Platform-Based Design for Embedded Systems

Luca P. Carloni [a] Fernando De Bernardinis [a,b] Claudio Pinello [a]
Alberto L. Sangiovanni-Vincentelli [a] Marco Sgroi [a,c]

[a] *University of California at Berkeley, Berkeley, CA 94720-1772*
[b] *Dipartimento di Ingegneria dell'Informazione, Università di Pisa, Italy*
[c] *DoCoMo Euro-labs, Munich, Germany*

Abstract

A platform is an abstraction layer that hides the details of several possible implementation refinements of the underlying layers. It is a library of elements characterized by models that represent their functionalities and offer an estimation of (physical) quantities that are of importance for the designer. The library contains interconnects and rules that define what are the legal composition of the elements. A legal composition of elements and interconnects is called a platform instance. Platform-based design is a meet-in-the-middle process, where successive refinements of specifications meet with abstractions of potential implementations that are captured in the models of the elements of the platform. It is this characteristic that makes platform-based design a novel design method.

We argue for the importance of structuring precisely the platform layers and we discuss how to define formally the transitions from one platform to the next. In particular, we emphasize the interplay of top-down constraint propagation and bottom-up performance estimation while illustrating the notion of articulation point in the design process. In this context, we study the key role played by the API platform together with the micro-architecture platform in embedded system design. Also, we report on three applications of platform-based design: at the system-level, we discuss network platforms for communication protocol design and fault-tolerant platforms for the design of safety-critical applications; at the implementation level, we present analog platforms for mixed-signal integrated circuit design.

Key words: Platform-based design, derivative design, embedded systems, networks, protocol design, mixed-signal design, safety-critical applications.

1 Introduction

The motivations behind *Platform-Based Design* [29] originated from three major factors that characterize the evolution of the electronics industry:

- The *disaggregation of the electronic industry*, a phenomenon that began about a decade ago and has affected the structure of the electronics industry favoring the move from a vertically-oriented business model into a horizontally-oriented one. In the past, electronic system companies used to maintain full control of the production cycle from product definition to final manufacturing. Today, the identification of a new market opportunity, the definition of the detailed system specifications, the development of the components, the assembly of these components, and the manufacturing of the final product are tasks that are mostly performed by distinct organizations. In fact, the complexity of electronic designs and the number of technologies that must be mastered to bring to market winning products have forced electronic companies to focus on their core competence. In this scenario, the integration of the design chain becomes a serious problem whose most delicate aspects occur at the *hand-off points* from one company to another.
- The pressure for reducing *time-to-market* of electronics products in the presence of exponentially increasing complexity has forced designers to adopt methods that favor component re-use at all levels of abstraction. Furthermore, each organization that contributes a component to the final product naturally strives for a position that allows it to make continuous adjustments and accommodate last-minute engineering changes.
- The dramatic increase in *Non-Recurring Engineering (NRE) costs* due to
  - mask making at the Integrated Circuit (IC) implementation level (a set of masks for the 90 nanometer technology node costs more than two millions US dollars),
  - development of production plants (a new fab costs more than two billions dollars), and
  - design (a new generation micro-processor design requires more than 500 designers with all the associated costs in tools and infrastructure!)

  has created the necessity of correct-the-first-time designs.

Major delays in the introduction of new products have been the cause of severe economic problems for a number of companies. The cost of fabs have changed the landscape of IC manufacturing forcing companies to team up for developing new technology nodes (for example, the recent agreement among Motorola, Philips and ST Microelectronics and the creation of Renesas in Japan). The costs and complexity of ASIC designs has caused several system companies (for example, Ericsson) to reduce substantially or to eliminate completely their IC design efforts. Traditional paradigms in electronic system and IC design have to be revisited and re-adjusted or altogether abandoned.

The combination of these factors has caused several system companies to reduce substantially their ASIC design efforts. Traditional paradigms in electronic system and IC design have to be revisited and re-adjusted or altogether abandoned. Along the same line of reasoning, IC manufacturers are moving towards developing parts that have guaranteed high-volume production form a single mask set (or that are likely to have high-volume production, if successful) thus moving differentiation and optimization to reconfigurability and programmability.

Platform-based design has emerged over the years as a way of coping with the problems listed above. The term "platform" has been used in several domains: from service providers to system companies, from tier 1 suppliers to IC companies. In particular, IC companies have been very active lately to espouse platforms. The TI OMAP platform for cellular phones, the Philips Viper and Nexperia platforms for consumer electronics, the Intel Centrino platform for laptops, are but a few examples. Recently, Intel has been characterized by its CEO Ottellini as a "platform company".

As is often the case for fairly radical new approaches, the methodology emerged as a sequence of empirical rules and concepts but we have reached a point where a rigorous design process was needed together with supporting EDA environments and tools. Platform-based design

- lies the foundation for developing economically feasible design flows because it is a structured methodology that *theoretically limits the space of exploration, yet still achieves superior results in the fixed time constraints of the design*;
- provides a formal mechanism for identifying the most critical hand-off points in the design chain: the hand-off point between system companies and IC design companies and the one between IC design companies (or divisions) and IC manufacturing companies (or divisions) represent the *articulation points* of the overall design process;
- eliminates costly design iterations because it fosters *design re-use* at all abstraction levels thus enabling the design of an electronic product by assembling and configuring platform components in a rapid and reliable fashion;
- provides an intellectual framework for the complete electronic design process.

This paper presents the foundations of this discipline and outlines a variety of domains where the PBD principles can be applied. In particular, in Section 2 we define the main principles of PBD. Our goal is to provide a precise reference that may be used as the basis for reaching a common understanding in the electronic system and circuit design community. Then, we present the platforms that define the articulation points between system definition and implementation (Section 3). As a demonstration of applicability of the

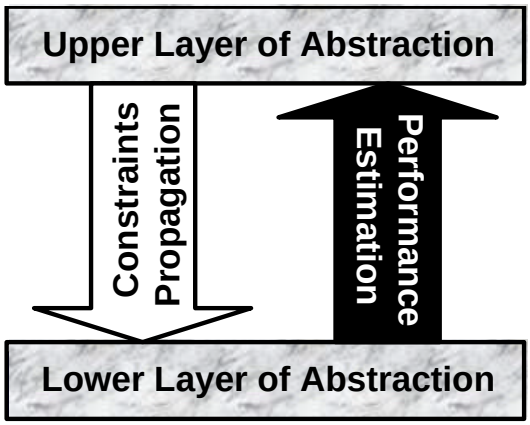


Fig. 1. Interactions Between Abstraction Layers.

Platform-Based Design paradigm to all levels of design. in the following sections, we show that platforms can be applied to very high levels of abstraction such as communication networks (Section 4) and fault-tolerant platforms for the design of safety-critical feedback-control systems (Section 5) as well as to low levels such as analog parts (Section 6), where performance is the main focus.

2 Platform-Based Design

The basic tenets of platform-based design are:

- The identification of design as a *meeting-in-the-middle process*, where successive refinements of specifications meet with abstractions of potential implementations.
- The identification of precisely defined layers where the refinement and abstraction processes take place. Each layer supports a design stage providing an opaque abstraction of lower layers that allows accurate performance estimations. This information is incorporated in appropriate parameters that annotate design choices at the present layer of abstraction. These layers of abstraction are called *platforms* to stress their role in the design process and their solidity.

A platform is 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 functional components. Each element of the library has a characterization in terms of performance parameters together with the functionality it can support. For every platform level, there is a set of methods used to map the upper layers of abstraction into the platform and a set of methods used to estimate performances of lower level

abstractions. As illustrated in Figure 1, the meeting-in-the-middle process is the combination of two efforts:

- **top-down**: map an instance of the top platform into an instance of the lower platform and propagate constraints;
- **bottom-up**: build a platform by defining the *library* that characterizes it and a performance abstraction (e.g., number of literals for tech. independent optimization, area and propagation delay for a cell in a standard cell library).

A *platform instance* is a set of architecture components that are selected from the library and whose parameters are set. Often the combination of two consecutive layers and their "filling" can be interpreted as a unique abstraction layer with an "upper" view, the top abstraction layer, and a "lower" view, the bottom layer. A *platform stack* is a pair of platforms, along with the tools and methods that are used to map the upper layer of abstraction into the lower level. Note that we can allow a platform stack to include several sub-stacks if we wish to span a large number of abstractions.

Platforms should be defined to eliminate large loop iterations for affordable designs: they should restrict design space via new forms of regularity and structure that surrender some design potential for lower cost and first-pass success. The library of function and communication components is the design space that we can explore at the appropriate level of abstraction.

Establishing the number, location, and components of intermediate platforms is the essence of platform-based design. In fact, designs with different requirements and specification may use different intermediate platforms, hence different layers of regularity and design-space constraints. A critical step of the platform-based design process is the definition of intermediate platforms to support *predictability*, which enables the abstraction of implementation detail to facilitate higher-level optimization, and *verifiability*, i.e. the ability to formally ensure correctness.

The trade-offs 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.

It is important to emphasize that the Platform-Based Design paradigm applies to all levels of design. 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. In the following sections, we will show that platforms can be applied to low levels of abstraction such as analog components, where flexibility is minimal and performance is the main focus, as well as to very high levels of abstraction such as networks, where platforms have to provide connectivity and services. In the former case platforms abstract hardware to provide (physical) implementation, while in the latter communication services abstract software layers (protocol) to provide global connectivity.

3 Platforms at the Articulation Points of the Design Process

As we mentioned above, the key to the application of the design principle is the careful definition of the platform layers. Platforms can be defined at several point of the design process. Some levels of abstraction are more important than others in the overall design trade-off space. In particular, the articulation point between system definition and implementation is a critical one for design quality and time. Indeed, the very notion of platform-based design originated at this point (see [3,10,15,18]). In [15,18,29], we have discovered that at this level there are indeed two distinct platforms forming a *system platform stack*. These need to be defined together with the methods and tools necessary to link them: a *(micro-)architecture platform* and an API platform. The API platform allows system designers to use the *services* that a (micro-)architecture offers them. In the world of personal computers, this concept is well known and is the key to the development of application software on different hardware that share some commonalities allowing the definition of a unique API.

3.1 (Micro-) Architecture Platforms

Integrated circuits used for embedded systems will most likely be developed as an instance of a particular *(micro-) architecture platform*. That is, rather than being assembled from a collection of independently developed blocks of silicon functionalities, they will be derived from a specific *family of micro-architectures*, possibly oriented toward a particular class of problems, that can be extended or reduced by the system developer. The elements of this family are a sort of "hardware denominator" that could be shared across multiple applications. Hence, an architecture platform is a family of micro-architectures that share some commonality, the library of components that are used to define the micro-architecture. Every element of the family can be obtained quickly

through the personalization of an appropriate set of parameters controlling the micro-architecture. Often the family may have additional constraints on the components of the library that can or should be used. For example, a particular micro-architecture platform may be characterized by the same programmable processor and the same interconnection scheme, while the peripherals and the memories of a specific implementation may be selected from the pre-designed library of components depending on the given application. Depending on the implementation platform that is chosen, each element of the family may still need to go through the standard manufacturing process including mask making. This approach then conjugates the need of saving design time with the optimization of the element of the family for the application at hand. Although it does not solve the mask cost issue directly, it should be noted that the mask cost problem is primarily due to generating multiple mask sets for multiple design spins, which is addressed by the architecture platform methodology.

The less constrained the platform, the more freedom a designer has in selecting an instance and the more potential there is for optimization, if time permits. However, more constraints mean stronger standards and easier addition of components to the library that defines the architecture platform (as with PC platforms). Note that the basic concept is similar to the cell-based design layout style, where regularity and the re-use of library elements allow faster design time at the expense of some optimality. The trade-off between design time and design "quality" needs to be kept in mind. The economics of the design problem must dictate the choice of design style. The higher the granularity of the library, the more leverage we have in shortening the design time. Given that the elements of the library are re-used, there is a strong incentive to optimize them. In fact, we argue that the "macro-cells" should be designed with great care and attention to area and performance. It makes also sense to offer a variation of cells with the same functionality but with implementations that differ in performance, area and power dissipation. Architecture platforms are, in general, characterized by (but not limited to) the presence of programmable components. Then, each of the platform instances that can be derived from the architecture platform maintains enough flexibility to support an application space that guarantees the production volumes required for economically viable manufacturing.

The library that defines the architecture platform may also contain re-configurable components. Reconfigurability comes in two flavors. With run-time reconfigurability, FPGA blocks can be customized by the user without the need of changing mask set, thus saving both design cost and fabrication cost. With design-time reconfigurability, where the silicon is still application-specific, only design time is reduced.

An *architecture platform instance* is derived from an architecture platform by choosing a set of components from the architecture platform library and/or by

setting parameters of re-configurable components of the library. The flexibility, or the capability of supporting different applications, of a platform instance is guaranteed by programmable components. Programmability will ultimately be of various forms. One is software programmability to indicate the presence of a microprocessor, DSP or any other software programmable component. Another is hardware programmability to indicate the presence of reconfigurable logic blocks such as FPGAs, whereby logic function can be changed by software tools without requiring a custom set of masks. Some of the new architecture and/or implementation platforms being offered on the market mix the two types into a single chip. For example, Triscend, Altera, and Xilinx are offering FPGA fabrics with embedded hard processors. Software programmability yields a more flexible solution, since modifying software is, in general, faster and cheaper than modifying FPGA personalities. On the other hand, logic functions mapped on FPGAs execute orders of magnitude faster and with much less power than the corresponding implementation as a software program. Thus, the trade-off here is between flexibility and performance.

3.2 API Platform

The concept of architecture platform by itself is not enough to achieve the level of application software re-use we require. The architecture platform has to be abstracted at a level where the application software "sees" a high-level interface to the hardware that we call Application Programm Interface (API) or Programmer Model. A software layer is used to perform this abstraction. This layer wraps the essential parts of the architecture platform:

- the programmable cores and the memory subsystem via a *real-time operating system (RTOS)*,
- the I/O subsystem via the *device drivers*, and
- the network connection via the *network communication subsystem*.

In our framework, the API is a unique abstract representation of the architecture platform via the software layer. Therefore, the application software can be re-used for every platform instance. Indeed the API is a platform itself that we can call the API platform. Of course, the higher the abstraction level at which a platform is defined, the more instances it contains. For example, to share source code, we need to have the same operating system but not necessarily the same instruction set, while to share binary code, we need to add the architectural constraints that force us to use the same ISA, thus greatly restricting the range of architectural choices.

The RTOS is responsible for the scheduling of the available computing resources and of the communication between them and the memory subsystem.

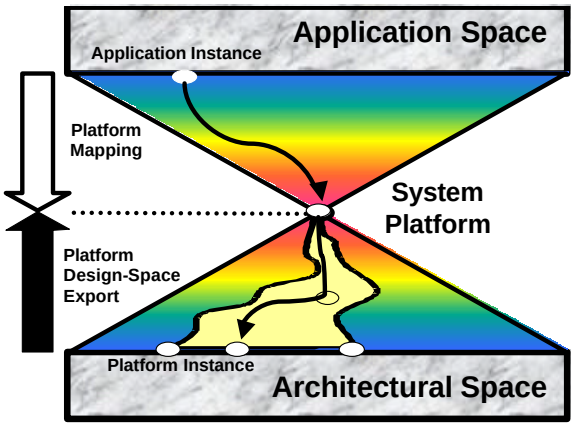


Fig. 2. System Platform Stack.

Note that in several embedded system applications, the available computing resources consist of a single microprocessor. In others, such as wireless handsets, the combination of a RISC microprocessor or controller and DSP has been used widely in 2G, now for 2.5G and 3G, and beyond. In set-top boxes, a RISC for control and a media processor have also been used. In general, we can imagine a multiple core architecture platform where the RTOS schedules software processes across different computing engines.

3.3 System Platform Stack

The basic idea of system platform-stack is captured in Figure 2. The vertex of the two cones represents the combination of the API and the architecture platform. A system designer maps its application into the abstract representation that "includes" a family of architectures that can be chosen to optimize cost, efficiency, energy consumption and flexibility. The mapping of the application into the actual architecture in the family specified by the API can be carried out, at least in part, automatically if a set of appropriate software tools (e.g., software synthesis, RTOS synthesis, device-driver synthesis) is available. It is clear that the synthesis tools have to be aware of the architecture features as well as of the API. This set of tools makes use of the software layer to go from the API platform to the architecture platform. Note that the system platform effectively decouples the application development process (the upper triangle) from the architecture implementation process (the lower triangle). Note also that, once we use the abstract definition of "API" as described above, we may obtain extreme cases such as traditional PC platforms on one side and full hardware implementation on the other. Of course, the programmer model for a full custom hardware solution is trivial since there is a one-to-one map between functions to be implemented and physical blocks that implement them. In the latter case, platform-based design amounts to adding to traditional design methodologies some higher level of abstractions.

4 Network Platforms

In distributed systems the design of the protocols and channels that support the communication among the system components is a difficult task due to the tight constraints on performances and cost. To make the communication design problem more manageable, designers usually decompose the communication function into distinct protocol layers, and design each layer separately. According to this approach, of which the OSI Reference Model is a particular instance, each protocol layer together with the lower layers defines a platform that provides communication services to the upper layers and to the application-level components. Identifying the most effective layered architecture for a given application requires one to solve a tradeoff between performances, which increase by minimizing the number of layers, and design manageability, which improve with the number of the intermediate steps. Present embedded system applications, due to their tight constraints, increasingly demand the co-design of protocol functions that in less-constrained applications are assigned to different layers and considered separately (e.g. cross-layer protocol design of MAC and routing protocols in sensor networks). The definition of an optimal layered architecture, the design of the correct functionality for each protocol layer, and the design space exploration for the choice of the physical implementation must be supported by tools and methodologies that allow to evaluate the performances and guarantee the satisfaction of the constraints after each step. For these reasons, we believe that the platform-based design principles and methodology provide the right framework to design communication networks. In this section, first, we formalize the concept of Network Platform. Then, we outline a methodology for selecting, composing and refining Network Platforms [30].

4.1 Definitions

A *Network Platform (NP)* is a library of resources that can be selected and composed together to form a Network Platform Instance (NPI) and support the interaction among a group of interacting components.

The structure of an NPI is defined abstracting computation resources as nodes and communication resources as links. Ports interface nodes with links or with the environment of the NPI. The structure of a node or a link is defined by its input and output ports, the structure of an NPI is defined by a set of nodes and the links connecting them.

The behaviors and the performances of an NPI are defined in terms of the type and the quality of the communication services it offers. We formalize the

behaviors of an NPI using the Tagged Signal Model [25]. NPI components are modeled as processes and events model the instances of the send and receive actions of the processes. An event is associated with a message which has a type and a value and with tags that specify attributes of the corresponding action instance (e.g. when it occurs in time). The set of behaviors of an NPI is defined by the intersection of the behaviors of the component processes.

A Network Platform Instance is defined as a tuple $NPI = (L, N, P, S)$, where

- $L = \{L_1, L_2, ...L_{Nl}\}$ is a set of directed links.
- $N = \{N_1, N_2, ...N_{Nn}\}$ is a set of nodes.
- $P = \{P_1, P_2, ...P_{Np}\}$ is a set of ports. A port P_i is a triple (N_i, L_i, d), where $N_i \in N$ is a node, $L_i \in L \cup Env$ is a link or the NPI environment and $d = in$ if it is an input port, $d = out$ if it is an output port. The ports that interface the NPI with the environment define the sets $P^{in} = \{(N_i, Env, in)\} \subseteq P, P^{out} = \{(N_i, Env, out)\} \subseteq P$.
- $S = \bigcap_{Nn+Nl} R_i$ is the set of behaviors, where R_i indicates the set of behaviors of a resource that can be a link in L or a node in N.

The basic services provided by an NPI are called *Communication Services (CS)*. A CS consists of a sequence of message exchanges through the NPI from its input to its output ports. A CS can be accessed by NPI users through the invocation of send and receive *primitives* whose instances are modeled as events. An *NPI Application Programming Interface (API)* consists of the set of methods that are invoked by the NPI users to access the CS. For the definition of an NPI API it is essential to specify not only the service primitives but also the type of CS they provide access to (e.g. reliable send, out-of-order delivery etc.). Formally, a Communication Service (CS) is a tuple ($\overline{P}^{in}, \overline{P}^{out}, M, E, h, g, <^t$), where $\overline{P}^{in} \subseteq P^{in}$ is a non-empty set of NPI input ports, $\overline{P}^{out} \subseteq P^{out}$ is a non-empty set of NPI output ports, M is a non-empty set of messages, E is a non-empty set of events, h is a mapping $h : E \rightarrow (\overline{P}^{in} \cup \overline{P}^{out})$ that associates each event with a port, g is a mapping $g : E \rightarrow M$ associating each event with a message, $<^t$ is a total order on the events in E.

A CS is defined in terms of the number of ports, that determine, for example, if it is a unicast, multicast or broadcast CS, the set M of messages representing the exchanged information, the set E including the events that are associated with the messages in M and model the instances of the send and receive methods invocations. The CS concept is useful to express the correlation among events, and explicit, for example, if two events are from the same source or are associated with the same message.

NPIs can be classified according to the number, the type, the quality and the cost of the CS they offer. Rather than in terms of event sequences, a CS is more conveniently described using *QoS parameters* like error rate, latency, throughput, jitter, and *cost parameters* like consumed power and manufacturing cost of the NPI components. QoS parameters can be simply defined using annotation functions that associate individual events with quantities, such as the time when an event occurs and the power consumed by an action. Hence, one can compare the values of pairs of input and output events associated with the same message to quantify the error rate, or compare the timestamp of events observed at the same port to compute the jitter. The most relevant QoS parameters are defined below using a notation where $e^{i,j} \in e^{M,(\overline{P}^{in} \cup \overline{P}^{out})}$ indicates an event carrying the i-th message and observed at the j-th port, $v(e)$ and $t(e)$ represents respectively the value of the message carried by event e and the timestamp of the action modeled by event e.

- **Delay:** The communication delay of a message is given by the difference between the timestamps of the input and output events carrying that message. Assuming that the i-th message is transferred from input port j_1 to output port j_2, the delay Δ_i of the i-th message, the average delay Δ_{Av} and the peak delay Δ_{Peak} are defined respectively as $\Delta_i = t(e^{j_2,i}) - t(e^{j_1,i}), \Delta_{Av} = \sum_{i=1}^{|M|} \frac{t(e^{j_2,i}) - t(e^{j_1,i})}{|M|}, \Delta_{Peak} = max_i\{t(e^{j_2,i}) - t(e^{j_1,i})\}$.
- **Throughput:** The throughput is given by the number of output events in an interval (t_0, t_1), i.e. the cardinality of the set $\Theta = \{e_i \in E | h(e_i) \in \overline{P}^{out}, t(e_i) \in (t_0, t_1)\}$.
- **Error rate:** The message error rate (MER) is given by the ratio between the number of lost or corrupted output events and the total number of input events. Given $LostM = \{e_i \in E | h(e_i) \in \overline{P}^{in}, \neg\exists e_j \in E \text{ s.t. } h(e_j) \in \overline{P}^{out} g(e_j) = g(e_i)\}$, $CorrM = \{e_i \in E | h(e_i) \in \overline{P}^{in}, \exists e_j \in E \text{ s.t. } h(e_j) \in \overline{P}^{out}, g(e_j) = g(e_i), v(e_j) \neq v(e_i)\}$ and $InM = \{e_i \in E | h(e_i) \in \overline{P}^{in}\}$, the message error rate $MER = \frac{|LostM| + |CorrM|}{|InM|}$. Using information on message encoding MER can be converted to Packet and Bit Error Rate.

The number of CS that an NPI can offer is large, so the concept of Class of Communication Services (CCS) is introduced to simplify the description of an NPI. CCS define a new abstraction (and therefore a platform) that groups together CS of similar type and quality. For example, a CCS may include all the CS that transfer a periodic stream of messages with no errors, another CCS all the CS that transfer a stream of input messages arriving at a bursty rate with a 1% error rate. CCS can be identified based on the type of messages (e.g. packets, audio samples, video pixels etc.), the input arrival pattern (e.g. periodic, bursty etc.), the range of QoS parameters. For each NPI supporting

multiple CS, there are several ways to group them into CCS. It is task of the NPI designer to identify the CCS and provide the proper abstractions to facilitate the use of the NPI.

4.3 Design of Network Platforms

The design methodology for NPs derive an NPI implementation by successive refinement from the specification of the behaviors of the interacting components and the declaration of the constraints that an NPI implementation must satisfy. The most abstract NPI is defined by a set of end-to-end direct logical links connecting pairs of interacting components. Communication refinement of an NPI defines at each step a more detailed NPI′ by replacing one or multiple links in the original NPI with a set of components or NPIs. During this process another NPI can be used as a resource to build other NPIs. A correct refinement procedure generates an NPI′ that provides CS equivalent to those offered by the original NPI with respect to the constraints defined at the upper level. A typical communication refinement step requires to define both the structure of the refined NPI′, i.e. its components and topology, and the behavior of these components, i.e. the protocols deployed at each node. One or more NP components (or predefined NPIs) are selected from a library and composed to create CS of better quality. Two types of compositions are possible. One type consists of choosing an NPI and extending it with a protocol layer to create CS at a higher level of abstraction (vertical composition). The other type is based on the concatenation of NPIs using an intermediate component called adapter (or gateway) that maps sequences of events between the ports being connected (horizontal composition).

5 Fault-Tolerant Platforms

The increasing role of embedded software in real-time feedback-control systems drives the demand for fault-tolerant design methodologies [24]. The aerospace and automotive industries offer many examples of systems whose failure may have unacceptable costs (financial, human or both). Designing cost-sensitive real-time control systems for safety-critical applications requires a careful analysis of the cost/coverage trade-offs of fault-tolerant solutions. This further complicates the difficult task of deploying the embedded software that implements the control algorithms on the execution platform. The latter is often distributed around the plant as it is typical, for instance, in automotive applications. In this section, we present a synthesis-based design methodology that relieves the designers from the burden of specifying detailed mechanisms for addressing the execution platform faults, while involving them in the defini-

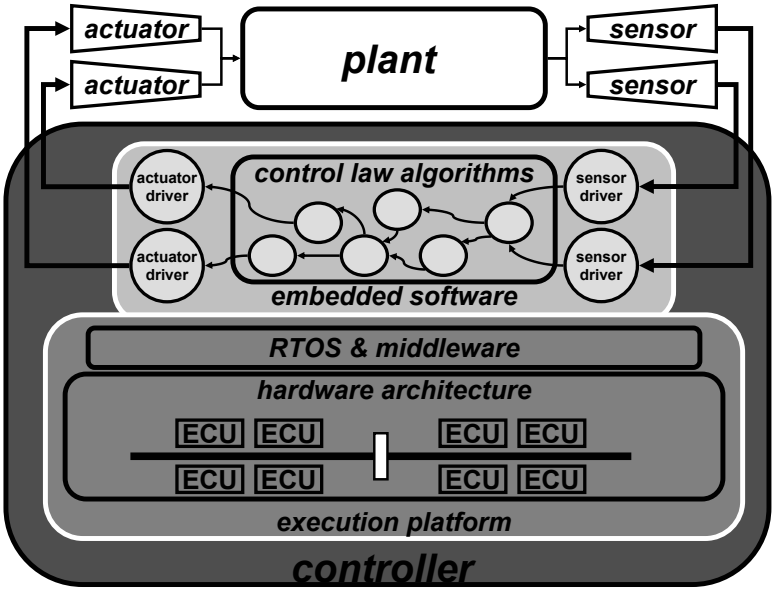


Fig. 3. A real-time control system.

tion of the overall fault-tolerance strategy. Thus, they can focus on addressing plant faults within their control algorithms, selecting the best components for the execution platform, and defining an accurate fault model. Our approach is centered on a new model of computation, Fault Tolerant Data Flows (FTDF), that enables the integration of formal validation techniques.

5.1 Types of Faults and Platform Redundancy

In a real-time feedback-control system, like the one of Figure 3, the controller interacts with the plant by means of sensors and actuators. A controller is a hardware-software system where the software algorithms that implement the control law run on an *execution platform*. An execution platform is a distributed system that is typically made of a software layer (RTOS, middleware services, ...) and a hardware layer (a set of processing elements, called electronic control units or ECUs, connected via communication channels like buses, crossbars, or rings). The design of these *heterogeneous reactive distributed systems* is made even more challenging by the requirement of making them resilient to faults. Technically, a fault is the cause of an error, an error is the part of the system state which may cause a failure, and a failure is the deviation of the system from the specification [23]. A *deviation from the specification* may be due to designers' mistakes ("bugs") or to accidents occurring while the system is operating. The latter can be classified in two categories that are relevant for feedback-control systems: *plant faults* and *execution platform faults*. Theoretically, all bugs can be eliminated before the system is deployed. In practice, they are minimized by using design environments that are based on precise models of computation (MoC), whose well-defined semantics en-

able formal validation techniques [1,12,13], (e.g., synchronous languages [6]). Instead, plant faults and execution platform faults must be dealt with on-line. Hence, they must be included in the specification of the system to be designed.

Plant faults, including faults in sensors and actuators, must be handled at the algorithmic level using estimation techniques and adaptive control methods. For instance, a *drive-by-wire system* might need to handle properly a tire puncture or the loss of one of the four brakes. Faults in the execution platform affect the computation, storage, and communication elements. For instance, a loss of power may turn off an ECU, momentarily or forever. System operation can be preserved in spite of platform faults if alternative resources supplying the essential functionality of the faulty one are available. Hence, the process of making the platform fault-tolerant usually involves the introduction of *redundancy* with obvious impact on the final cost. While the replication of a bus or the choice of a faster microprocessor may not affect sensibly the overall cost of a new airplane, their impact is quite significant for high-volume products like the ones of the automotive industry. The analysis of the trade-offs between higher redundancy and lower costs is a challenging HW-SW co-design task that designers of fault-tolerant systems for cost-sensitive applications must face in addition to the following two: (1) how to introduce redundancy, and (2) how to deploy the redundant design on a distributed execution platform. Since these two activities are both tedious and error prone, designers often rely on off-the-shelf solutions to address fault tolerance, like Time-Triggered Architecture (TTA) [20]. One of the main advantages of off-the-shelf solutions is that the application does not need to be aware of the fault tolerant mechanisms that are transparently provided by the architecture to cover the execution platform faults. Instead, designers may focus their attention on avoiding design bugs and tuning the control algorithms to address the plant faults. However, the rigidity of off-the-shelf solutions may lead to suboptimal results from a design cost viewpoint.

5.2 Fault-Tolerant Design Methodology

We present an interactive design methodology that involves designers in the exploration of the redundancy/cost trade-off [27]. To do so efficiently, we need automatic tools to bridge the different platforms in the system platform stack. In particular, we introduce automatic synthesis techniques that process simultaneously the algorithm specification, the characteristics of the chosen execution platform, and the corresponding *fault model*. Using this methodology, the designers focus on the control algorithms and the selection of the components and architecture for the execution platform. In particular, they also specify the relative criticality of each algorithm process. Based on a statistical analysis of the failure rates, which should be part of the characterization of the execution

platforms library, designers specify the expected set of platform faults, i.e. the fault model. Then, we use this information to (1) automatically deduce the necessary software process replication, (2) distribute each process on the execution platform, and (3) derive an optimal scheduling of the processes on each ECU to satisfy the overall timing constraints. Together, the three steps (replication, mapping, and scheduling) result in the automatic deployment of the embedded software on the distributed execution platform. Platforms export performance estimates, and we can determine for each control process its worst case execution time (WCET) on a given component [1] Then, we can use a set of verification tools to assess the quality of the deployment, most notably we have a static timing analysis tool to predict the worst case latency from sensors to actuators. When the final results do not satisfy the timing constraints for the control application, precise guidelines are returned to the designers who may use them to refine the control algorithms, modify the execution platform, and revisit the fault model. While being centered on a synthesis step, our approach does not exclude the use of pre-designed components, such as TTA modules, communication protocols like TTP [19] and fault-tolerant operating systems. These components can be part of a library of building blocks that the designer uses to further explore the fault-coverage/cost trade-off. Finally, the proposed methodology is founded on a new MoC, *fault tolerant data flow (FTDF)*, thus making it amenable to the integration of formal validation techniques. The corresponding API platform constists primarily of the FTDF MoC.

Fault Model. For the sake of simplicity we assume *fail silence*: components either provide correct results or do not provide any result at all. Recent work shows that fail-silent platforms can be realized with limited area overhead and virtually no performance penalty [4]. The fail silence assumption can be relaxed if invalid results are detected otherwise, as in the case of CRC-protected communication and voted computation [16]. However, it is important to notice that the proposed API platform (FTDF) is fault model independent. For instance, the presence of value errors, where majority voting is needed, can be accounted for in the implementation of the FTDF communication media (see Section 5.3). The same is true for Byzantine failures, where components can have any behavior, including malicious ones like coordinating to bring the system down to a failure [22]. In addition to the *type* of faults, a fault model also specifies the number (or even the mix) of faults to be tolerated [31]. A statistical analysis of the various components MTBFs (mean time between faults), their interactions and MTBR (mean time between repairs), should determine which subsystems have a compound MTBF that is so short to be of concern, and should be part of the platform component characterization. The use of *failure patterns* to capture effectively these interactions was proposed in [11], which is the basis of our approach [27].

[1] See [14] for some issues and techniques to estimate WCETs.

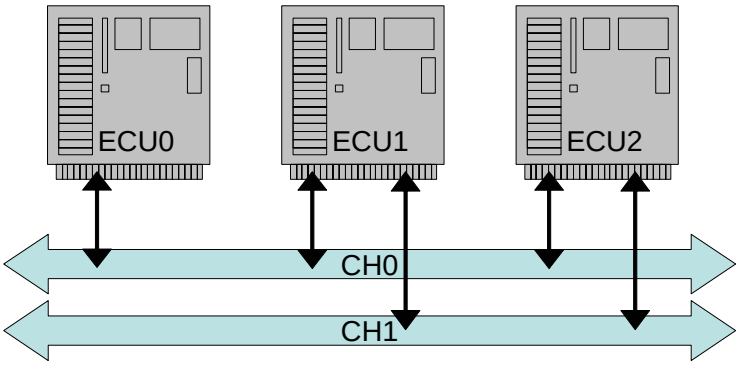


Fig. 4. A simple platform graph.

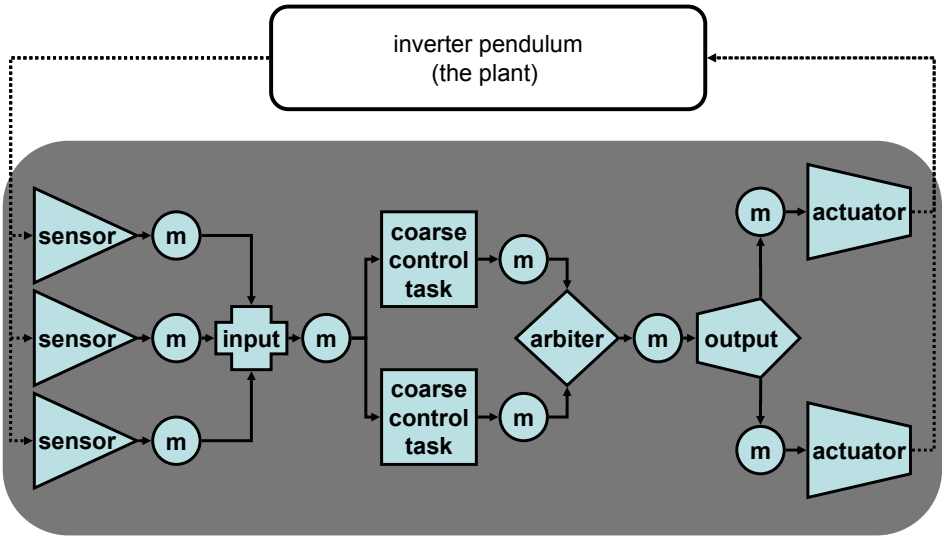


Fig. 5. Controlling an inverted pendulum.

Setup. Consider the feedback control system in Figure 3. The control system repeats the following sequence at each period $T_{\max}$: (1) sensors are sampled, (2) software routines are executed, and (3) actuators are updated with the newly-processed data. The actuator updates are applied to the plant at the end of the period to help minimize jitter, a well known technique in the real-time control community [17,32]. In order to guarantee correct operation, the worst-case execution time among all possible iterations, i.e. the worst case latency from sensors to actuators, must be smaller than the given period $T_{\max}$ (the real-time constraint), which is determined by the designers of the controller based on the characteristics of the application. Moreover, the critical subset of the control algorithms must be executed in spite of the specified platform faults.

Example. Figure 5 illustrates a FTDF graph for a paradigmatic feedback-control application, the inverted pendulum control system. The controller is described as a bipartite directed graph $\mathcal{G}$ where the vertices, called actors and communication media, represent software processes and data communication. Figure 4 illustrates a possible *platform graph* PG, where vertices represent ECUs and communication channels and edges describe their interconnections.

Platform Characteristics. Each vertex of PG is characterized by its failure rate and by its timing performance. A *failure pattern* is a subset of vertices of PG that may fail together during the same iteration, with a probability so high to be of concern. A set of failure patterns identify the fault scenarios to be tolerated. Based on the timing performance, we can determine the WCET of actors on the different ECUs and the worst case transmission time of data on channels. Graphs $\mathcal{G}$ and PG are related in two ways:

- **fault-tolerance binding:** for each failure pattern the execution of a corresponding subset of the actors of $\mathcal{G}$ must be guaranteed. This subset is identified a-priori based on the relative criticality assignment.
- **functional binding:** a set of mapping constraints and performance estimates indicate where on PG each vertex of $\mathcal{G}$ may be mapped and the corresponding WCET.

These bindings are the basis to derive a fault-tolerant deployment of $\mathcal{G}$ on PG. We use *software replication* to achieve fault tolerance: critical routines are replicated statically (at compile time) and executed on separate ECUs and the processed data are routed on multiple communication paths to withstand channel failures. In particular, to have a correct deployment in absence of faults, it is necessary that all actors and data communications are mapped to ECUs and channels in PG. Then, to have a correct fault-tolerant deployment, critical elements of $\mathcal{G}$ must be mapped to additional PG vertices to guarantee their correct and timely execution under any possible failure pattern in the fault model.

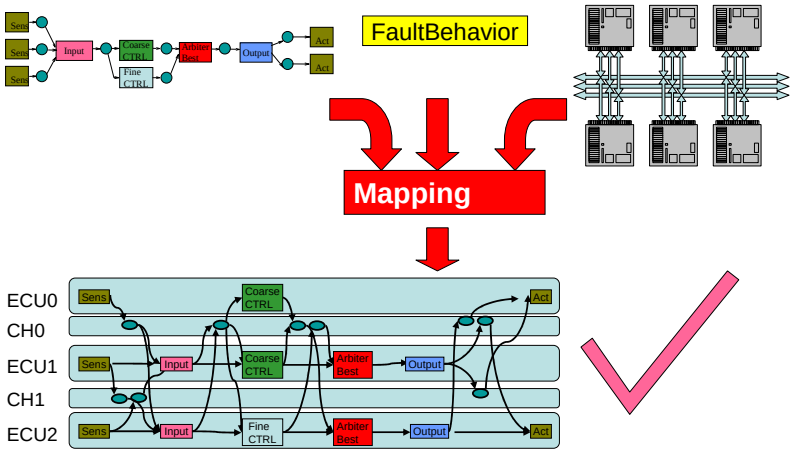


Fig. 6. Proposed Design Flow.

Design Flow. Using the interactive design flow of Figure 6 designers

- specify the controller (the top-left FTDF graph);
- assemble the execution platform (the top-right PG);
- specify a set of failure patterns (subsets of PG);
- specify the fault tolerance binding (fault behavior);
- specify the functional binding.

All this information contributes to specifying what the system should do and drive how it should be implemented. A synthesis tool automatically

- introduces redundancy in the FTDF graph;
- maps actors and their replicas onto PG;
- schedules their execution.

Finally, a verification tool checks whether the fault-tolerant behavior and the timing constraints are met. If no solution is found, the tool returns a *violation witness* that can be used to revisit the specification and to provide hints to the synthesis tool.

5.3 The API Platform (FTDF Primitives)

In this section we present the structure and general semantics of the FTDF MoC. The basic building blocks are actors and communication media. FTDF actors exchange data tokens at each iteration with synchronous semantics [6].

An actor belongs to one of six possible classes: sensors, actuators, inputs, outputs, tasks, arbiters. Sensor and actuator actors read and update respectively the sensor and actuator devices interacting with the plant. Input actors perform sensor fusion, output actors are used to balance the load on the actuators, while task actors are responsible for the computation workload. Arbiter

actors mix the values that come from actors with different criticality to reach to the same output actor (e.g. braking command and anti-lock braking system (ABS) [2]). Finally, state memories are connected to actors and operate as one-iteration delays. With a slight abuse of terminology the terms *state memory* and *memory actor* are used interchangeably in this paper.

Tokens. Each token consists of two fields: *Data*, the actual data being communicated; *Valid*, a boolean flag indicating the outcome of fault detection on this token. When Valid is "false" either no data is available for this iteration, or the available data is not correct. In both cases the Data field should be ignored. The Valid flag is just an abstraction of more concrete and robust fault detection implementations.

Communication Media. Communication occurs via unidirectional (possibly many-to-many) communication media. All replicas of the *same* source actor write to the same medium, and all destination actors read from it. Media act both as mergers and as repeaters sending the single "merged" result to all destinations. More formally, the medium provides the correct merged result or an invalid token if no correct result is determined.

Assuming *fail-silence*, merging amounts to selecting *any* of the valid results; assuming *value errors* majority voting is necessary; assuming Byzantine faults requires rounds of voting (see the consensus problem [5]). Communication media must be *distributed* to withstand platform faults. Typically, this means to have a repeater on each source ECU and a merger on each destination ECU (broadcasting communication channels helps reducing message traffic greatly). Using communication media, actors always receive exactly one token per input and the application behavior is independent of the type of platform faults. The transmission of tokens is initiated by the *active elements*: regular actors and memory actors.

Regular Actors. When an actor fires, its sequential code is executed. This code is: *stateless* (state must be stored in memory actors), *deterministic* (identical inputs generates identical outputs), *non-blocking* (once fired, it does not await for further tokens, data, or signals from other actors) and *terminating* (bounded WCET). The firing rule specifies which subsets of input tokens must be valid to fire the actor, typically all of them (*and firing rule*). However, the designer may need to specify *partial* firing rules for input and arbiter actors. For example, an input actor reading data from three sensors may produce a valid result even when one of the sensors cannot deliver data (e.g. when the ECU where the sensor is mapped is faulty).

[2] We advocate running non-safety critical tasks, e.g. door controllers, on separate HW. However some performance enhancement tasks, e.g. side-wind compensation, may share sensors and actuators with critical tasks (steer-by-wire). It may be profitable to have them share the execution platform as well.

Memory Actors (State Memories). A memory provides its state at the beginning of an iteration and has a source actor, possibly replicated, that updates its state at every iteration. State memories are analogous to latches in a sequential digital circuit: they store the results produced during the current iteration for use in the next one.

Finally FTDF graphs can express redundancy, i.e. one or more actors may be replicated. All the replicas of an actor $v \in A$ are denoted by $\mathcal{R}(v) \subset A$. Note that any two actors in $\mathcal{R}(v)$ are of the same type and must compute the same function. This basic condition is motivated in Section 5.5 where replica determinism is discussed. Note that the replication of sensors and actuators is not performed automatically because they may have a major impact on cost, we discuss the implications of this choice in [27].

5.4 Fault-Tolerant Deployment

The result of the synthesis is a redundant mapping $\mathcal{L}$, i.e. an association of elements of the FTDF network to multiple elements of the execution platform, and for each element in the execution platform a schedule $\mathcal{S}$, i.e. a total order in which actors should be executed and data should be transmitted. A pair $(\mathcal{L}, \mathcal{S})$ is called a *deployment*. To avoid deadlocks, the total orders defined by $\mathcal{S}$ must be compatible with the partial order in $\mathcal{L}$, which in turn derives directly from the partial order in which the FTDF actors in the application must be executed. To avoid causality problems, memory actors are scheduled before any other actor, thus using the results of the previous iteration. Schedules based on total orders are called static: there are no run-time decisions to make, each ECU and each channel controller simply follows the schedule. However, in the context of a faulty execution platform an actor may not receive enough valid inputs to fire and this may lead to starvation. This problem is solved by *skipping* an actor if it cannot fire and by skipping a communication if no data is available [11].

5.5 Replica Determinism

Given a mapping $\mathcal{L}$it is important to preserve *replica determinism*: if two replicas of a same actor fire, they produce identical results. For general MoCs the order of arrival of results must also be the same for all replicas. Synchrony of FTDF makes this check unnecessary. Clearly the execution platform must contain the implementation of a synchronization algorithm [21].

Replica determinism in FTDF can be achieved enforcing two conditions: (1) all replicas compute the same function, and (2) for any failure pattern, if

two replicas get a firing subset of inputs they get the *same* subset of inputs. Condition (1) is enforced by construction by allowing only identical replicas. Condition (2) amounts to a consensus problem and it can either be checked at run-time (like for Byzantine agreement rounds of voting), or it can be analyzed statically at compile time (if the fault model is milder). Our interest in detectably faulty execution platforms makes the latter approach appear more promising and economical. Condition (2) is trivially true for all actors with the "AND firing rule". For input and arbiter actors the condition must be checked and enforced [27].

6 Analog Platforms

Emerging applications such as multimedia devices (video cell phones, digital cameras, wireless PDAs to mention but a few) are driving the SoC market towards the integration of analog components in almost every system. Today, system-level analog design is a design process dominated by heuristics. Given a set of specifications/requirements that describes the system to be realized, the selection of a feasible (let alone optimal) implementation architecture comes mainly out of experience. Usually, what is achieved is just a feasible point at the system level, while optimality is sought locally at the circuit level. This practice is caused by the number of second order effect that are very hard to deal with at high level without actually designing the circuit. Platform-based design can provide the necessary insight to develop a methodology for analog components that takes into consideration system level specifications and can choose among a set of possible solutions including digital approaches wherever it is feasible to do so. If the "productivity gap" between analog and digital components is not overcome, time-to-market and design quality of SoC will be seriously affected by the small analog sections required to interface with the real world. Moreover, SoC designs will expose system level explorations that would be severely limited if the analog section is not provided with a proper abstraction level that allows system performance estimation in an efficient way and across the analog/digital boundary. Therefore, there is a strong need to develop more abstract design techniques that can encapsulate analog design into a methodology that could shorten design time without compromising the quality of the solutions, leading to a *hardware/software/analog* co-design paradigm for embedded systems

6.1 Definitions

The platform abstraction process can be extended to analog components in a very natural way. Deriving behavioral and performance models, however, is

more involved due to the tight dependency of analog components on device physics that requires the use of continuous mathematics for modeling the relations among design variables. Formally, an Analog Platform (AP) consists of a set of components, each decorated with:

- a set of input variables $u \in \mathcal{U}$, a set of output (performance) variables $y \in \mathcal{Y}$, a set of "internal" variables (including state variables) $x \in \mathcal{X}$, a set of configuration parameters $\kappa \in \mathcal{K}$; some parameters take values in a continuous space, some take values in a discrete set, for example when they encode the selection of a particular alternative;
- a *behavioral model* that expresses the behavior of the component represented implicitly as $\mathcal{F}(u, y, x, \kappa) = 0$, where $\mathcal{F}(\cdot)$, may include integro-differential components; in general, this set determines uniquely x and y given u and κ. Note that the variables considered here can be function of time and that the functional $\mathcal{F}$ includes constraints on the set of variables (for example, the initial conditions on the state variables).
- a *feasible performance model*. Let $\phi_y(u, \kappa)$ denote the map that computes the performance y corresponding to a particular value of u and κ by solving the behavioral model. The set of feasible analog performance (such as gain, distortion, power), is the set described by the relation $\mathcal{P}(y(u)) = 1 \Leftrightarrow \exists \kappa', y(u) = \phi_y(\kappa', u)$.
- *validity laws* $\mathcal{L}(u, y, x, \kappa) \leq 0$ i.e., constraints (or assumptions) on the variables and parameters of the component that define the range of the variables for which the behavioral and performance models are valid.

Note that there is no real need to define the feasible performance model since the necessary information is all contained in the behavioral model. We prefer to keep them separate because of the use we make of them in explaining our approach.

At the circuit level of abstraction, the behavioral models are the circuit equations with x being the voltages, currents and charges, y being a subset of x and/or a function of x and κ when they express performance figures such as power or gain. To compute performance models, we need to solve the behavioral models that implies solving ordinary differential equations, a time consuming task. In the past, methods to approximate the relation between y and κ (the design variables) with an explicit function were proposed. In general, to compute this approximation, a number of evaluations of the behavioral model for a number of parameters κ is performed (by simulation, for example) and then an interpolation or approximation scheme is used to derive the approximation to the map ϕ_y. We see in Section 6.2 how to compute an approximation to the feasible performance set directly.

Example — Considering an OTA for an arbitrary application, we can start building a platform from the circuit level by defining:

- $\mathcal{U}$ as the set of all possible input voltages $V_{in}(t)$ s.t. $|V_{in}| < 100$ mV and bandwidth $V_{in} < 3$ MHz; $\mathcal{Y}$ as the space of vectors $\{V_{out}(t), \text{gain}, \text{IIP3}, r_{out}\}$ (IIP3 is the third order intermodulation intercept point referred to the input, r_{out} is the output resistance) $\mathcal{X}$ the set of all internal current and voltages, and $\mathcal{K}$ the set of transistor sizings.
- for a transistor level component, the behavioral model $\mathcal{F}$ consists of the solution of the circuit equations, e.g. through a circuit simulator.
- $\phi_y(u, \kappa)$ as the set of all possible y;
- validity laws $\mathcal{L}$ are obtained from Kirchoff laws when composing individual transistors and other constraints, e.g. maximum power ratings of breakdown voltages.

We can build a higher level (level 1) OpAmp platform where:

- $\mathcal{U}^1$ is the same, $\mathcal{Y}^1$ is the output voltage of the OpAmp, $\mathcal{X}$ is empty, $\mathcal{K}^1$ consists of possible {gain, IIP3, r_{out}} triples (thus it is a projection of $\mathcal{Y}^0$);
- $\mathcal{F}^1$ can be expressed in explicit form,

$$\begin{cases} y_1(t) = h(t) \otimes (a_1 \cdot u(t) + a_3 \cdot u(t)^3) + \text{noise} \\ y_2 = a_1; y_3 = \sqrt{\frac{4}{3} \cdot \frac{a_1}{a_3}} \end{cases} \tag{1}$$

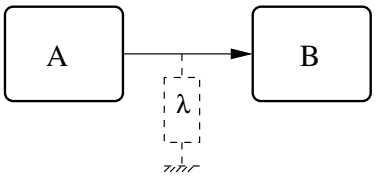
- ϕ_y is the set of possible y;
- there are no validity constraints, $\mathcal{L} < 0$ always.

■

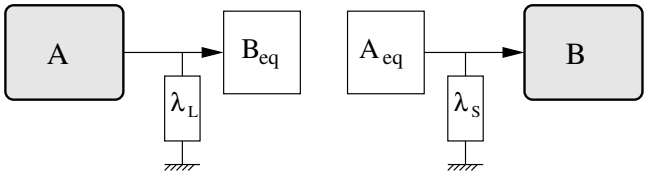
When a platform instance is considered, we have to compose the models of the components to obtain the corresponding models for the instance. The platform instance is then characterized by

- a set of internal variables of the platform $\xi = [\xi_1, \xi_2, ..., \xi_n] \in \Xi$,
- a set of inputs of the platform, $h \in \mathcal{H}$
- a set of performances $\upsilon \in \Upsilon$,
- a set of parameters $\zeta \in \mathcal{Z}$.

The variable names are different from the names used to denote the variables of the components to stress that there may be situations where some of the component variables change roles (for example, an input variable of one component may become an internal variable; a new parameter can be identified in the platform instance that is not visible or useful at the component level). To compose the models, we have to include in the platform the composition rules. The *legal compositions* are characterized by the interconnect equations that specify which variables are shared when composing components and by constraints that define when the composition is indeed possible. These constraints may involve range of variables as well as nonlinear relations among variables.



a) Platform composition A driving B with interface paramater λ



b) Characterization setup for platform A and B

Fig. 7. Interface parameter λ during composition A-B and characterization of A and B.

Formally, a connection is establishing a pairwise equality between internal variables for example $\xi_i = \xi_j$, inputs and performance; we denote the set of *interconnect relations* with $c(h, \xi, \zeta, \kappa) = 0$ that are in general a set of linear equalities. The *composition constraints* are denoted by $\mathcal{L}(h, \xi, \upsilon, \zeta) \leq 0$, that are in general, non linear inequalities. Note that in the platform instance all internal variables of the components are present as well as all input variables. In addition, there is no internal or input variable of the platform instance that is not an internal or input variable of one of the components. The behavioral model of the platform instance is the union of all behavioral models of the components conjoined with the interconnect relations. The validity laws are the conjunction of the validity laws of the components and of the composition constraints. The feasible performance model may be defined anew on the platform instance but it may also be obtained by composition of the performance models of the components. There is an important and interesting case when the composition may be done considering only the feasible performance models of the components obtained by appropriate approximation techniques. In this case, the composition constraints assume the semantics of defining when the performance models may be composed. For example, if we indicate with λ the parameters related to internal nodes that characterizes the interface in Fig. 7a) (e.g. input/output impedance in the linear case), then matching between λ has to be enforced during composition. In fact, both $\mathcal{P}_A$ and $\mathcal{P}_B$ were characterized with specific λs (Fig. 7b)), so $\mathcal{L}$ has to constrain $A - B$ composition consistently with performance models. In this case, an architectural exploration step consisting of forming different platform instances out of the component library and evaluating them, can be performed very quickly albeit possibly with restrictions on the space of the considered instances caused by the composition constraints.

Example — We can build a level 2 platform platform consisting of an OpAmp (OA) and a unity gain buffer following it (UB, the reader can easily find a proper definition for it), then we can define a higher level OpAmp platform component so that:

- $\xi_1 = V_{in}^{OA}$, $\xi_2 = V_{out}^{OA}$, $\xi_3 = V_{in}^{UB}$, $\xi_4 = V_{in}^{UB}$ and connect them in series specifying $\xi_2 = \xi_3$;
- h connected to ξ_1 is the set of input voltages $V_{in}(t)$;
- Υ is the space of $\upsilon_1(t)$, the cascade response in time, $\upsilon_2 = gain$, $\upsilon_3 = IIP3$. In this case υ_2 immediately equals y_2^{OA}, while υ_3 is a non linear function of y^{OA} and y^{UB};
- $\mathcal{Z}$ consists of all parameters specifying a platform instance, in this case we may have $\mathcal{Z} = \mathcal{Y}_{OA} \cup \mathcal{Y}_{UB}$.
- a platform instance composability law $\mathcal{L}$ requires that the load impedance $Z_L > 100 r_{out}$ both at the output of the OpAmp and the unity buffer.

■

6.2 Building Performance Models

An important part of the methodology is obtaining performance models. We already mentioned that we need to approximate the set $\hat{\mathcal{Y}}$ explicitly eliminating the dependence on the internal variables x. To do so a simulation-based approach is proposed.

6.2.1 Performance Model Approximation

In general terms, simulation maps a configuration set (typically connected) $\mathcal{K}$ into a performance set in $\mathcal{Y}$, thus establishing a relation among points belonging to the mapped set. Classic regression schemes provides an efficient approximation to the mapping function $\phi(\cdot)$, however our approach requires dealing with performance data in two different ways. The first one, referred to as performance model $\mathcal{P}$, allows discriminating between points in $\hat{\mathcal{Y}}$ and points in $\mathcal{Y} \backslash \hat{\mathcal{Y}}$. A second one, $\mu(\cdot) = \phi^{-1}(\cdot)$, implementing the inverse mapping from $\hat{\mathcal{Y}}$ into $\mathcal{K}$, used to map down from a higher-level platform layer to a lower one. However, fundamental issues (i.e. $\phi(\cdot)$ being an invertible function) and accuracy issues (a regression from $\mathbb{R}^m$ into $\mathbb{R}^n$) suggest a table-lookup implementation for $\mu(\cdot)$, possibly followed by a local optimization phase to improve mapping. Therefore, we will mainly focus on basic performance models $\mathcal{P}$.

The set $\hat{\mathcal{Y}} \subset \mathcal{Y}$ defines a relation in $\mathcal{Y}$ denoted with $\mathcal{P}$. We use Support Vector Machines (SVMs) as a way of approximating the performance relation $\mathcal{P}$ [8].

SVMs provide approximating functions of the form

$$f(x) = \text{sign}(\sum_i \alpha_i e^{-\gamma|x-x_i|^2} - \rho) \tag{2}$$

where x is the vector to be classified, x_i are observed vectors, α_is are weighting multipliers, ρ is a biasing constant and γ is a parameter controlling the fit of the approximation. More specifically, SVMs exploit mapping to Hilbert spaces so that hyperplanes can be exploited to perform classification. Mapping to high dimensional spaces is achieved through *kernel* functions, so that a kernel $k(\kappa, \cdot)$ is associated at each point κ. Since the only general assumption we can make on $\phi(\cdot)$ is continuity and on $\mathcal{K}$ is connectivity [3], we can only deduce that $\hat{\mathcal{Y}}$ is connected as well. Therefore, the radial basis function Gaussian kernel is chosen, $k(\kappa, \kappa') = e^{-\gamma \cdot \|\kappa - \kappa'\|^2}$, where γ is a parameter of the kernel and controls the "width" of the kernel function around κ. We resort to a particular formulation of SVMs known as one-class SVM where an optimal hyperplane is determined to separate data from the origin. The optimal hyperplane be computed very efficiently through a quadratic problem, as detailed in [28].

6.2.2 Optimizing the approximation process

Sampling schemes for approximating unknown functions are exponentially dependent on the size of the function support. In the case of circuit, none but very simple circuits could be realistically characterized in this way. Fortunately, there is no need to sample the entire space $\mathcal{K}$ since we can use additional information obtained from design considerations to exclude parts of the parameter space. The set of "interesting" parameters is delimited by a set of constraints of two types:

- *topological* — constraints derived from the use of particular circuit structures, such as two stacked transistor sharing the same current or a set of V_{DS} summing to zero;
- *physical* — constraints induced by device physics, such as V_{GS}-V_{DS} relation to enforce saturation or g_m-I_D relations;
- *performance* — constraints on circuit performances, such as minimum gain or minimum phase margin, that can be achieved.

Additional constraints can be added as designers' understanding of circuit improves. The more constraints we add, the smaller the interesting configuration space $\mathcal{K}$. However, if a constraint is tight, i.e., it either defines lower dimensional manifolds for example when the constraint is an equality, or the measure of the manifold is small, the more likely it is to introduce some bias in the

[3] more in general, a union of a finite number of connected sets

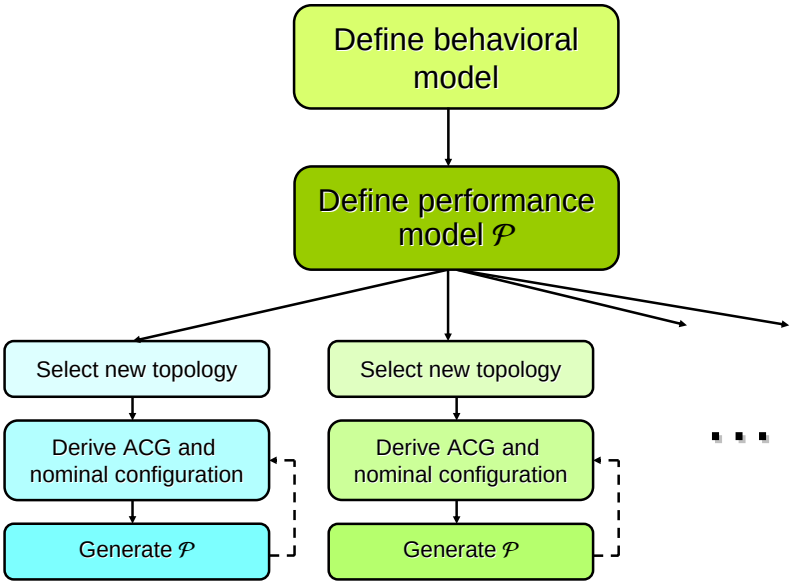


Fig. 8. Bottom-Up phase for generating Analog Platforms

sampling mechanism because of the difficulty in selecting points in these manifolds. To eliminate this ill-conditioning effect, we "relax" these constraints to include a larger set of interesting parameters. We adopt a statistical means of relaxing constraints by introducing random errors with the aim of dithering systematic errors and recovering accuracy in a statistical sense. Given an equality constraint $f(\kappa) = 0$ and its approximation $\tilde{f}(\kappa) = 0$, we derive a relaxation $|\tilde{f}(\kappa)| \leq \epsilon$. For each constraint f some statistics have to be gathered on ϵ so as to minimize the overhead on the size of $\mathcal{K}$ for introducing it.

Once we have this set of constraints, we need to take them into account to define the space of *interesting parameters*. Analog Constraint Graphs (ACGs) are introduced as a bipartite graph representation of configuration constraints. One set of nodes corresponds to equations, the other to variables κ. Bipartite graphs are a common form for dealing with system of equations [9]. A maximal bipartite matching in the ACG is used to compute an evaluation order for equations that is then translated into executable code capable of generating configurations in $\mathcal{K}$ *by construction*. In our experience, even with reasonably straightforward constraints, ratios of the order of 10^{-6} were observed for $\frac{\text{size}(\tilde{\mathcal{K}})}{\text{size}(\mathcal{K})}$ with $\mathcal{K} \subset \mathbb{R}^{16}$.

When we deal with the intersection of achievable performance and performance constraints in the top-down phase of the design, we can add to the set of constraints we use to restrict the sampling space the performance constraints so that the results reported above are even more impressive.

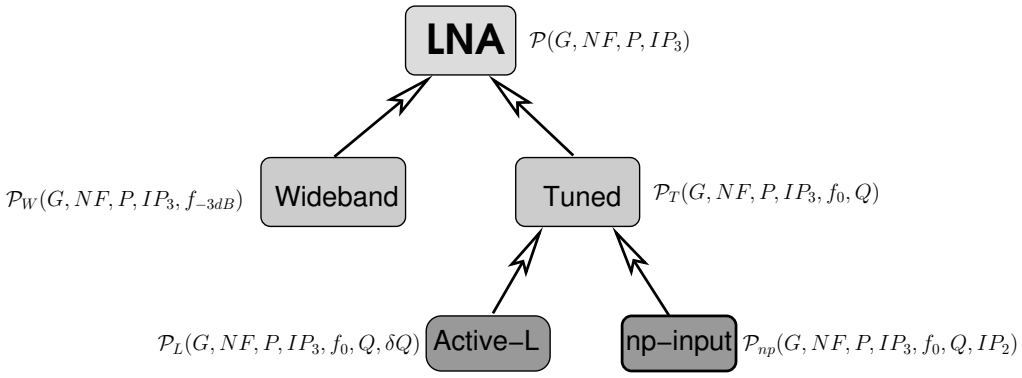


Fig. 9. Sample model hierarchy for an LNA platform. The root node provides performance constraints for a generic LNA, which is then refined by more detailed $\mathcal{P}$ for specific classes of LNAs.

6.3 Mixed-Signal Design Flow with Platforms

The essence of platform-based design is building a set of abstractions that facilitate the design of complex systems by a successive refinement/abstraction process. The abstraction takes place when an existing set of components forming a platform at a given level of abstraction is elevated to a higher level by building appropriate behavioral and performance models together with the appropriate validity laws. This process can take either components at a level of abstraction and abstract each of them, or abstract a set of platform instances. Since both platform instances and platform components are described at the same level of abstraction the process is essentially the same. What changes is the exploration approach. On the other side of the coin, the top-down phase progresses through refinement. Design goals are captured as constraints and cost function. At the highest level of abstraction, the constraints are intersected with the feasible performance set to identify the set of achievable performance that satisfy design constraints. The cost function is then optimized with respect to the parameters of the platform instances at the highest level of abstraction ensuring they lie in the intersection of the constraint and the feasible performance set. This constrained optimization problem yields a point in the feasible performance space and in the parameter space for the platform instances at the highest level of abstraction. Using the inverse of the abstraction map ϕ_y, these points are mapped back at a lower level of abstraction where the process is repeated to yield a new point in the achievable performance set and in the parameter space until we reach a level where the circuit diagrams and even a layout is available. If the abstraction map is a conservative map, then every time we map down, we always find a consistent solution that can be achieved. Hence the design process can be shortened considerably. The crux of the matter is how many feasible points are not considered because of the conservative approximation. Thus the usual design speed versus design quality trade-off has to be explored.

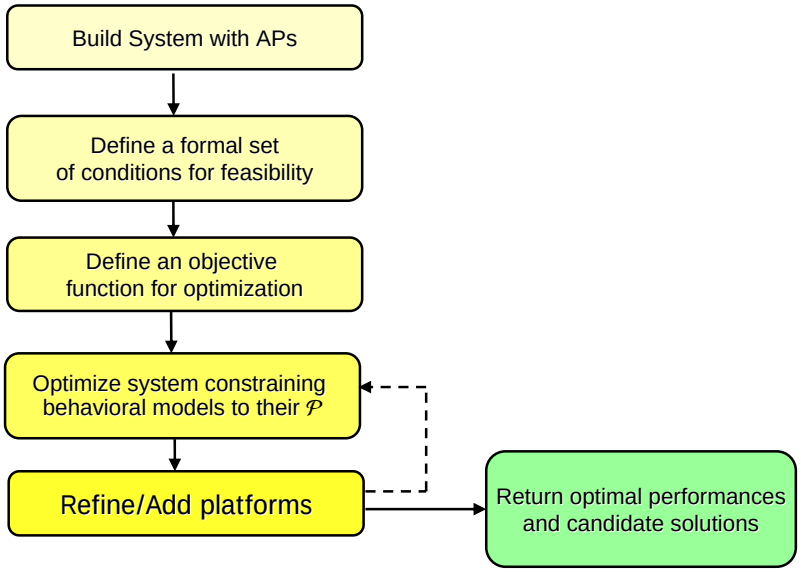


Fig. 10. Top-Down phase for analog design space exploration

In mathematical terms, the bottom-up phase consists of defining an abstraction ψ^l that maps the inputs, performance, internal variables, parameters, behavioral and performance models, and validity laws of a component or platform instance at level l into the corresponding objects at the next level $(l+1)$. The map is *conservative* if all feasible performance vectors $\hat{y}^{l+1}$ correspond to feasible performance vectors $\hat{y}^l$. Note that if approximations are involved in defining the models and the maps, this is not necessarily true, i.e., abstraction maps may be non conservative. In other words, a feasible performance vector at level $l+1$ may not correspond to a feasible one at level l. A simplified diagram of the bottom-up phase for circuit level components is shown in Fig. 8. For each library component, we define a behavioral model and a performance model. Then, a suitably topology is determined, an ACG is derived to constrain the configuration space $\mathcal{K}$ and a performance model is generated. This phase can be iterated, leading to a library that can span multiple topologies as reported in Fig. 9.

The top-down phase then proceeds formulating a top-level design problem as an optimization problem with a cost function $\mathcal{C}(y_{top})$ and a set of constraints defined in the $\mathcal{Y}_{top}$ space, $g_{top}(y_{top}) \leq 0$ that identifies a feasible set in $\mathcal{Y}_{top}$. The complete optimization problem has to include the set $\hat{\mathcal{Y}}_{top}$ that defines the set of achievable performance at the top level. The intersection of the two sets define the feasible set for the optimization process. The result of the process is a ${y_{top}}^{opt}$. Then the process is to map back the selected point to the lower levels of the hierarchy. If the abstractions are conservative, the top-down process is straightforward. Otherwise, at each level of the hierarchy, we

have to verify using the performance models, the behavioral models and the validity laws. In some cases, a better design may be obtained by introducing in the top-down phase cost functions and constraints that are defined only at a particular abstraction level. In this case, the space of achievable performances intersected with this new set of constraints defines the search space for the optimization process. At times, it is more convenient to project down the cost function and the constraints of the higher-level abstraction to the next level down. In this case, then the search space is the result of the intersection of three sets in the performance space and the cost function is a combination of the projected cost function and the one defined at this level. A flow chart summarizing the top-down flow with platforms is shown in Fig. 10. In Fig. 11 the set of configurations evaluated during an optimization run for the UMTS frontend in [7] is reported visualizing how multiple topologies are exploited in selecting optimal points.

The peculiarity of a platform approach to mixed signal design resides in the accurate performance model constraints $\mathcal{P}$ that propagate to the top-level architecture related constraints. For example, a platform stack can be built where multiple analog implementation architectures are presented at a common level of abstraction together with digital enhancement platforms (possibly including several algorithms and hardware architectures), each component being annotated with feasible performance spaces. Solving the system design problem at the top level where the platforms contain both analog and digital components, allows selecting optimal platform instances in terms of analog and digital solutions, comparing how different digital solutions interact with different analog topologies and finally selecting the best tradeoff.

The final verification step is also greatly simplified by the platform approach since, at the end, models and performances used in the top-down phase were obtained with a bottom-up scheme. Therefore, a consistency check of models, performances and composition effects is all that is required at a hierarchical level, followed by more costly, low-level simulations that check for possible important effects that were neglected when characterizing the platform.

6.4 Reconfigurable platforms

Analog platforms can also be used to model programmable fabrics. In the digital implementation platform domain, FPGAs provide a very intuitive example of platform, for example including microprocessors on chip. The appearance of Field Programmable Analog Arrays [26] constitutes a new attempt to build reconfigurable Analog Platform. A platform stack can be built by exploiting the software tools that allow mapping complex functionalities (filters, amplifiers, triggers and so on) directly on the array. The top level platform,

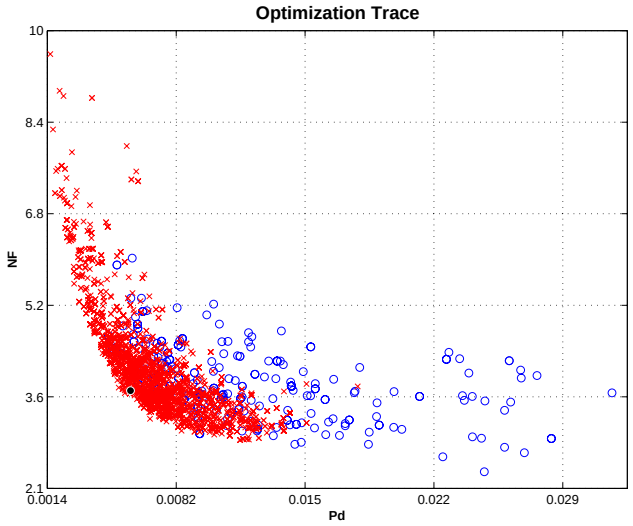


Fig. 11. Example of architecture selection during top-down phase. In the picture, an LNA is being selected. Circles correspond to architecture 1 instances, crosses to architecture 2 instances. The black circle is the optimal LNA configuration. It can be inferred that after an initial exploration phase alternating both topologies, simulated annealing finally focuses on the architecture 1 to converge.

then, provides an API to map and configure analog functionalities, exposing analog hardware at the software level. By exploiting this abstraction, not only design exploration is greatly simplified, but new synergies between higher layers and analog components can be leveraged to further increase the flexibility/reconfigurability and optimize the system. From this abstraction level, implementing a functionality with digital signal processing (FPGA) or analog processing (FPAA) becomes subject to system level optimization while exposing the same abstract interface. Moreover, very interesting tradeoffs can be explored exploiting different partitionings between analog and digital components and leveraging the reconfigurability of the FPAA. For example, limited analog performances can be mitigated by proper reconfiguration of the FPAA, so that a tight interaction between analog and digital subsystems can provide a new optimum from the system level perspective.

7 Concluding Remarks

We defined platform-based design as an all-encompassing intellectual framework in which scientific research, design tool development, and design practices can be embedded and justified. In our definition, a platform is simply an abstraction layer that hides the details of the several possible implementation refinements of the underlying layer. Platform-based design allows designers to trade-off various components of manufacturing, NRE and design costs while sacrificing as little as possible potential design performance. We presented

examples of these concepts at different key articulation points of the design process, including system platforms as composed of two platforms (micro-architecture and API), network platforms, and analog platforms.

This concept can be used to interpret traditional design steps in ASIC development such as synthesis and layout. In fact, logic synthesis takes a level of abstraction consisting of HDL representation (HDL platform) and maps it into a set of gates that are defined in a library. The library itself is the gate-level platform. The logic synthesis tools are the mapping methods that select a platform instance (a particular netlist of gates that implements the functionality described at the HDL platform level) according to a cost function defined on the parameters that characterize the quality of the elements of the library in view of the overall design goals. The present difficulties in achieving timing closure in this flow indicate the need for a different set of characterization parameters for the implementation platform. In fact, in the gate-level platform the cost associated to the selection of a particular interconnection among gates is not reflected, a major problem since the performance of the final implementation depend critically on this. The present solution of making a larger step across platforms by mixing mapping tools such as logic synthesis, placement and routing may not be the right one. Instead, a larger pay-off could be had by changing levels of abstractions and including better parametrization of the implementation platform.

We argued in this paper that the value of PBD can be multiplied by providing an appropriate set of tools and a general framework where platforms can be *formally* defined in terms of rigorous semantics, manipulated by appropriate synthesis and optimization tools and verified. Examples of platforms have been given using the concepts that we have developed. We conclude by mentioning that the Metropolis design environment [2], a federation of integrated analysis, verification, and synthesis tools supported by a rigorous mathematical theory of meta-models and agents, has been designed to provide a general open-domain PBD framework.

Acknowledgments

We gratefully acknowledge the support of the Gigascale Silicon Research Center (GSRC), of the Center for Hybrid Embedded System Software (CHESS) supported by an NSF ITR grant, of the Columbus Project of the European Community and the Network of Excellence ARTIST. Alberto Sangiovanni-Vincentelli would like to thank Alberto Ferrari, Luciano Lavagno, Richard Newton, Jan Rabaey, and Grant Martin for their continuous support in this research.

We also thank the member of the DOP center of the University of California at Berkeley for their support and for the atmosphere they created for our work. The Berkeley Wireless Research Center and our industrial partners, (in particular: Cadence, Cypress Semiconductors, General Motors, Intel, Xilinx and ST Microelectronics) have contributed with designs and continuous feedback to make this approach more solid. Felice Balarin, Jerry Burch, Roberto Passerone, Yoshi Watanabe and the Cadence Berkeley Labs team have been invaluable in contributing to the theory of metamodels and the Metropolis framework.

References

[1] R. Alur, T. Dang, J. Esposito, Y. Hur, F. Ivancic, V. Kumar, I. Lee, P. Mishra, G. J. Pappas, and O. Sokolsky. Hierarchical Modeling and Analysis of Embedded Systems. *Proc. of the IEEE*, 91(1):11–28, January 2003.

[2] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli. Metropolis: an integrated electronic system design environment. *IEEE Computer*, 36:45–52, April 2003.

[3] Felice Balarin, Massimiliano Chiodo, Paolo Giusto, Harry Hsieh, Attila Jurecska, Luciano Lavagno, Claudio Passerone, Alberto Sangiovanni-Vincentelli, Ellen Sentovich, Kei Suzuki, and Bassam Tabbara. *Hardware-Software Co-Design of Embedded Systems: The POLIS Approach.* Kluwer Academic Publishers, Boston/Dordrecht/London, 1997.

[4] M. Baleani, A. Ferrari, L. Mangeruca, A. Sangiovanni-Vincentelli, M. Peri, and S. Pezzini. Fault-tolerant platforms for automotive safety-critical applications. In *Proc. of the Intl. Conf. on Compilers, Architectures and Synthesis for Embedded Systems*, pages 170–177. ACM Press, 2003.

[5] M. Barborak, M. Malek, and A. Dahbura. The consensus problem in fault-tolerant computing. *ACM Computing Surveys*, 25(2):171–220, June 1993.

[6] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The Synchronous Language Twelve Years Later. *Proc. of the IEEE*, 91(1):64–83, Jan. 2003.

[7] "F. De Bernardinis, S. Gambini, F. Vinci, F. Svelto, R. Castello, and A. Sangiovanni Vincentelli". "design space exploration for a umts front-end exploiting analog platforms". In *Proc. Intl. Conf. on Computer-Aided Design*, 2004.

[8] F. De Bernardinis, M.I. Jordan, and A.L. Sangiovanni Vincentelli. Support vector machines for analog circuit performance representation. In *Proc. of the Design Automation Conf.*, June 2003.

[9] Peter Bunus and Peter Fritzson. A debugging scheme for declarative equation based modeling languages. In *Practical Aspects of Decl. Languages : 4th Int. Symp.*, page 280, 2002.

[10] Henry Chang, Larry Cooke, Merrill Hunt, Grant Martin, Andrew McNelly, and Lee Todd. *Surviving the SOC Revolution: A Guide to Platform Based Design,* . Kluwer Academic Publishers, Boston/Dordrecht/London, 1999.

[11] C. Dima, A. Girault, C. Lavarenne, and Y. Sorel. Off-line real-time fault-tolerant scheduling. In *Euromicro 2001*, Mantova, Italy, Feb. 2001.

[12] S. Edwards, L. Lavagno, E. Lee, and A. Sangiovanni-Vincentelli. Design of embedded systems: Formal methods, validation and synthesis. *Proc. of the IEEE*, 85(3):266–290, March 1997.

[13] 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. of the IEEE*, 91(1):127–144, January 2003.

[14] Christian Ferdinand, Reinhold Heckmann, Marc Langenbach, Florian Martin, Michael Schmidt, Henrik Theiling, Stephan Thesing, and Reinhard Wilhelm. Reliable and precise WCET determination for a real-life processor. *Lecture Notes in Computer Science*, 2211:469–485, 2001.

[15] A. Ferrari and A. L. Sangiovanni-Vincentelli. System Design: Traditional Concepts and New Paradigms. In *Proc. Intl. Conf. on Computer Design*, pages 1–12, October 1999.

[16] Brasileiro FV, Ezhilchelvan PD, Shrivastava SK, Speirs NA, and Tao S. Implementing fail-silent nodes for distributed systems. *IEEE Transactions on Computers*, 45(11):1226–1238, November 1996.

[17] T. A. Henzinger, B. Horowitz, and C. M. Kirsch. Embedded control systems development with GIOTTO. In *Proc. of Languages, Compilers, and Tools for Embedded Systems*, pages 64–72. ACM Press, 2001.

[18] K. Keutzer, S. Malik, A. R. Newton, J. Rabaey, and A. Sangiovanni-Vincentelli. System level design: Orthogonalization of concerns and platform-based design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(12), December 2000.

[19] H. Kopetz and G. Grundsteidl. TTP - A Protocol for Fault-Tolerant Real-Time Systems. *IEEE Computer*, 27:14–23, January 1994.

[20] H. Kopetz and D. Millinger. The transparent implementation of fault tolerance in the time-triggered architecture. In *Dependable Computing for Critical Applications*, San Jose, CA, 1999.

[21] Lamport L. and Melliar-Smith P. Byzantine clock synchronization. In *3rd ACM Symposium on Principles of Distributed Computing*, pages 68–74, New York, 1984. ACM.

[22] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Trans. on Progr. Languages and Systems*, 4(3):382–401, July 1982.

[23] J.C. Laprie, editor. *Dependability : basic concepts and terminology in English, French, German, Italian and Japanese*, volume 5 of *Series title: Dependable computing and fault-tolerant systems*. Springer–Verlag, New York, 1992.

[24] E. A. Lee. What's ahead for embedded software? *Computer*, 33(9):18–26, 2000.

[25] E. A. Lee and A. Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(12):1217–1229, December 1998.

[26] I. Macbeth. Programmable Analog Systems: the Missing Link. In *EDA Vision (www.edavision.com)*, July 2001.

[27] C. Pinello, L. P. Carloni, and A. Sangiovanni-Vincentelli. Fault-tolerant deployment of embedded software for cost-sensitive real-time feedback-control applications. In *Proc. European Design and Test Conf.* ACM Press, 2004.

[28] J. Platt. Sequential minimal optimization: A fast algorithm for training support vector machines. Technical Report MSR-TR-98-14, Microsoft Research, 1998.

[29] A. L. Sangiovanni-Vincentelli. Defining Platform-Based Design. In *EEDesign. Available at www.eedesign.com/story/OEG20020204S0062)*, February 2002.

[30] Marco Sgroi. *Platform-based Design methodologies for Communication Networks*. PhD thesis, University of California, Berkeley, Electronics Research Laboratory, December 2002.

[31] H.S. Siu, Y.H. Chin, and W.P. Yang. Reaching strong consensus in the presence of mixed failure types. *Trans. Parallel and Distr. Systems*, 9(4), April 1998.

[32] A. J. Wellings, L. Beus-Dukic, and D. Powell. Real-time scheduling in a generic fault-tolerant architecture. In *Proc. of RTSS'98)*, Madrid, Spain, Dec 1998.