The Art and Science of Integrated Systems Design

Luca P. Carloni[†], Fernando De Bernardinis[†§],
Alberto L. Sangiovanni-Vincentelli[†], and Marco Sgroi[†]
[†]*University of California at Berkeley,*
Berkeley, CA 94720-1772
[§]*Dipartimento di Ingegneria dell'Informazione*
Università di Pisa, Italy
www-cad.eecs.berkeley.edu/~{lcarloni,fdb,alberto,sgroi}

Abstract

The design of next-generation integrated systems requires that "Art", a mix of knowledge, experience, intuition and creativeness, be supported by "Science", i.e., design methodologies that provide rigorous foundations and guarantee correctness either by construction or by a set of powerful synthesis and verification tools. We present platform-based design as the overarching principle to develop a class of methodologies that satisfy in part the requirements set above. A platform is an abstraction layer that hides the details of several possible implementation refinements of the underlying layers. We discuss the importance of carefully defining the platform layers and formally deriving the transitions from one platform to the next, including the role of top-down constraint propagation and bottom-up performance estimation. Finally, we present examples of these concepts at different key articulation points of the design process: system platforms and implementation platforms including analog design, thus covering the entire spectrum of design from conception and algorithms to final SoC implementation.

1. Introduction

Time-to-market pressure, design complexity and cost of ownership for masks are driving the electronics industry towards more disciplined design styles that favor design re-use and correct-the-first-time implementations. However, traditional computer-aided design (CAD) tool flows come short of providing proper support to reach these goals mainly because they are built as juxtaposition of independently-conceived stages. Each stage manipulates a representation of the final design by applying certain transformations on it before the next stage takes over. The exchange of information between stages is based on standard data formats, where functional specification, structural information and performance-related annotation are intertwined. These data formats are autonomously interpreted by the tools at each stage. The tools have different purposes (functional manipulation, structural transformation, performance optimization) and, based on their interpretation of the current status of the design, take decisions that have global effects and that are difficult to reverse. As a consequence, several time-consuming iterations between subsequent stages of the design flow are often necessary. For instance, due to the increasing impact of second-order physical effect in deep submicrometer (DSM) technologies (0.13μ and below), designers are forced to iterate many times between hardware-description language (HDL) specification and layout, because logic synthesis uses statistical delay models which badly estimate the impact of post-layout wire load capacitance and the netlist lack of structure prevents manual intervention on the layout. Furthermore, HDLs allow poor control on physical design and the output of logic synthesis is not robust with respect to small variations in the HDL specification. The general problem is that the design flow does not provide formal methods to propagate (and refine) design constraints from the first stages of the process, where the specs are set, down to the last stages where the final implementation is derived. Similarly, there is lack of formal methods to uniformly feed the first stages with those technology and performance parameters that are imposed by the adoption of a given process technology and the choice of a specific library of pre-design components. The ultimate consequence is that designers struggle in:

- orienting themselves among the various design representation,

- understanding the combined behavior of the tools, and

- tracking the evolution of *their* design.

Naturally, as they must spend time coping with these issues, designers encounter increasing difficulties in leveraging their knowledge, experience, intuition, and creativeness.

We argue that now more than ever the art of integrated system design needs to be supported by science, i.e., design methodologies that offer rigorous foundations to build a design flow that guarantees correctness of the design by construction or features powerful verification

and synthesis tools. In particular, the creation of an economically feasible electronic design flow requires a structured methodology that *theoretically limits the space of exploration, yet still achieves superior results in the fixed time constraints of the design*. This approach has been very powerful in design for both integrated circuits and computer programs. For computer programs, the use of high-level programming languages has replaced for the most part assembly languages; for integrated circuits, regular structures such as gate arrays and standard cells have replaced transistors as a basic building block.

In addition to the obvious pressure towards increasingly shorter time-to-market, there is a new phenomenon that is taking place in the electronics industry and that has to be addressed by design methodologies. 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. Product specification, intellectual property (IP) creation, design assembly and manufacturing are, for the most part, no longer taking place in the same organization. Over the last decade, we have been assisting to the *disaggregation of the electronic industry* from a vertically-oriented model into a horizontally-oriented one. In this situation, integration of the supply chain has become a serious problem. In particular, the hand-off points in the design chain from one company to another are at risk. If they are not defined clearly and with no ambiguity, then costly re-designs are the likely result. The most critical hand-off points in the design chain are between system, subsystem and IC companies on one hand, and between IC design and manufacturing. For this reason, we call these points *articulation points* to stress their importance in the overall design process.

Finally, the cost of mask ownership and "custom" designs together with the need of maintaining flexibility up to the last moment to be able to accommodate engineering and specification changes, has caused a decrease in number of design starts that is increasingly noticeable. ASIC solutions are less and less appealing for system companies and application-specific ICs are not always economically feasible for semiconductor companies. Hence, semiconductor companies that manufacture and design silicon will increasingly design standardized chips, capable of serving a range of applications via reconfigurable hardware and software, thus spreading the multimillion-dollar cost of a design across a range of applications. The boundaries between semiconductor companies and system companies have to be even more clearly defined than in the previous case. In fact, semiconductor companies need to minimize risks when designing these standardized chips. Hence they need to have a fairly complete characterization of the application spaces they wish to address with their products with the associated constraints in terms of affordable costs and performance levels. By the same token, system companies need to have an accurate characterization of the capabilities of the chips in terms of performance such as
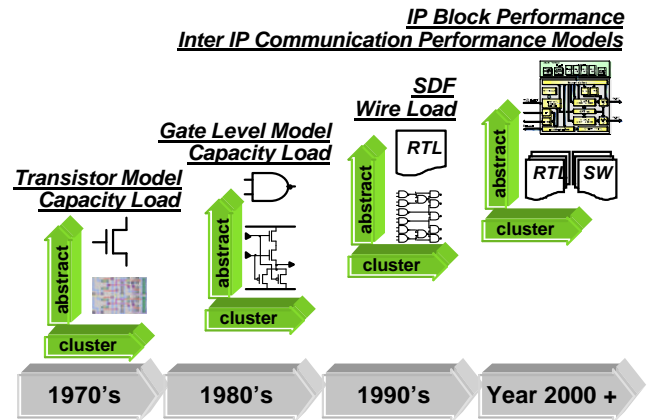

Figure 1. A brief history of abstraction in design (Source: F. Schirrmeister).

power consumption, size and timing, as well as *"Application Program Interfaces" (APIs)* that allow the mapping of their application into the chip at a fairly abstract level. APIs must then support a number of tools to ease the possibly automatic generation of the personalization of the programmable components of the chips.

Platform-Based Design is a methodology that has been indicated as an effective method to cope with the difficulties we exposed above [7]. To place this methodology in a historical perspective, it is useful to see it as the result of a natural progression in the quest for higher level of abstractions illustrated in Figure 1. Since the term "platform" has been used in several ways by the various segments of the electronic industry, we believe there is a substantial need of defining precisely this term and the accompanying methodology in an extended framework so that it can be used to support both traditional ASIC design flows and design flows that extensively use software and reconfigurable hardware to implement system functionality. For us, a *platform represents a layer in the design flow for which the underlying, subsequent design-flow steps are abstracted*. Thus a design from conception to implementation can be seen in our framework as a set of platforms (abstraction layers) and of methods to transform the design from one platform to the next. Platform-based design provides a rigorous foundation to design re-use, *correct-by-construction* assembly of pre-designed and pre-characterized components (versus full-custom design methods), design flexibility (through an extended use of reconfigurable and programmable modules) and efficient compilation from specifications to implementations. At the same time, it allows us to trade-off various components of manufacturing, NRE and design costs while sacrificing as little as possible potential design performance.

In this paper, we first summarize (Section 2) the motivations and principles for platform-based design. Our goal is to define what these characteristics are so that a common understanding can be built and a precise reference can be given to the electronic system and circuit de-
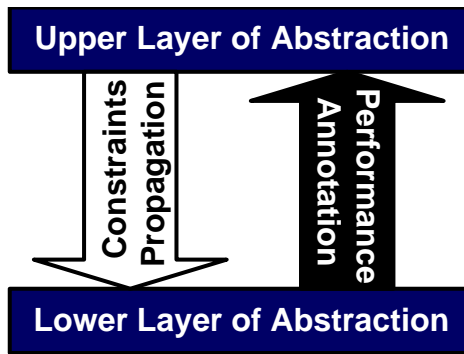
Figure 2. Interactions Between Abstraction Layers.

sign community. The rest of the paper is about the use of this principle in system design and in silicon implementation. We begin in Section 3, with the description of Platforms that are of great importance to define the hand-off of design from system conception to system implementation, the ones that have received most of the attention in the past few years. For pure system design, we selected network design as an important example of application (Section 4). For silicon implementation, we believe that analog components will have an increasing importance in determining the overall design time and effort. For this reason, we will show in Section 5 how to apply the basic principles of platforms to the analog domain to make the design of System-on-Chip (SoC) easier and based on a uniform methodology. This part is quite novel since most of the work on platforms has taken place at the architecture/micro-architecture boundaries.

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.

In general, 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 intercon-

nect 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 2, 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. From a historical perspective and the newly formed concepts stated above, we can state that the general definition of a platform is an abstraction layer in the design flow that facilitates a number of possible refinements into a subsequent abstraction layer in the design flow. 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. Every pair of platforms, along with the tools and methods that are used to map the upper layer of abstraction into the lower level, is a *platform stack*. 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 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*, that enables the abstraction of implementation detail to facilitate higher-level optimization, and *verifiability*, i.e. the ability to formally ensure correctness. On the other hand, when design-time and product volume permit it, it may be useful to skip intermediate platforms. This is equivalent to enlarge the design space and, therefore, can potentially produce a superior design. However, even if a "large-step-across-platform flow" can be adopted, there is still a benefit, from an evaluation standpoint, in having a lower-bound on the optimality of the feasible design as the one that can

be provided by a more constrained and predictable flow. Naturally, the larger the step across platforms, the more difficult is predicting performance, optimizing at system level, 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. Looking at the involved platforms, 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 System Definition-Implementation Articulation Point

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 [1, 3, 4, 5]). In studying this articulation point (see [7] for full details), we have discovered that at this level there are indeed two distinct platforms that form the "system platform stack" that need to be defined together with the methods and tools necessary to link the two: an "architecture" platform and an "API" platform. The API platform is used for 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 commonality 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 by personalizing an appropriate set of parameters that control 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 particular implementation may be selected from the pre-designed library of components depending on the particular 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 allows 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 have to 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. That means that 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. Run-time reconfigurability, where FPGA blocks can be customized by the user without the need of changing mask set, thus saving both design cost and fabrication cost. Design-time reconfigurability, where the silicon is still application-specific; in this case, 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 re-configurable 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 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 Programmers 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 or Programmers Model is a unique abstract representation of the architecture platform via the software layer. With an API so defined, the application software can be re-used for every platform instance. Indeed the Programmers Model (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
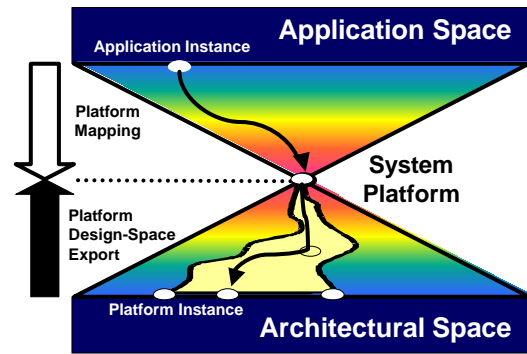

Figure 3. System Platform Stack.

more instances it contains. For example, to share source code, we need to have the same operating system but not necessarily the same instruction set, while to share binary code, we need to add the architectural constraints that force to use the same ISA, thus greatly restricting the range of architectural choices.

The RTOS is responsible for the scheduling of the available computing resources and of the communication between them and the memory subsystem. Note that in 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 3. The vertex of the two cones represents the combination of the API or Programmers' Model 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 Programmers' Model or API can be carried out, at least in part, automatically if a set of appropriate software tools (e.g., software synthesis, RTOS synthesis, device-driver synthesis) is available. It is clear that the synthesis tools have to be aware of the architecture features as well as of the API. 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 this latter case, platform-based design amount to adding to traditional design methodologies some higher level of abstractions.

4. Network Platforms

One of the most challenging problems in the design of distributed systems is the choice of the resources, such as physical links and protocols, that support the communication among the system components while satisfying a given set of constraints on cost and performances. To simplify the problem, designers usually decompose the problem into a stack of distinct protocol layers following the OSI layering principle. Each protocol layer together with the lower layers defines a platform providing services to its users, i.e., the upper layers and, at the topmost of the stack, the application-level components. Identifying from the communication requirements of the application the number and type of protocol layers requires finding a good compromise between optimality (minimize the number of layers) and design manageability (maximize the number of intermediate steps), that is the problem of platform-based design as defined in Section 2. Furthermore, exploring the design space to determine the functionality of each protocol layer and its implementation requires the use of tools and methodologies that allow to evaluate the performances and guarantee the satisfaction of constraints after each step. For these reasons, we believe that the Platform-Based Design principles and methodology outlined in the previous section can be very effective in protocol design. In this section, first, we formalize the concept of Network Platform. Second, we outline a methodology for selecting, composing and refining network platforms [8].

4.1. Definitions

A *Network Platform (NP)* is a set of resources that are composed together to form a Network Platform Instance (NPI) and provide *Communication Services (CS)* to a group of interacting system components (NPI users). Hence, the behaviors and the performances of an NPI are defined in terms of the type and the quality of the CS it is capable to provide. The structure of an NPI is defined by a set of nodes and links connecting them. Ports interface nodes with links or with the external environment of an NPI. We formalize the behaviors of an NPI using the event as a communication primitive that models either a send or a receive action of an NPI component. An event is associated with a message and identified by the type and the value of the message and by tags specifying attributes such as ordering or time of the corresponding action. A behavior of an NPI component, is defined by a totally ordered sequence $s = (e_1, e_2...e_n)$ of events e_i observed at its input and output ports. The set of behaviors of an

NPI can be obtained by intersecting the behaviors of the individual components.

A Network Platform Instance is 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 $(N_i, L_i, \pm)$, where $N_i \in N$ is a node, $L_i \in L \cup E$ is either a link or the environment E and $+(-)$ identifies an input (output) port.

- S is a set of behaviors, each identified by a totally ordered sequence of events observed at the ports $P_i \in P$.

The services provided by an NPI are called Communication Services (CS) and allow users to exchange messages between NPI input and output ports. Properties of a CS (commonly called quality of service or QoS) are, for example, the correctness and the delay of the message transfer. Formally, a Communication Service (CS) is a tuple $(\overline{P}^{in}, \overline{P}^{out}, M, E, h, g, <^t, t)$, 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, t is a mapping $t : E \rightarrow T$ associating each event with its timestamp. 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 belong to the same user or are associated with the same message. To facilitate the design task it is key to provide a user with an abstraction that hides the details of the internal components of the NPI and describes only the behavior which is observable at the ports interfacing the NPI with the environment. This abstraction is an Application Programming Interface (API) and consists of a set of methods that can be invoked by external users to access the services provided by the NPI. An API layer is defined by a set of CS and the methods available to access them. Using CS allows to express much more compactly the behavior of an NPI.

4.2. Quality of Service

NPIs can be classified according to the number, the type, the quality and the cost of the CS they offer. Number, type and quality of the supported CS define the capabilities of the NPI. To describe effectively the properties of a CS, it is convenient to export a set of QoS parameters such

as error rate, latency, throughput, jitter. To quantify such parameters, we use event annotations and compare value and timestamps of the events in CS. For example one can compare the values of pairs of input and output events associated with the same message to measure the error rate, or compare the timestamp of events observed at the same port to compute the jitter. We label events with indexes j and i, so that $e^{j,i} \in e^{(\overline{P}^{in} \cup \overline{P}^{out}),M}$ indicates the event carrying the i-th message and observed at the j-th port, and define the main QoS parameters as follows:

- **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 [1] $LostM = \{e_i \in E | h(e_i) \in \overline{P}^{in}, \neg\exists e_j \in E \text{ s.t. } h(e_j) \in \overline{P}^{out} g(e_j) = g(e_i)\}$, $CorrM = \{e_i \in E | h(e_i) \in \overline{P}^{in}, \exists e_j \in E \text{ s.t. } h(e_j) \in \overline{P}^{out}, g(e_j) = g(e_i), v(e_j) \neq v(e_i)\}$ and $InM = \{e_i \in E | h(e_i) \in \overline{P}^{in}\}$, the message error rate $MER = \frac{|LostM| + |CorrM|}{|InM|}$ [2].

Using the above definitions, the following types of CS can be defined:

- **Unicast vs. Broadcast CS:** a CS is unicast if every message is transferred to at most one output port, broadcast if messages are transferred to all output ports.

- **Lossless vs. Lossy CS:** a CS is lossless if $MER = 0$, otherwise is called lossy.

- **In-order vs. Out-of-order CS:** in-order communication services ensure that the order of messages at the input ports is maintained at the output ports, i.e. $\forall m_i, m_j \in M(m_i \neq m_j) e^{in,i} < e^{in,j} \Rightarrow e^{out,i} < e^{out,j}$

- **Synchronous vs. Asynchronous CS:** in synchronous CS the communication delay is negligible and the input and output events carrying

---

[1] $v(e_i)$ gives the value of the message carried by event e_i.

[2] MER can be converted to Packet and Bit Error Rate, if the encoding of the messages is known.

the same message have the same timestamp, i.e. $\forall m_i, m_j \in M(m_i \neq m_j), t(e_i^{in}) = t(e_i^{out})$

4.3. Examples of Network Platforms

A wire and two end nodes define an elementary type of NPI that provides in-order and lossy CS due to the noise in the wire. A bounded FIFO channel is an NPI providing asynchronous and in-order CS. It offers lossy (when overflows) or lossless CS depending on the arrival pattern of the input events. If the protocol at the end nodes includes a blocking write mechanism, it can provide lossless CS for a larger set of input patterns. More complex NPIs can be designed selecting protocols and physical media or reusing existing NPIs as building blocks. If the type and the quality of the services that a given NPI provides do not satisfy the communication requirements, additional protocol layers can be introduced to derive another NPI offering more sophisticated services. For example, if a physical link designed to transmit messages between one pair of users is to be shared by multiple pairs of communicating users, one must introduce a MAC protocol that schedules transmissions and avoids collisions. The components of the extended NPI are the physical link initially given and the MAC protocol. Figure 4 shows three examples of NPIs: a reliable one-hop NPI that supports lossless communication among components (in the figure labeled as A and B) directly connected by a single hop link, a multi-hop NPI for communication between end nodes (C and D in the figure) that are connected by a path composed of multiple nodes and links, and a reliable end-to-end NPI for lossless communication between end users (E and F).

4.4. Design of Network Platforms

The starting point of the design of an NPI is the set of behaviors of the interacting system components that define an abstract NPI and a set of constraints on the quality of the CS that the fully implemented NPI must provide. The procedure we propose is based on the concept of *successive refinement*: successive because it usually consists of a sequence of steps and refinement because at each step the communication is specified at a greater level of detail and at a finer granularity in time, space or data types. The refinement of an NPI consists of defining a more detailed NPI′ by replacing one or more components in the original NPI with a set of components or 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. For example, an NPI consisting of a direct link between end nodes can be refined into an NPI′ where intermediate nodes are used as repeaters (multi-hop network). The refinement is correct if the end-to-end delay of the multi-hop network is within the delay constraints satisfied by the direct link. 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 com-
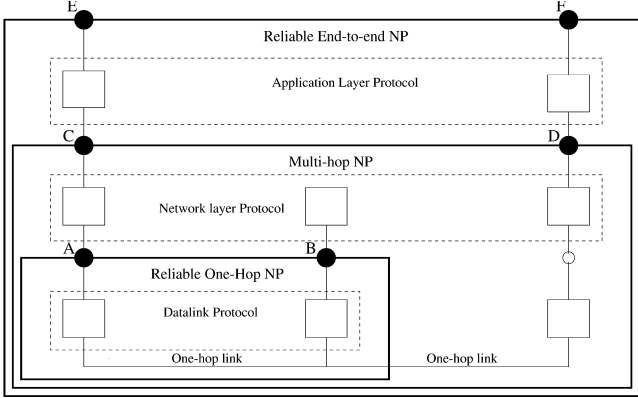
Figure 4. Examples of NPIs

ponents (or predefined NPIs) are selected from a library and composed to create CS of higher quality. Two types of compositions are possible. One type consists of choosing a 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 (and therefore of their CS) using an intermediate object called adapter (or gateway) that maps sequences of events between the ports being connected (horizontal composition).

5. Analog Platforms

Analog components are required in every system that interfaces with the physical world to acquire signals, and, hence, they are needed in any System-On-a-Chip (SOC) implementation. In addition, even when analog solutions could be replaced by digital ones from a pure functional point of view (for example, radio links), performance requirements may be so tight that the digital approach is out of the question.

Analog design has been traditionally the most difficult discipline of IC design. This difficulty stems from the effects that physical implementations have on the functionality of analog circuits. In the digital case, functionality depends on discrete sequences of discrete (binary) signals. Not so in the analog case, where continuous sequences (waveforms) of continuous values encode the information we need to manipulate and use. For this reason, any second order physical effect may have a significant impact on the function and performance of an analog circuit. Hence, the design of analog components has traditionally pivoted around low-level "clever tricks" (art or black magic?) that involve transistor layout and parameter selection, thus making virtually impossible to use higher levels of abstraction. At this juncture of the evolution of IC design where complexity and time-to-market reign, we believe there is a need to develop more abstract design techniques that can encapsulate some of the art of analog design into a methodology that could shorten design time without compromising the quality of the solutions. Actually, given that traditional analog design is carried out

at the transistor level, design space exploration is quite limited because of the time needed to evaluate alternative solutions with circuit simulators.

We believe that 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. 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. The reason for this is the difficulty in the analog world of knowing whether something is realizable without actually attempting to design the circuit. The number of effects to consider and their complex interrelations make this problem approachable only through the experience of past designs.

5.1. Definitions

We define an *Analog Platform* (AP) as a library of analog components and interconnect. Each components is a parameterized block with parameters varying over a continuous space. An AP is characterized by an input (parameter) space and an output (performance) space. The input space $\mathcal{I}$ defines the region of the architectural space that is captured by the components of the Analog Platform. For example, in a Low Noise Amplifier (LNA) we may define $\mathcal{I}$ as $\{W_{in}, L_{in}, I_{bias}\}$, where the first two quantities are related to the input transistor and I_{bias} is the bias current. Each quantity spans a given interval of values. The LNA component is itself a platform and represents only the implementations defined by $\mathcal{I}$, thus constraining the architecture space. A platform instance defines specific values for $\{W_{in}^*, L_{in}^*, I_{bias}^*\}$, i.e. the actual circuit to be considered. The output space $\mathcal{O}$ defines the performance parameters that are needed in order to evaluate (the mapping of a behavior on) the platform (performance annotation). Continuing with the LNA example, $\mathcal{O}$ may be defined as $\{Gain, Power, Noise\}$. For each platform being considered, an `evaluate()` method maps $\mathcal{I}$ into $\mathcal{O}$. Mathematically, we can consider such a method to be a function $\phi(\cdot) : \mathcal{I} \to \mathcal{O}$. The function $\phi(\cdot)$ can be defined in different ways, depending on the level of abstraction of the current platform, spanning from circuit simulations to circuit equations.

A key concept of Analog Platforms is that only $\mathcal{O}$ has to be exported at the current level of abstraction (*platform opaqueness*), because the function $\phi(\cdot)$ and $\mathcal{I}$ are not necessary to evaluate the performances of mapped behaviors. Because of this, Analog Platforms are also suitable candidates to protect sensitive data when exporting IPs. Platform performances can be represented through mathematical relations, $\mathcal{P}(Gain, Power, Noise) = 1$, which are sat-

isfied only by those *n*-tuples of performance figures that are actually obtainable with the current platform. In this way it is possible to capture all the interrelations among the different performance figures and annotate behavioral models consequently.

In order to allow hierarchical exploitation of platform instances, three operations have to be defined and implemented in terms of $\mathcal{P}$s:

- **abstraction** - given a platform relation $\mathcal{P}(Gain, Power, Noise, IP3, SR) = 1$ we may want to derive an abstracted view of the platform (*virtual* platform) that only relates a subset of performance figures, e.g. $\mathcal{P}'(Gain, Power, Noise) = 1 \Leftrightarrow \exists IP3^*, SR^*$ s.t. $\mathcal{P}(Gain, Power, Noise, IP3^*, SR^*) = 1$. This operation is needed every time we use models at higher levels of abstraction.

- **composition** - given two platforms A and B and some interface parameters λ (e.g. the output load for A/input load for B), we may want to derive $\mathcal{P}_{AB}(x, y) = 1 \Leftrightarrow \exists \lambda$ s.t. $\mathcal{P}_A(x, \lambda) = 1$ and $\mathcal{P}_B(y, \lambda) = 1$. $\mathcal{P}_{AB}$ represents the set of compatible performances of the composition $A \rightarrow B$.

- **merge** - given *n* platforms for the same functionality, a super-set platform can be defined by *or*-ing the respective $\mathcal{P}$'s after moving to a common level of abstraction. $\mathcal{P}_{merge} = 1 \Leftrightarrow \exists i$ s.t. $\mathcal{P}_i = 1$. Intuitively, this means considering the union of the performance space of individual platforms. For example, if each platform models a different topology for a given circuit functionality, a platform instance will define both the topology and the circuit to be refined, thus intrinsically performing architecture selection.

The generation of platform performances $\mathcal{P}$s is in general accomplished through sampling $\phi(\cdot)$ over the input space $\mathcal{I}$. This is because in the general case $\phi(\cdot)$ is a complex non-linear function, and no explicit representation for it might be available (in order to provide an accurate characterizations, $\phi(\cdot)$ may be implicitly defined by circuit simulation). The goal is to get a smooth continuous representation of the performance relation $\mathcal{P}$. This constitutes the bottom-up phase for building a platforms library and is the crucial step for achieving accurate performance annotations. The characterization of a platform over its input space is exponentially complex with the dimensionality of $\mathcal{I}$. Also, the characterization process generates large amounts of data that need to be effectively represented. The first problem requires some hints from the analog designer to limit the dimensionality of $\mathcal{I}$, i.e. which "knobs" are most meaningful for defining the platform and which constraints on the value of each parameter can be exploited to prune the characterization space. Design of experiments techniques may also be used to achieve optimal accuracy/complexity tradeoffs. Even so, large numbers of multi-dimensional samples may be
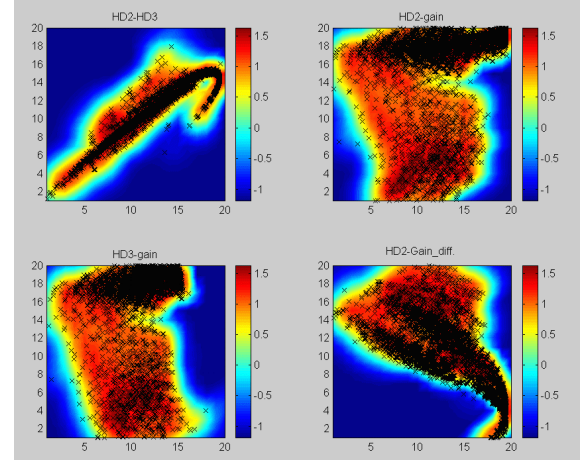

Figure 5. Projections of a 7-dimensional $\mathcal{P}$ for an LNA.

generated, on the order of thousands to hundreds thousands. As Figure 5 illustrates, an effective way of representing the information provided by these simulations is to use machine learning techniques, in particular Support Vector Machines (SVMs). SVMs make possible to store only a small portion of the original results (typically 10% or lower) while providing multidimensional interpolation that enables the extrapolation of performance figures out of the available data based on some continuity assumptions.

5.2. Analog Platform Stack

An Analog Platform Stack contains several layers of abstraction with the appropriate representation at each level. In the analog world, finding a model for a platform is a complex matter. Behavioral models allow fast simulations thus enabling more extensive explorations/optimizations. Furthermore, they are tailored for a specific platform to include at the behavioral level specific idiosyncrasies of the platform. By constraining models to reflect platform performances, i.e. constraining model parameters to satisfy the platform $\mathcal{P}$, a mapping of functionality onto architecture is achieved.

Analog Platform Stacks provide hierarchies of behavioral models at different levels of abstraction to reflect the refinement process typical of top-down flows. In order to promote design space exploration, all the models are derived from root models characterized by functionality families (see Figure 6). For example, we can have LNA, mixer and PLL families, which in turn generate trees of models at different levels of abstraction. The nodes of these trees represent higher-level platforms and the leaves implementation platforms. By exploiting the abstraction and merge operations defined above, all platforms in a given family can be included in a tree originating from the same root model (the most abstract platform). The refinement process proceeds selecting branches in the tree and, therefore, more detailed platforms, transforming con-
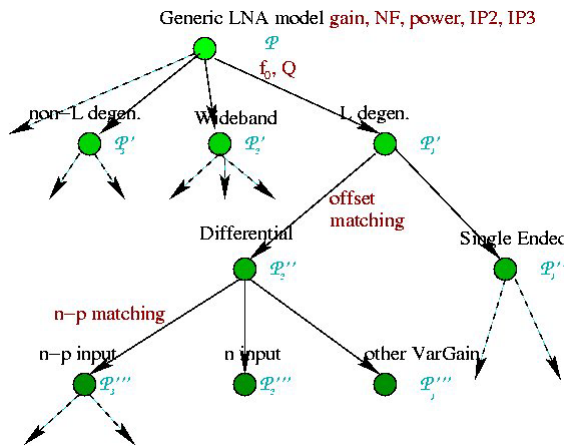
Figure 6. Sample model hierarchy for LNAs.

straints from a more abstract platform to a more detailed platform (exactly as described in the digital case). The mapping process ends with the selection of a platform instance of a particular platform.

Platforms require to explicitly model communication. Communication is a main actor in Analog Platforms, with even more emphasis than in the digital case. In fact, interactions between communication and computation in analog blocks are much more involved than for digital blocks. We can consider the communication between two blocks as well as the respective behaviors to be the fixed-point solution of the composition of the blocks. Because of this, orthogonalization becomes harder for detailed platforms and communication has to be modeled together with behavior. Therefore, the semantics of the component families requires to specify the allowed block interconnections. In this sense, the communication at the output of an LNA is modeled with respect to the mixer that is supposed to follow the LNA. At the most detailed level, this means that a given LNA can work with some mixers and not with others. Of course, different domains of application may require different composability characteristics among the component families.

Analog Platforms can be hierarchically composed to generate new Analog platforms at higher level of abstraction. PLLs provide examples of hierarchical composition since their complexity allows them to be considered as systems *per se*. The result of the composition is a new Analog Platform that needs to be endowed with a set of behavioral models. The process of deriving new platforms is the same as deriving platform for circuits (*fractal nature of design*), where $\phi(\cdot)$ now refers to the composition of APs in place of simple circuits. Therefore hierarchical composition provides a more general way to generate more abstract platforms other than the abstraction/merge operations previously defined.

5.3. Design of Analog Platforms

Up to now, analog system level design has been carried out only through experience and trial and error. APs enable systematic, high-level design exploration. The refinement and mapping processes available with APs allow top-down flows to evaluate system level trade-offs with respect to the performance space of the analog platform ($\mathcal{P}$). The process that maps constraints from one platform to the next proceeds until a platform implementation instance is selected, i.e. the specifications for all blocks have been defined. From here on, it is up to the IP provider to generate an implementation with the required performances or up to designers to size the schematic and generate the layout. The novelty of the approach consists in the fact that the use of $\mathcal{P}$s guarantees by construction that the set of specifications generated is feasible, thus drastically reducing the number of iterations required to get feasible specifications and achieving optimal points over larger parameter spaces.

If we focus on a wireless transceiver as an example, at the most abstract level (largest design space) its functionality is transferring information between two points using a shared, nonideal channel. The channel models both the physical channel and the effects deriving from hardware limitations (analog and digital) in terms of noise, linearity, gain, numerical accuracy and so on. An optimal transceiver implementation maximizes some function of the Bit Error Rate over a set of constraints and specification scenarios. A global optimum in the transceiver implementation implies optimum decomposition between analog and digital functionalities, optimum transceiver architecture, optimum block topology and finally optimum circuit design. It is then evident that optimality requires much more than optimal circuit synthesis (that deals only with local optima). Furthermore, optimizations at the circuit level may be outperformed by different circuits/topologies implementing the same block functionality. To really optimize a design, exploration is a must. By exploiting Analog Platforms, system trade-offs can be evaluated over much larger design spaces and optimal partitions of constraints can be derived for each individual platform. The bottom-up nature of the platform characterization guarantees that the required performances are achievable, and APs can actually provide means of automatically generating the requested platform instance.

Analog Platforms also separate the design of individual blocks from system design, thus allowing more effective explorations and optimizations at higher levels of abstraction where the design space is much larger. The introduction of the platform concept enables the integration of heterogeneous IPs, and does not put any constrain on analog designers for developing new circuit solutions for sensitive blocks. In this sense, APs may be exploited as a fast evaluation method for new solutions, generating coarse performance regions at first to evaluate the basic performances of a new solution at the system level, and, then, refining both the architecture and the functionality
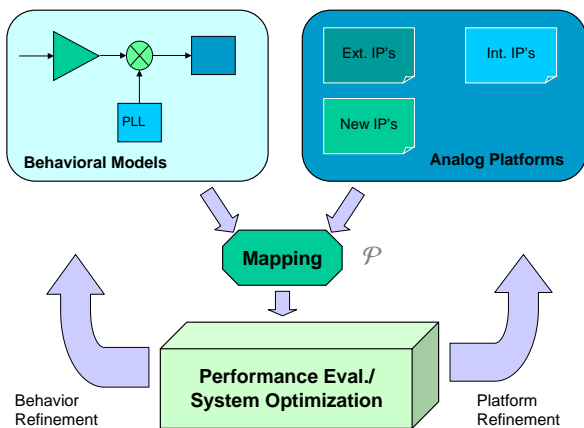
Figure 7. Analog design flow exploiting Analog Platforms.

to take full advantage of the new solution (if convenient). This process can be pictorially represented by the classic Y chart that is essentially the same as in the digital case (Figure 7).

Hierarchical platforms allow to progressively refine the system while maintaining accurate performance estimations in the process. This is a major obstacle in top-down methodologies that do not leverage the "meeting-in-the-middle" approach of platform based design. We can actually consider the platform performance relation $\mathcal{P}$ as a more sophisticated version of the flexibility function that was at the heart of the refinement process of the methodology proposed in [2]. Accurate platform evaluation methods are the key to the use of behavioral models for evaluating trade-offs at the analog/digital boundary, where fast and reliable methods are needed to determine which functionalities should be mapped on analog platforms and which on digital platforms. Taken one step further, even protocol and network topologies could be evaluated in a unified platform environment, leading to a *hardware/software/analog* co-design paradigm for embedded systems.

In the digital implementation platform domain, FPGAs have played an important role in providing flexibility with fast design time. In the analog domain, while several attempts have been made in the past, this level of reconfigurability has not been reached due to the difficulties in obtaining satisfactory analog designs by programming interconnects and functionalities of homogeneous building blocks. Field Programmable Analog Arrays [6] are a new attempt at this reconfigurable Analog Platform. FPAAs provide a software configurable switched capacitor array that can be programmed on the fly by microcontrollers in the platform. In this way, they provide an analog platform that is analogous to FPGAs in the digital world. 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, while the array already provides a basic platform level, 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, 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.

6. Conclusions

We have 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. A platform is an abstraction layer that hides the details of the several possible implementation refinements of the underlying layer. Platform-based design allows us 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 network platforms, implementation platforms, and analog platforms.

The platform-based methodology is supported by the Metropolis framework, a federation of integrated analysis, verification and synthesis tools supported by a rigorous mathematical theory of metamodels and agents. This framework will be available soon through the standard channels for software distribution handled by the Department of EECS, University of California at Berkeley.

7. Acknowledgments

[1] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, E. Sentovich, K. Suzuki, and B. Tabbara. *Hardware-Software Co-Design of Embedded Systems: The POLIS Approach,* . Kluwer Academic Publishers, Boston/Dordrecht/London, 1997.

[2] H. Chang, E. Charbon, U. Choudhury, A. Demir, E. Felt, E. Liu, E. Malavasi, A. Sangiovanni-Vincentelli, and I. Vassiliou. *A Top-Down, Constraint-Driven Design Methodology for Analog Integrated Circuits.* Kluwer Academic Publishers, Boston/Dordrecht/London, 1996.

[3] H. Chang, L. Cooke, M. Hunt, G. Martin, A. McNelly, and L. Todd. *Surviving the SOC Revolution: A Guide to Platform Based Design,* . Kluwer Academic Publishers, Boston/Dordrecht/London, 1999.

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

[5] 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.

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

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

[8] A. L. Sangiovanni-Vincentelli and M. Sgroi. Service based Model and Methodology for Network Platforms. Technical Report available at www-cad.eecs.berkeley.edu/s̄groi/tech_reports, June 2002.