/CS/HyPlans/hudak-paul/hudak-dir/dsl/index.htm

[119] HYCON. [Online]. Available: http://www.hycon.org

[120] IBM. *Analysis, Design and Construction Tools*. [Online]. Available: http://www-306.ibm.com/software/rational/offerings/design.html

[121] IBM. (1999). *Coreconnect Bus Architecture*. [Online]. Available: http://www.chips.ibm.com/products/coreconnect

[122] Cadence Design Systems Inc. *Incisive Verification Platform*. [Online]. Available: http://www.cadence.com

[123] Cadence Design Systems Inc. (1998). *Cierto vcc User Guide*. [Online]. Available: http://www.cadence.com

[124] CoWare Inc. *Convergensc*. [Online]. Available: http://www.coware.com/products/

[125] National Instruments. *Labview*. [Online]. Available: www.ni.com/labview/

[126] ISIS. *Model-Based Integrated Simulation: Milan*. [Online]. Available: http://milan.usc.edu/

[127] G. Karsai, J. Sztipanovits, A. Ledeczi, and T. Bapty, "Model-integrated development of embedded software," *Proc. IEEE,* vol. 91, no. 1, pp. 145–164, Jan. 2003.

[128] Kermeta, *The Kermeta Metamodelling Workbench,* 2003.

[129] K. Keutzer, S. Malik, A. R. Newton, J. Rabaey, and A. Sangiovanni-Vincentelli, "System level design: Orthogonolization of concerns and platform-based design," *IEEE Trans. Computer-Aided Design Integrated Circuits Syst.,* vol. 19, no. 12, pp. 1523–1543, Dec. 2000.

[130] B. Kienhuis, E. Deprettere, P. van der Wolf, and K. Vissers, "A methodology to design programmable embedded systems," *SAMOS: Systems, Architectures, Modeling, and Simulation,* p. 2268, Nov. 2001.

[131] H. Kopetz, *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Boston, MA: Kluwer, 1997.

[132] R. Kumar, V. Zyuban, and D. M. Tullsen, "Interconnections in multi-core architectures: Understanding mechanisms, overheads and scaling," in *Proc. Ann. Int. Symp. Computer Architecture*.

[133] J. Kunkel, "COSSAP: A stream driven simulator," presented at the IEEE Int. Workshop Microelectrion. Commun., Interlaken, Switzerland, Mar. 1991.

[134] L. Lavagno, M. Di Natale, A. Ferrari, and P. Giusto, "Softcontract: Model-based design of error-checking code and property monitors," *UML 2004 Satellite Activities,* vol. 3297, *LNCS*, pp. 150–162, 2005.

[135] E. A. Lee, "Absolutely positively on time: What would it take?" *IEEE Comput.,* vol. 38, no. 7, pp. 85–87, Jul. 2005.

[136] E. A. Lee and A. Sangiovanni-Vincentelli, "A framework for comparing models of computation," *IEEE Trans. Computer-Aided Design Integrated Circuits Syst.,* vol. 17, pp. 1217–1229, Dec. 1998.

[137] E. A. Lee and Y. Xiong, "System-level types for component-based design," in *Proc. First Int. Workshop Embedded Software, EMSOFT'01*, vol. 2211, LNCS, T. A. Henzinger and C. M. Kirsch, Eds., 2001, pp. 32–49.

[138] Y. T. S. Li and S. Malik, "Performance analysis of embedded software using implicit path enumeration," in *Proc. Design Automation Conf.*, Jun. 1995.

[139] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment," *J. ACM,* vol. 20, no. 1, pp. 46–61, 1973.

[140] M. Loghi, F. Angiolini, D. Bertozzi, L. Benini, and R. Zafalon, "Analyzing on-chip communication in a mpsoc environment," in *Proc. Design, Automation Test Eur. Conf. (DATE'04),* 2004.

[141] ARM Ltd., *Realview Maxsim*.

[142] D. MacQueen and R. Milner, *The Definition of Standard ML*. Cambridge, MA: MIT Press, 1997.

[143] R. Marculescu, U. Y. Ogras, and N. H. Zamora, "Computation and communication refinement for multiprocessor soc design: A system-level perspective," *ACM Trans. Design Automation Electronic Systems,* vol. 11, no. 3, pp. 564–592, Jul. 2006.

[144] R. Marculescu, J. Rabaey, and A. Sangiovanni-Vincentelli, "Is 'Network' the next 'Big Idea' in design?" Mar. 2006.

[145] G. Martin and B. Salefski, "Methodology and technology for design of communications and multimedia products via system-level IP integration," in *Proc. Conf. Design, Automation Test Eur.*, IEEE Computer Soc., 1998.

[146] D. Mathaikutty, H. Patel, and S. Shukla, "A functional programming framework of heterogeneous model of computations for system design," in *Proc. Forum Specification and Design Languages (FDL),* 2004.

[147] D. Mathaikutty, H. Patel, S. Shukla, and A. Jantsch, "Umoc++: A C++-based multi-moc modeling environment," in *Advances in Design and Specification Languages for SoCs—Selected Contributions Ffrom FDL'05*, A. Vachoux, Ed., 2005, Ch. 7.

[148] The Mathworks. *Real-Time Workshop*. [Online]. Available: http://www.mathworks.com/products/rtw/

[149] K. Keutzer and M. Gries, Eds., *Building ASIPs: The Mescal Methodology*. New York: Springer, 2005.

[150] K. L. McMillan, "A compositional rule for hardware design refinement," in *Proc. Computer-Aided Verification Conf., Lecture Notes in Computer Science,* 1997, pp. 24–35.

[151] J. D. Meindl, "Interconnect opportunities for gigascale integration," *IEEE Micro,* vol. 23, no. 3, pp. 28–35, May–Jun. 2003.

[152] G. De Micheli, "Hardware synthesis from C/C++ models," in *Proc. Design, Automation Test Eur. (DATE 99),* 1999, pp. 382–383.

[153] IBM Microelectronics. [Online]. Available: http://www.ibm.com/chips

[154] Modelica and the Modelica Association. [Online]. Available: http://www.modelica.org/

[155] MICA motes. [Online]. Available: http://www.xbow.com/

[156] TELOS motes. [Online]. Available: http://www.moteiv.com/

[157] OCP-IP, *Open Core Protocol Specification,* release 2.1.3.

[158] OMG. [Online]. Available: http://www.omg.org

[159] F. Fleurey, P.-A. Muller, and J.-M. Jzquel, "Weaving executability into object-oriented meta-languages," in *Proc. MODELS 2005: 8th Int. Conf. Model Driven Eng. Languages Syst.*, Oct. 2005.

[160] L. Palopoli, C. Pinello, A. L. Sangiovanni-Vincentelli, L. El-Ghaoui, and A. Bicchi, Synthesis of robust control systems under resource constraints," in *Hybrid Systems: Computation and Control*, M. Greenstreet and

C. Tomlin, Eds. Heidelberg, Germany: Springer-Verlag, 2002, vol. LNCS 2289, *Lecture Notes in Computer Science*, pp. 337–350.

[161] G. Papadopoulos and F. Arbab, *Coordination Models and Languages*. Berlin, Germany: Springer, 1998.

[162] S. Pasricha, N. Dutt, and M. Ben-Romdhane, *Using TLM for Exploring Bus-Based SOC Communication Architectures*, 2005.

[163] R. Passerone, "Semantic foundations for heterogeneous systems," Ph.D. dissertation, Univ. California, Berkeley, 2004.

[164] R. Passerone, L. de Alfaro, T. A. Henzinger, and A. Sangiovanni-Vincentelli, "Convertibility verification and converter synthesis: Two faces of the same coin," in *Proc. Int. Conf. Computer Aided Design*, Nov. 2002.

[165] R. Passerone, J. Rowson, and A. Sangiovanni-Vincentelli, System and a method for automatically synthesizing interfaces between incompatible protocols, U.S. Patent.

[166] A. D. Pimentel, "The Artemis workbench for system-level performance evaluation of embedded systems," *Int. J. Embedded Systems*, 2005.

[167] A. D. Pimentel, L. O. Hertzberger, P. Lieverse, P. van der Wolf, and E. F. Deprettere, "Exploring embedded-systems architectures with Artemis," *Computer*, vol. 34, no. 11, pp. 57–63, 2001.

[168] A. Pinto, A. Bonivento, A. Sangiovanni-Vincentelli, R. Passerone, and M. Sgroi, "System-level design paradigms: Platform-based design and communication synthesis," *ACM Trans. Embedded Computing Syst.*, vol. 5, no. 5, May 2006.

[169] J. Rabaey, H. De Man, J. Vanhoof, G. Goossens, and F. Catthoor, "CATHEDRAL-II: A synthesis system for multiprocessor DSP systems," *Silicon Compilation*, Dec. 1987.

[170] K. Van Rompaey, "Designing SOC platforms for maximum flexibility," *EE Times*, 2000.

[171] K. Van Rompaey, D. Verkest, I. Bolsens, and H. De Man, "Coware-A design environment for heterogeneous hardware/software systems," in *Proc. EURO-DAC*, Sep. 1996, pp. 252–257.

[172] I. Sander and A. Jantsch, "System modeling and transformational design refinement in forsyde," *IEEE Trans. Computer-Aided Design Integrated Circuits Syst.*, vol. 23, no. 1, pp. 17–32, Jan. 2004.

[173] A. Sangiovanni-Vincentelli, "Defining platform-based design," *EEDesign of EETimes*, Feb. 2002.

[174] A. Sangiovanni-Vincentelli, L. Carloni, F. De Bernardinis, and M. Sgroi, "Benefits and challenges for platform-based design," in *Proc. DAC*, Jun. 2004, pp. 409–414.

[175] A. Sangiovanni-Vincentelli and A. Ferrari, "System design—Traditional concepts and new paradigms," in *Proc. ICCD*, Oct. 1999.

[176] A. Sangiovanni-Vincentelli and G. Martin, "Platform-based design and software design methodology for embedded systems," *IEEE Design Test Computers*, vol. 18, no. 6, pp. 23–33, 2001.

[177] B. Selic, G. Gullekson, and P. Ward, *Real-Time Object-Oriented Modeling*. New York: Wiley, 1994.

[178] Semiconductor Industry Assoc. (SIA), San Jose, CA, *International Technology Roadmap for Semiconductors Update (Design)*, 2004.

[179] M. Sgroi, M. Sheets, A. Mihal, K. Keutzer, S. Malik, J. Rabaey, and A. Sangiovanni-Vincentelli, "Addressing the system-on-a-chip interconnect woes through communication-based design," pp. 667–672, Jun. 2001.

[180] M. Sgroi, A. Wolisz, A. Sangiovanni-Vincentelli, and J. Rabaey, "A service-based universal application interface for *ad hoc* wireless sensor networks," in *White paper, U.C. Berkeley*, 2004.

[181] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority inheritance protocols: An approach to real-time synchronization," *IEEE Trans. Comput.*, vol. 39, pp. 1175–1185, Sep. 1990.

[182] K. S. Shanmugan, P. Titchener, and W. Newman, "Simulation based caad tools for communication and signal processing systems," in *Conf. Rec. IEEE Int. Conf. World Prosperity Through Commun.*, 1989, pp. 1454–1461.

[183] Simulink. [Online]. Available: http://www.mathworks.com/

[184] A. P. Sistla and E. M. Clarke, "The complexity of propositional linear temporal logics," in *Proc. 14th Ann. ACM Symp. Theory of Computing*, ACM Press, 1982, pp. 159–168.

[185] G. Smith, "Platforms and marketing: What a concept," *EE Times*, Sep. 2002.

[186] Sonics. *Smart Interconnect*. [Online]. Available: http://sonicsinc.com

[187] SPIRIT. [Online]. Available: http://www.spiritconsortium.org/home

[188] J. A. Stankovic, I. Lee, A. Mok, and R. Rajkumar, "Opportunities and obligations for physical computing systems," *IEEE Comput.*, vol. 38, no. 11, pp. 23–31, 2005.

[189] StateFlow. [Online]. Available: www.mathworks.com/products/stateflow/

[190] T. Stefanov, C. Zissulescu, A. Turjan, B. Kienhuis, and E. F. Deprettere, "System design using Kahn process networks: The compaan/laura approach," in *Proc. Int. Conf. Design, Automation Test Eur. (DATE)*, Feb. 2004, pp. 340–345.

[191] STMicroelectrinics. *Stbus Specification*. [Online]. Available: http://mcu.st.com/

[192] Cocentric System Studio. [Online]. Available: http://www.synopsys.com/products/

[193] S. Sutherland, S. Davidmann, and P. Flake, *SystemVerilog For Design: A Guide to Using SystemVerilog for Hardware Design and Modeling*. New York: Springer, 2004.

[194] K. Suzuki and A. Sangiovanni-Vincentelli, "Efficient software performance estimation methods for hardware/software codesign," in *Proc. Design Automation Conf.*, Jun. 1996, pp. 605–610.

[195] Proc. 2007 NOC Symp. Call for papers. [Online]. Available: http://www.nocsymposium.org/

[196] SystemC. [Online]. Available: http://www.systemc.org

[197] Telelogic. *Rhapsody*. [Online]. Available: http://www.ilogix.com/homepage.aspx

[198] Telelogic. *Systems Engineering, Software Development and Testing Automation*. [Online]. Available: http://www.telelogic.com/products/tau/index.cfm

[199] M. Tsien, private communication, 2004, General Motors.

[200] TSMC. [Online]. Available: http://www.tsmc.com

[201] UMC. [Online]. Available: http://www.umc.com

[202] Virtio. *Virtual Platforms*. [Online]. Available: http://www.virtio.com

[203] G. K. Wallace, "The JPEG still picture compression standard," *Commun. ACM*, vol. 34, no. 4, pp. 30–44, Apr. 1991.

[204] Wikipedia. [Online]. Available: http://en.wikipedia.org/wiki/embeddedsystems

[205] A. Willig, K. Matheus, and A. Wolisz, "Wireless technology in industrial networks," *Proc. IEEE*, vol. 93, no. 6, pp. 1130–1151, Jun. 2005.

[206] WindRiver. *Vxworks*. [Online]. Available: http://www.windriver.com/vxworks/index.html

[207] D. Wingard, *Micronetwork-Based Integration for SOCS*, 2001.

[208] W. Wolf, *Computers as Components: Principles of Embedded Computing Systems Design*, San Mateo, CA: Morgan Kaufmann, 2000.

[209] NoC Workshop. (2006). [Online]. Available: http://async.org.uk/noc2006/

[210] Xilinx. (2005). *Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet*. [Online]. Available: http://www.xilinx.com/bvdocs/publications/ds083.pdf

[211] G. Yang *et al.*, "Separation of concerns: Overhead in modeling and efficient simulation techniques," in *Proc. 4th ACM Int. Conf. Embedded Software*, Sep. 2004.

[212] G. Yang, H. Hsieh, X. Chen, F. Balarin, and A. Sangiovanni-Vincentelli, "Constraints assisted modeling and validation in metropolis framework," in *Proc. Asilomar Conf. Signals, Systems, and Computers*, Nov. 2006.

[213] J. Yoshida, "Philips semi see payoff in platform-based design," *EE Times*, Oct. 2002.

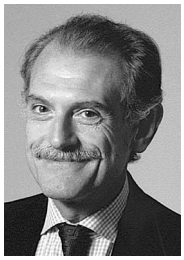
[214] ——, "St steps up set-top box strategy," *EE Times*, May 2004.

[215] H. Zeng, S. Sonalkar, S. Kanajan, C. Pinello, A. Davare, and A. Sangiovanni-Vincentelli, "Design space exploration of automotive platforms in metropolis," in *Proc. SAE World Congr.*, 2006.

[216] V. Zivojnovic and H. Meyr, "Compiled HW/SW co-simulation," in *Proc. Design Automation Conf.*, 1996.

[217] R. Zurawski, "Introduction to special issue on industrial communication systems," *Proc. IEEE*, vol. 93, no. 6, pp. 1067–1072, Jun. 2005.

ABOUT THE AUTHOR



Alberto Sangiovanni-Vincentelli (Fellow, IEEE) received the "Dottore in Ingegneria" (*summa cum laude*) from the Politecnico di Milano, Milano, Italy, in 1971.

In 1980 and 1981, he was a Visiting Scientist at the Mathematical Sciences Department of the IBM T.J. Watson Research Center. In 1987, he was Visiting Professor at MIT. Currently, he holds the Edgar L. and Harold H. Buttner Chair of Electrical Engineering and Computer Sciences at the University of California, Berkeley, where he has been on the faculty since 1976. He was a Cofounder of Cadence and Synopsys, the two leading companies in the area of Electronic Design Automation. He is the Chief Technology Advisor of Cadence Design System. He is a member of the Board of Directors of Cadence, Sonics Inc., Gradient Design Automation, UPEK, Value Partners, and Accent. He is a Cofounder of the Cadence Berkeley Laboratories. He was a member of the HP Strategic Technology Advisory Board. He is a member of the GM Science and Technology Advisory Board, and the Founder and Scientific Director of the Project on Advanced Research on Architectures and Design of Electronic Systems (PARADES), a European Group of Economic Interest supported by Cadence, United Technologies Corporation, and ST Microelectronics. He is a member of the Advisory Board of the Lester Center for Innovation of the Haas School of Business and of the Center for Western European Studies and a member of the Berkeley Roundtable of the International Economy (BRIE). He is a member of the High-Level Group and of the Steering Committee of the EU Artemis Technology Platform. He is an author of over 800 papers and 15 books in the area of design tools and methodologies, large-scale systems, embedded controllers, hybrid systems and innovation.

Dr. Sangiovanni-Vincentelli received the Distinguished Teaching Award of the University of California, in 1981. He received the worldwide 1995 Graduate Teaching Award of the IEEE (a Technical Field award for "inspirational teaching of graduate students"). He has received numerous best paper awards including the Guillemin–Cauer Award (1982–1983) and the Darlington Award (1987–1988). In 2002, he was the recipient of the Aristotle Award of the Semiconductor Research Corporation. In 2001, he was given the prestigious Kaufman Award of the Electronic Design Automation Council for pioneering contributions to EDA. He was elected to the National Academy of Engineering in 1998.