

Fast Prototyping of Real Time Systems: A New Challenge?

Jan M. Rabaey

As electronic systems become more complex and gather more and more functionality, the attention of the design automation community is gradually drifting away from the chip prototyping problem and starts to focus on the development of system design technology. One might wonder how system prototyping differs from the traditional chip design problem. In this paper, we will demonstrate that there are indeed some fundamental differences, which will effect the design methodology at both the specification, simulation and implementation levels. On the other hand, we will also argue that a large part of the chip design methodology is valuable at the system level as well.

1. INTRODUCTION

When arguing about system design, a single question always comes to mind: What is really understood under the term *system* and where does it differ from, for instance, an integrated circuit. Answering this question brings us a long way towards understanding the difference between system and traditional integrated circuit prototyping. We have adopted the following definition of a system, which has, as will become obvious in the rest of the paper, a profound impact on the design methodology and tools: *a system is a self-contained entity, which is composed of a variety of sub-components with heterogeneous properties, communicating with each other using a variety of protocols.*

This definition has some important repercussions. First of all, no mention is made of a hardware platform. A system can span just a single integrated circuit, a printed circuit board, a card cage or a number of boxes. Secondly, the emphasis in system design is on heterogeneity. It brings together a number of widely varying components and architectures, some of them implemented in hardware, others realized in software, which use different means of communication and synchronization (for instance, synchronous or asynchronous, message passing based versus master-slave, etc.). This is in contrast to the traditional chip prototyping, which is based on a single architectural paradigm and uses a single communication mechanism, often being a synchronous, statically scheduled protocol.

Figure 1 shows an example of a complex system [1], which will be used as the driver example in the rest of the paper. It demonstrates the basic properties of a system in an ample way. The overall compute system consists of a number of components with varying architectures (compute servers, base stations or terminals), communicating with each other using a variety of protocols (wireless or wired). Each component on itself is once again a system. For instance, the infopad terminal consists of a number of elements, such as a radio modem or a video decompression unit. In this particular case, the units converse with each other using a buffered non-blocking message passing protocol.

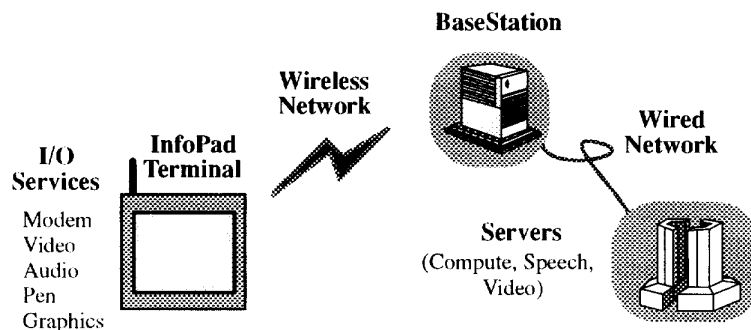


Figure 1 Infopad: a communication and computing system with wireless access.

This heterogeneity will be reflected throughout the design flow, presented in the following sections. We will consequently discuss issues regarding specification, simulation/emulation, design partitioning and implementation.

2. DESIGN SPECIFICATION/DESCRIPTION

As mentioned, a typical system merges a number of entities, whose underlying semantics are very diverging. Furthermore, these modules tend to communicate with each other using a variety of mechanisms. Believing that all these domains with their particular properties can be adequately modeled in a single environment is wishful thinking. From the VHDL experience, we have learned that meaningful design analysis and synthesis is only possible when the semantics of a domain and its interface with the surrounding environment are well understood.

Therefore, it is our belief that a heterogeneous specification environment is essential for the description of complex systems. In such an environment, each entity is

modeled using an appropriate *domain*, which matches its underlying concepts. For instance, most signal processing algorithms are adequately modeled using (static or dynamic) data flow. A finite state machine model is a good match for a global controller and an event driven model fits a communication channel with its random nature.

The hierarchical nature of a system, furthermore, requires that these domains can be nested. For instance, our experience has shown that a *communicating processes* domain [2, 3] is often the best suited for the description of a system at the uppermost level. The individual components of the system are modeled as a set of concurrent processes, communicating with each other by message passing. Each of the components is described in the appropriate semantic model (data-flow, sequential code, etc.) or, on its own turn, can be a complete system.

The idea of hierarchical, heterogeneous system modeling forms the core concept of the Ptolemy system [4], developed at U.C. Berkeley. It is implemented as an object-oriented framework, within which diverse models of computations can co-exist and interact.

As an example, Figure 2 shows a high level description of the Infopad system in the Ptolemy environment. It models the base station, the wireless channel and the wireless terminal as a set of processes. Depending upon the abstraction level, each pro-

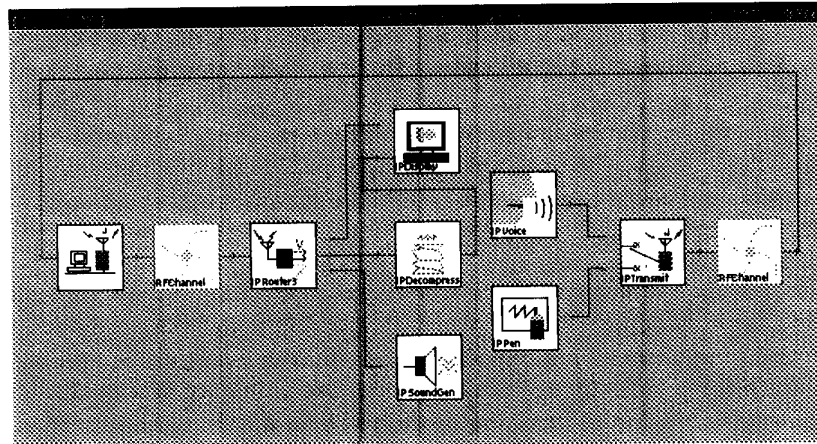


Figure 2 Communicating processes model of Infopad.

cess on its own is modeled in many different ways, either as an abstract model representing the I/O behavior only or, in more detail, by a complete sub-domain.

3. SIMULATION/EMULATION

Heterogeneous modeling results in leaner and more efficient simulation: as each sub-system is defined in its own domain with well understood semantics, a dedicated simulation engine can be used, which exploits the properties of that particular domain and avoids the overhead of the generic serve-all simulation approach. As an example, efficient architecture simulation of a micro-processor is possible when adopting a single evaluation per clock cycle regime. A simulator based on this concept is easily about a 100 times faster than a VHDL simulator, which can do exactly the same, but has to carry the extra overhead of being a full fledged event driven simulator.

The clear definition of the communication semantics, both within and between domains has another important advantage. The simulation and analysis of complex system, such as the multi-media computation and communication system of Figure 1, requires a tremendous amount of computation cycles, for instance to determine network bottlenecks or to analyze the effects of channel errors. Often, we would like to accelerate the simulation by distributing it over multiple workstations or by swapping in the actual hardware, when available. As an example, as the video server is implemented on an existing RAID server, video streams can be delivered directly from that server over the network instead of using a file based approach.

Such a hardware-software co-simulation is more easily achieved when the communication mechanism is well understood. In the communicating processes domain, for instance, a system-node can be migrated from the simulator to actual hardware, by replacing the simulation model of the node by a wrapper process, which behaves exactly like the original node from an input-output perspective, but has no other internal function but to transmit and receive the messages to/from the accelerator hardware [3].

This concept is demonstrated in the following example, which analyzes the basestation-terminal link of the infopad system. The basestation, router and audio sub-systems are implemented in software, while the video part of the terminal is implemented on an accelerator board, consisting mainly of a TMS320C30 (Figure 4). The link between the host workstation and the accelerator board is rather complex and proceeds over the ethernet to the VME-bus via a bridge, implemented on a single board micro-processor (Figure 3). It is worth mentioning that the migration of the code from the software domain to the hardware accelerator is totally automatic and all necessary software processes (at the host workstation, bridge and accelerator site) are synthesized from the defined communication protocol and its properties. Figure 4 shows a model of the simulation. The messages gen-

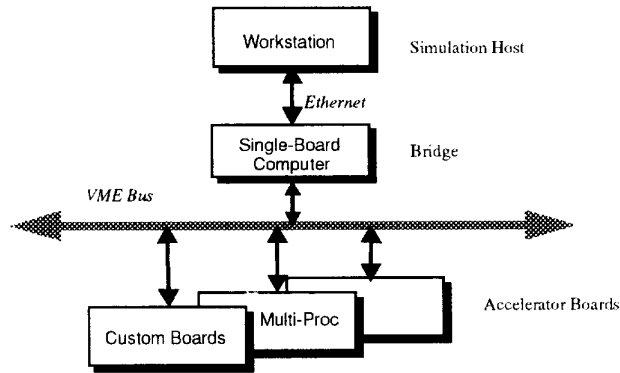


Figure 3 Hardware-Software Co-Simulation Environment.

erated by the remote process to monitor the simulation are shown in a separate window.

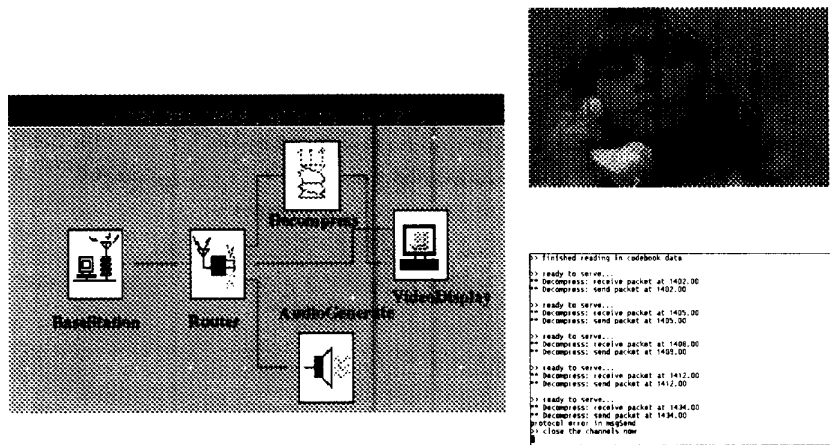


Figure 4 Hardware-Software Co-Simulation of Infopad.

The hardware-software co-simulation approach can result in an *incremental prototyping* design methodology. The major hurdle to the realization of this concept is the long latency of contemporary computer networks. Efforts towards the realization of low-latency networks are under way, however [5].

4. DESIGN PARTITIONING

One of the key challenges in the system design process is the partitioning of the design into sub-components and the choice of the appropriate architectural and hardware platform for each of them. This task is often quoted to be *intuitive*, but actual design experiences have shown the opposite. For instance, in the infopad design, moving functionality from the terminal to the base-station or the servers can reduce the power consumption of the hand-held terminal, but might result in bandwidth bottlenecks or reduced quality.

The major problem with system level partitioning is a lack of information on how a high level decision impacts the design cost and system performance. To decide if a certain module should be implemented in software or hardware, one needs an approximative idea with regards to the performance, cost or power consumption for both alternatives. As the actual architecture might not been known at that time or as a complete design synthesis of every alternative might take too much time, there exists a dire need for effective techniques to generate an reliable performance model of a module (area, time, power), given an architectural template.

One would hope that a single estimation or performance analysis tool might be sufficient to cover the complete architectural and algorithmic space. Unfortunately, that is not the case. As the estimator is a simplified version of the actual synthesis/ compilation process, its mechanisms will be influenced by the nature of the architecture, the properties of the application and the cost factor to be optimized. In all, four classes of performance analysis tools can be identified [6]:

- *Deterministic* estimators, based on a simplified synthesis process.
- *Stochastic* Estimators, which obtain a statistical mode from the analysis of a large set of design examples.
- *Benchmarking*. This approach is especially effective for black box synthesis/ compilation tools.
- *Profiling* (simulation based) to anticipate the impact of data dependencies.

What estimation approach to use depends upon the specification paradigm, the target architecture and nature of the mapping process. Published results for each of the estimator classes indicate that average prediction accuracies within 5-10% are definitely within reach. The availability of efficiently generated performance models is the first step towards either an interactive or automatic system partitioning and design style selection environment.

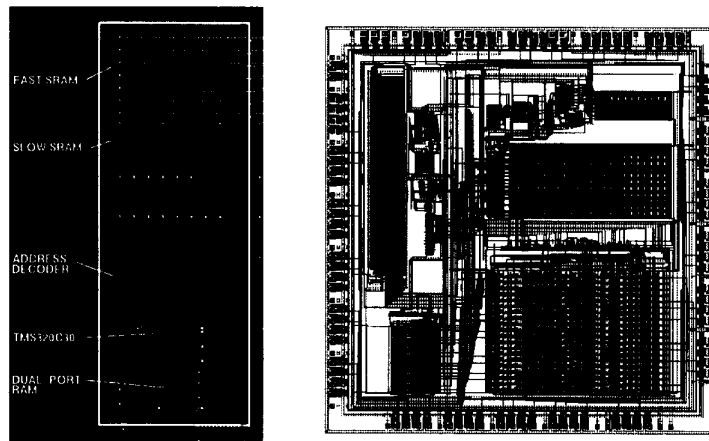
5. IMPLEMENTATION

From an implementation perspective, we can divide the components into two classes: *Data Processing Modules*, which implement the various application specific algorithms and the *Interconnect Modules*, which implement the communication protocols linking the Data Processing Modules [2].

As obvious from the preceding discussions, no single architectural template will cover all the potential implementations of the Data Processing Modules. A mix of structural and behavioral module generators, hence, have to be provided. At this level, we can borrow extensively from the design approaches and methodologies used in custom design: for instance, database and design management tools, used for ASIC's are readily adapted to the system level. The concepts of design re-use and modular design, which made complex ASIC design possible in the first place, are extendable as well.

For example, a library of parameterized hardware modules can be provided for commonly used entities, such as microprocessor or signal processor sub-systems, memory modules, data acquisition modules and bus interfaces. Parameters can be the number of processors in a multi-processor module, the size of the memory or the number of ports in an interface macro-element. Besides generating the netlist, the module generator might also define the placement, invoke board or MCM routers and even might synthesize some of the glue logic. In the SIERA system [7], a structure description language (called *sdl*) with powerful parametrization facilities is used to describe the module topology. An example of such a module is shown in Figure 5a, which displays an instance of a uni-processor module (build around the TMS320C30). The SIERA module library contains elements such as a VME slave module, a static RAM block and a multi-processor sub-system.

Other module generators start from a behavioral description. Existing ASIC synthesis tools, such as C2Silicon [8] and HYPER [9] are readily available as system level building block generators. An example of a module, implementing a robot controller and generated using the C2Silicon and LAGER [10] systems, is shown in Figure 5b. These blocks are easily incorporated into the global system, once a black box model, specifying the input/output interface, is available. One can even go one step further: combining a software compiler with a pre-existing processor board or a parameterized processor module, results in a "*software*" module generator. This demonstrates that there is no conceptual difference between hardware and software from a system perspective. Concepts such as modularity and reusability are valid in both cases. The key to the integration between hardware and software is a clear understanding of the interface semantics.



(a) A uni-processor structural module. (b) C2Silicon Generated Processor

Figure 5 System level module generators.

The latter will have a distinct impact on the nature of the Interconnect Modules. Depending upon the protocol chosen and the hardware platform, an interconnect module can vary from pure wires, a composition of glue logic, a complex hardware module containing buffer elements and complex control or a simple piece of software, implementing a semaphore and running on a processor module. A interconnect generator can, hence, vary from the very simple to the complex. For instance, on a typical integrated circuit, the communication protocol is implicit and based on a clock-driven, statically scheduled event ordering. A message passing protocol can easily be overlaid on top of the VME slave module, mentioned earlier as well as implemented as an RPC call, as was discussed earlier. Event driven protocols, such as often occur in board level design, require the generation of an asynchronous control module [11]. These protocols, and the associated generation, can be pretty complex, as is illustrated in Figure 6, which shows the protocol governing the write from a processor to a memory module.

6. SUMMARY

The central concept in system prototyping is heterogeneity, which is reflected at all levels of the design process. A clear understanding of the semantics of each subsystem and its relations with the environment results in modular, hierarchical design process, not that distinct from the well-established chip prototyping methodology.

