Specification and Design of Embedded Software/Hardware Systems

Daniel D. Gajski* and Frank Vahid**

Technical Report CS-94-08
October 24, 1994

* Department of Information and Computer Science
University of California
Irvine, CA 92717
gajski@uci.edu

** Department of Computer Science
University of California
Riverside, CA 92521
vahid@cs.ucr.edu

Abstract

System specification and design consists of describing a system's desired functionality, and of mapping that functionality for implementation on a set of system components, such as processors, ASIC's, memories, and buses. In this article, we describe the key problems of system specification and design, including specification capture, design exploration, hierarchical modeling, software and hardware synthesis, and cosimulation. We highlight existing tools and methods for solving those problems, and we discuss issues that remain to be solved.

Contents

List of Figures

.

1 Introduction

Embedded systems have become commonplace in recent years. Examples include automobile cruise-control, fuel-injection systems, aircraft autopilots, telecommunication products, interactive television processors, network switches, video focusing units, robot controllers, and numerous medical devices. While there is no widespread agreement of what defines an embedded system, we note that such systems possess a few key characteristics. The design of an embedded system is very heavily influenced by the system's interactions with its environment. Embedded systems also often have numerous modes of operation, must respond rapidly to exceptions, and often possess a great deal of concurrency. Unfortunately, there are few tools and methodologies to assist a designer tackle the difficult problem of designing a complex embedded system.

To illustrate the embedded-system design task, consider the design of an interactive TV processor system used to support interactive multimedia. The system stores video frames, and displays them as still pictures with accompanying text and audio. The system resides in a set-top box similar to a cable TV box, and a user interacts by selecting menu items with a keypad. A diagram of the overall system is shown in Figure 1. Design of the *Digital subsystem* involves first creating a specification of the subsystem's functionality, called a functional specification, and then mapping it to a system-level architecture, as shown in Figure 2. The subsystem is implemented with six components: three memories, two ASICs and a processor. The *Memory1* component stores two arrays used to hold audio bytes, while *Memory2* stores a video array. *Memory3* stores a fonts array and an array of characters to be displayed on the screen. The *ASIC1* component implements the functions that store incoming audio, and that generate the audio on demand. *ASIC2* implements the functions that store and generate video frames, and also that store special command bytes that may be encoded in the audio-video (AV) input. Finally, the *Processor* component implements the functions that process the special AV commands, the main computer commands, and user commands, and that overlays characters on the screen.

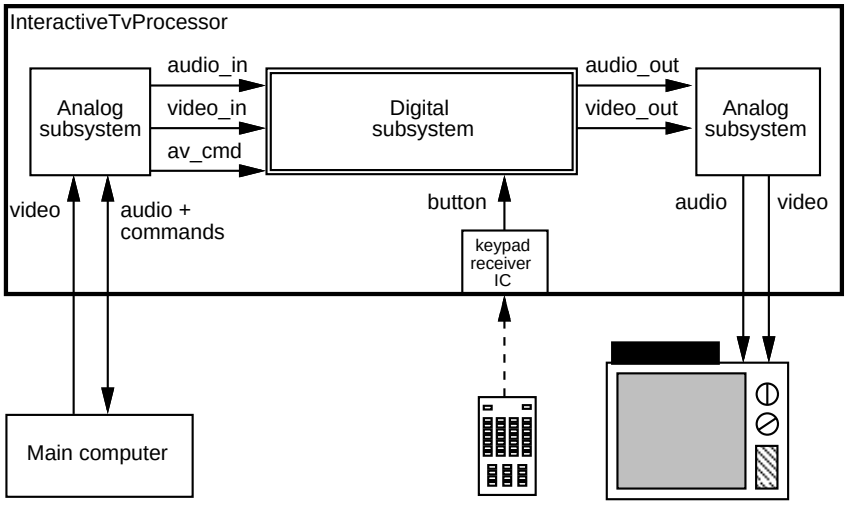


Figure 1: ITVP environment

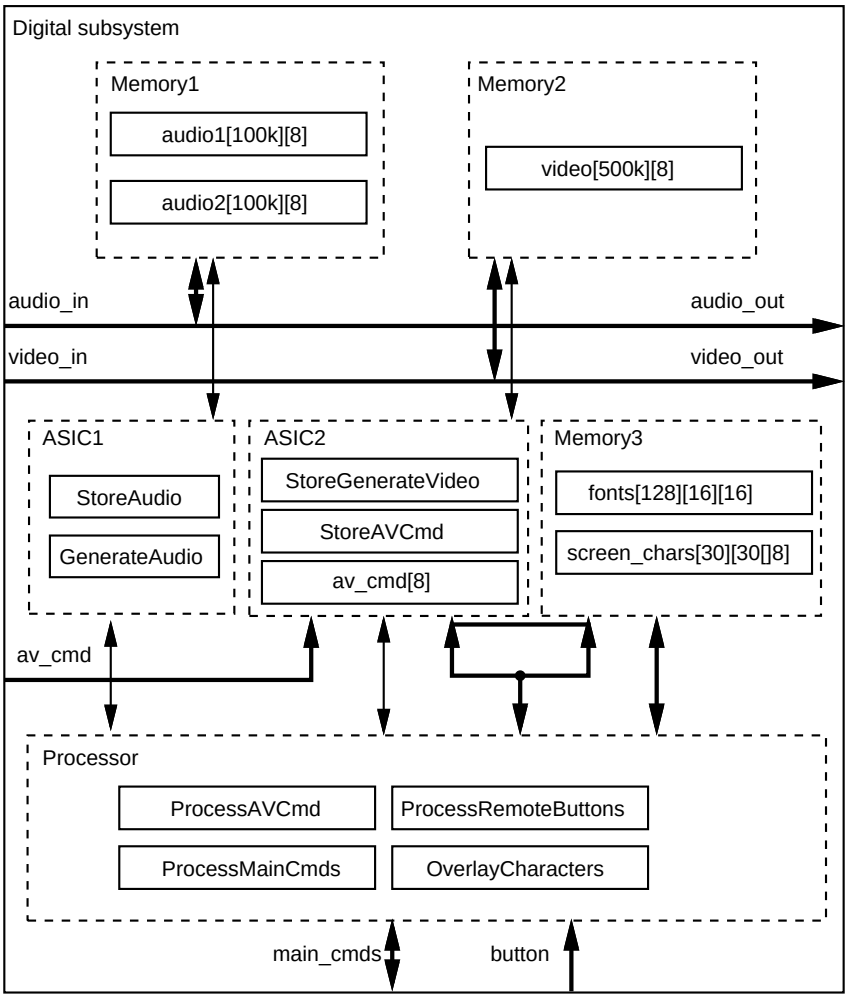


Figure 2: ITVP system-level design option

There are several tasks that must be performed to create a system-level design, as illustrated in Figure 3. First, we perform **specification capture**, whereby we specify the desired system functionality. To do this, we first decompose the functionality into pieces by creating a conceptual *model* of the system. We generate a *description* of this model in a language. We *validate* this description using simulation or verification techniques. The result of specification capture is a *functional specification*, which is void of any implementation detail.

Second, we perform **exploration**, in which we explore numerous design alternatives in order to find one that best satisfies our constraints. To do this, we *transform* the initial description into one more suitable for implementation. We *allocate* a set of system components and specify their physical and performance constraints, as in the example where we allocated three memories, two ASICs, one processor, and several buses. We *partition* the functional specification among allocated components. To guide these three sub-problems, we *estimate* the quality of each alternative design.

Third, we perform **specification refinement**, whereby we **refine** the initial specification into a new description that reflects the decisions that we made. To do this, we move each variable into a *memory*, insert *interface* protocols between components, and add *arbiters* to linearize concurrent accesses to a single resource. We *generate* a system description with the above details, consisting of a description of the system's processors, memories, and buses. We verify that this refined description is equivalent to the initial specification using *cosimulation*. The result of specification refinement is thus a *system-level description*, that possesses some implementation detail related to the system-level architecture that we have developed, but otherwise is still largely functional.

Fourth, we perform **software and hardware design**, whereby we create an implementation for each component, using software and hardware design techniques. A standard processor component requires *software synthesis*, which determines software execution order to satisfy resource and performance constraints. A custom processor component's architecture and design can be obtained through *high-level (behavioral) synthesis* [1], which converts the behavioral description into a structure of components from a register-transfer (RT) library containing microarchitectural components, such as ALUs, registers, counters, register files and memories. The control logic and some RT components are synthesized with finite-state machine and *logic synthesis* techniques [2]. The result of software and hardware design is an *RT-level description*, which contains optimized C code for software and RT-level netlists for custom components.

Finally, during **physical design**, we generate manufacturing data for each component. For software, this is as simple as *compiling* software into an instruction set sequence, whereas for hardware, we convert an RTL netlist into layout data for gate arrays, FPGA's, or custom ASIC's using physical design tools for *placement, routing* and *timing*.

The above five tasks roughly define embedded system-design methodology from product conceptualization to manufacturing. After each task, we generated a more refined description of the system, reflecting the decisions made by each task. Such a **hierarchical modeling methodology** preserves consistency through all levels, and offers high productivity gain by avoiding unnecessary iteration when inconcistencies are discovered late in the design cycle. Each model is used to verify different properties of the system. The functional specification serves to verify the completeness and correctness of the system functions. The system-level description is used to verify system performance and communication protocols. The RT-level description is used to verify the developed software code and operation of the custom design during each clock cycle. The physical description is used to verify detailed timing and electrical characteristics of the system. This hierarchical modeling technique distinguishes modern system-level methodologies from past ones, when only the physical model was captured, late in the design cycle, making specification or architecture changes impossible.

Today's system designer has little assistance available to help perform system-design tasks. There is no widely accepted methodology or tool to help with specification capture, exploration, specification refinement and hardware/software design. As such, most system design is done in an ad-hoc manner, relying heavily on informal and manual techniques, and exploring only a handful of possibilities. In this article, we describe the first three system-level tasks in more detail, describing possible solutions that have recently evolved. We only briefly sketch the tasks of hardware, software and physical design, since several excellent references exist on those subjects. Furthermore, we discuss the tasks of simulation and cosimulation. We conclude with the status and future of system design tools and methodology and their acceptability for production use. Further details are found in [3].

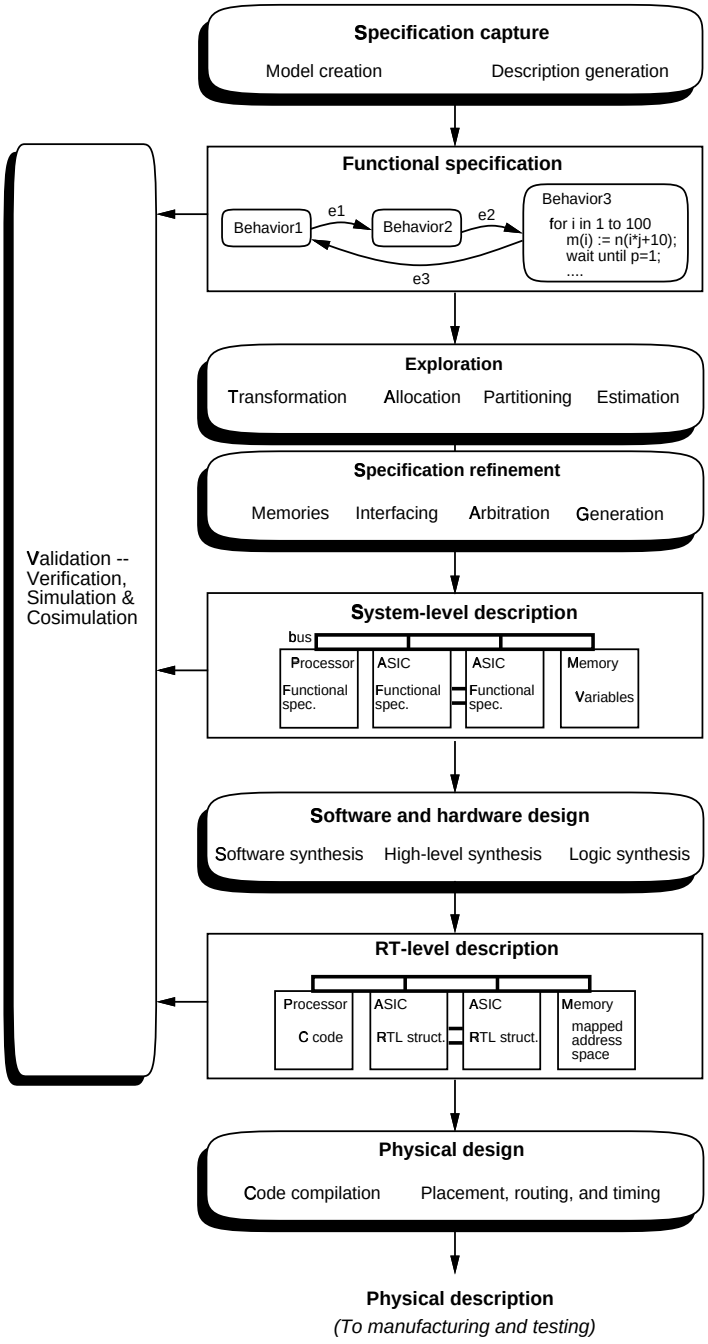
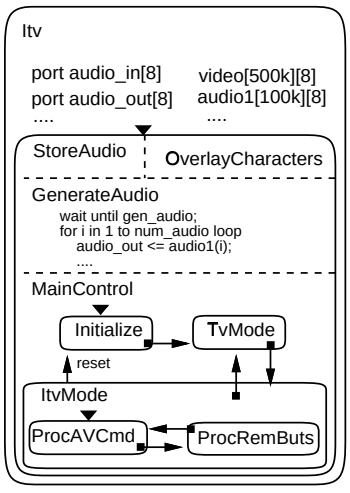


Figure 3: System-level design using hierarchical modeling

2 Specification capture

Specification capture unambiguously defines desired system functionality; in other words, given any sequence of input values to the system, a specification tells us what the system's response would be. Specification capture is a difficult problem, because (a) today's systems are complex, (b) the system's functionality is not exactly known at the outset, and (c) specification techniques are often imprecise. To make the problem worse, the specification capture stage does not receive nearly as much attention as subsequent implementation stages, and thus many functional errors are not detected until a low-level implementation is available. Unfortunately, functional errors are far more difficult to correct at the late stages of product development than during the specification stage [4]. To remedy this situation, most researchers propose the use of a formal, simulatable specification language, which allows creation of a precise specification that can be simulated, thus helping us to detect and correct functional errors at an early stage, and reducing overall design time.

Capturing a precise specification, as required by a formal language, can be difficult for complex systems. Specification capture is not a simple process of "writing down" a well-understood functionality; in contrast, it is the process of learning, understanding, organizing and defining a functionality. Specification capture consists of three tasks: model creation, description generation, and simulation. We usually must iterate through these tasks several times before we obtain a complete and correct functional specification.



(a)

```
entity ItvE is port (
      audio_in : in byte;
      audio_out : out byte;
      ....  ); end;
architecture ItvA of ItvE is begin

behavior Itv type
      concurrent subbehaviors is
   type audiomemtype is array ...
   signal audio1 : audiomemtype;
begin
   StoreAudio : ;
   GenerateAudio : ;
   OverlayCharacters : ;
   MainControl : ;

   behavior GenerateAudio
         type code is begin
      wait until gen_audio;
      for i in 1 to num_audio loop
         audio_out <= audio1(i)
      ....
   behavior MainControl type
         sequential subbehaviors is
   begin
      Initialize : (TOC, true, TvMode);
      TvMode : (TOC, true, ItvMode);
      ItvMode : (TOC, true, TvMode),
                (TI, reset, Initialize);
```

(b)

```
entity ItvTestE is end;
architecture ItvTestA
      of ItvTestE is
   component ItvE
      port (
         audio_in : in integer;
         audio_out : out integer
      );
   end component;
-- port maps
....
process begin
   -- input audio
   for i in 1 to num_audio loop
      audio_in <= audio_data(i);
      wait for atime;
   ....
   -- send audio output command
   gen_audio <= '1';
   -- check audio output
   for i in 1 to num_audio loop
      assert (audio_out=audio_data(i))
         report "audio mismatch";
      wait for atime;
   ....
```

(c)

Figure 4: ITVP specification: (a) PSM model, (b) model described in a language, (c) simulation testbench

2.1 Model creation

In order to specify a system's functionality, we must first decompose that functionality into pieces, and describe the relationships between those pieces. For example, the ITVP functionality was decomposed into various functions, such as video storing, audio storing, video generation and audio generation, and the relationships between those functions would be expressed in terms of the order in which they are executed and the data passed between them. In general, a model is a formalization of allowable pieces and their relationships. There are many models that we can use to describe a system's functionality. One model is the dataflow graph [1, 5, 6], in which functionality is decomposed into activities that transform data (such as a piece of a program), and the flows of data between those activities. Another model is the finite-state machine (FSM), which represents the system as a set of states, and a set of arcs between those states that indicate transition of the system from one state to another when certain events occur. This model has been extended to include hierarchy and concurrency [7]. A third model is communicating sequential processes (CSP) [8]. In this model, the system is decomposed into a set of concurrently-executing processes, each of which executes a sequence of program instructions that include variable assignments, loops, branches, and procedure calls. A fourth model, the program-state machine (PSM) [9], combines the previous two models by permitting each state of a hierarchical/concurrent FSM to contain actions described using program instructions. Other models include Petri-nets [10], flowcharts [6, 11], entity-relationship diagrams [12], Jackson diagrams [13], control-dataflow graphs [14], object-oriented models [15], and queueing models [16].

No one model is ideal for all classes of systems. For example, the dataflow model may be most natural to use for a system that repeats the same data transformations over time on streams of data, such as a digital-signal processing system. The FSM model may be most appropriate for a system that doesn't perform complex computations, but that must respond to complex sequences of external events, such as a control-dominated system. The CSP model is most appropriate for systems that perform complex data transformations, possibly in parallel, such as many software applications. The PSM model in many ways subsumes the FSM and CSP models, so it is appropriate not only for control-dominated systems, but also for data-dominated systems such as software applications. However, the best model for any application is the one that matches closely the characteristics of the system it models. For this reason, we must define characteristics of embedded software/hardware systems. They are: (1) hierarchy, (2) concurrency, (3) state-transitions, (4) exceptions, and (5) program instructions.

For example, Figure 4(a) illustrates a partial PSM model for the ITVP. We decompose the ITVP into concurrent processes (four are shown). We describe the *GenerateAudio* process as program instructions, whereas we describe the *MainControl* process as a state machine, which transitions based on certain exceptions. Each state is further described as concurrent processes, a state machine, or as program instructions. Because this system was most easily described using a combination of hierarchy, concurrency, state-transitions, exceptions, and sequential instructions, it is most easily captured using the PSM model in this case.

2.2 Description generation

The choice of a model is the most important factor that influences our ability to understand and define system functionality during specification. Once we've chosen the appropriate model, the system functionality must be captured in a functional specification. Such a specification may be captured in many different languages. A functional specification is easy to generate if there is a one-to-one correspondence between model characteristics and language constructs. If a language construct does not exist for a particular characteristic, then we must apply some effort to use a set of constructs that describes that characteristic, which in turn leads to a less readable description, possibly with more functional errors.

There are several languages that are commonly used to specify a system's functionality. VHDL [17] and Verilog [18] are popular standards that, through their process and sequential statement constructs, support easy description of a CSP model. They are also commonly used to describe FSMs, although neither language possesses explicit constructs to directly support state transitions. Esterel [19] is similar to those languages, adding constructs to support exceptions. Statecharts [20] supports description of hierarchical and concurrent FSMs, including exceptions. SpecCharts [21] supports capture of the CSP model, hierarchical/concurrent FSMs, and the PSM model. SDL [22], a standard of the CCITT, supports description of hierarchical dataflow diagrams with an FSM at the leaf level. Finally, Silage [23] supports easy description of dataflow models through it data stream and recurrence constructs. The table in Figure 5 summarizes several languages with respect to their ability to capture characteristics commonly found in models for embedded systems.

Language	Embedded System Features					
	State Transitions	Behavioral Hierarchy	Concurrency	Program Constructs	Exceptions	Behavioral Completion
VHDL	○	◍	●	●	○	●
Verilog	○	●	●	●	●	●
HardwareC	○	◍	●	●	○	●
CSP	○	●	●	●	○	●
Statecharts	●	●	●	○	●	○
SDL	●	◍	●	○	○	●
Silage	–	–	●	–	–	–
Esterel	○	●	●	●	●	●
SpecCharts	●	●	●	●	●	●

● Feature fully supported ◍ Feature partially supported ○ Feature not supported – Not applicable

Figure 5: Language support for conceptual model characteristics of embedded systems

As an example, Figure 4(b) shows the PSM model of the ITVP captured with the SpecCharts

language. Since SpecCharts is intended to capture the PSM model, there is a nearly one-to-one correspondence between the model constructs and the SpecCharts constructs. A language that does not support all PSM constructs, such as VHDL, will require more effort and more lines of code.

3 Exploration

Given a functional specification for a system, we must proceed to create a system-level design of interconnected components, each component implementing a portion of that specification. A design's acceptability is evaluated by how well it satisfies constraints on design metrics, such as performance, size, power and cost. Since substantial time and effort are needed to evaluate a potential design, designers usually examine only a few potential designs, often those that they can evaluate quickly because of previous experience.

By using a formal specification, we can automatically explore large numbers of potential designs rapidly. Exploration of potential designs can be decomposed into four interdependent subproblems: allocation, partitioning, transformation and estimation. We need not solve these problems in the given order; in fact, we will usually need to iterate many times before we are satisfied with our system-level design. We shall now describe each exploration subproblem separately.

3.1 Allocation

Allocation is the problem of finding a set of system components to implement the system's functions. An example is shown in Figure 6, which provides an allocation of two memories of type V100 and V500, one ASIC of type Xilinx XC4020, one Intel 8086 processor, and two buses for the ITVP example.

There are usually hundreds of components to choose from. At one extreme, we have very fast but expensive custom-hardware components, such as ASICs. At the other extreme, we have cheaper but slower general-purpose programmable microprocessors. Between these two extremes lie innumerable components that vary in cost, performance, modifiability, power, size, reliability, and design effort, including a variety of microprocessors, microcontrollers, field-programmable gate arrays, parallel processors, the newly-evolving application-specific instruction-set processors (ASIPs) [24], and hundreds of predesigned components that implement a particular function, such as memories, arbiters, DMA controllers, floating-point multipliers, and fast Fourier transforms. Adding to the number of choices are cores or megacells, in which processors or other predesigned components can be embedded within an ASIC component. New components surface every year. These components can be characterized by: (1) instruction sets, (2) parametrized descriptions, or (3) hardware object number. General purpose processors are characterized by instruction sets. Any part of a specification implemented with a processor must be converted to a sequence of instructions. Many special purpose components, such as floating-point multipliers, Fourier transforms, and DMA controllers, are characterized with a parametrized function. Those components execute the same program with slight variations defined by the parameters. Any part of a specification executed on these components must be transformed to match exactly the parametrized descriptions. Finally, ASIC's, FPGA's and gate arrays are characterized by the numbers of hardware objects, such as transistors, combinational-logic blocks or gates, that they can contain. Any part of a specification implemented on an ASIC must be converted to

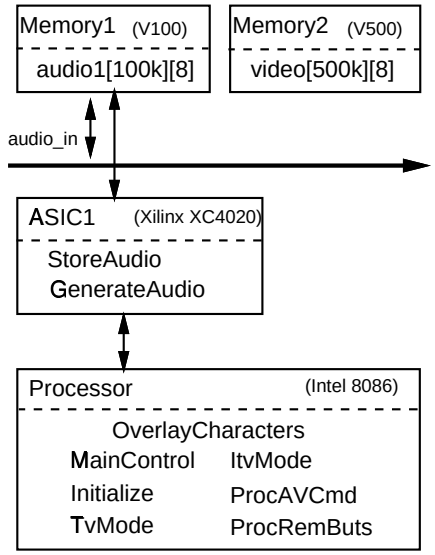


Figure 6: ITVP allocation and partition example

an interconnection of register-transfer and logic level components.

The designer's job is therefore to choose the proper mix of components from an enormous number of possibilities. Newly evolved system-design tools assist in making this choice. In [25], an approach is presented to automatically determine an allocation of processors on which to implement a given set of functions, such that performance and cost constraints are satisfied. In [9], an environment is described that provides rapid feedback of performance, size and cost metrics for a given distribution of functions on any allocation of processors, ASICs, memories and buses. Tools described in [26, 27] assist in mapping a specification onto a fixed allocation of one processor, one ASIC, one memory and one bus, while tools in [28, 29] assist in mapping onto a single processor with multiple ASICs.

3.2 Partitioning

Given a functional specification and an allocation of system components, we need to partition the specification and assign each part to one of the allocated components. In fact, we can distinguish three types of specification objects that must be partitioned separately. One type of object is a *variable*, which stores data values. Variables in the specification must be assigned to memory components. The second type of object is a *behavior*, which transforms data values. A behavior may consist of programming statements, such as assignment, if and loop statements, and it generates a new set of values for a subset of variables. Behaviors must be assigned to custom or standard processors. The third type of object is the *channel*, which transfers data from one behavior to another. Channels must be assigned to buses. Such specification partitioning is driven by the requirement that we satisfy constraints. Constraints may exist on the number of program bytes for a processor or microcontroller, the number of gates or pins on an ASIC, the number of words in a memory, the execution time of a function, or the bitrate of an input/output port.

There are two very different approaches to system partitioning. In **structural partitioning**, the system is first implemented with fine-grained structural objects, such as gates, and those objects are then partitioned among several custom components. While easy to automate, this approach does not consider software implementations. It also does consider inter-component delay in the design since the design is completed before partitioning. In a very different approach, called **functional partitioning**, the various system functions are first partitioned into groups of functions, where each group is assigned to a system component. Each group is then implemented as software (for a processor component) or as hardware (for an ASIC component).

In developing a functional partitioning technique, we must consider several issues. First, we must define **object granularity**, which defines the smallest indivisible functional objects used during partitioning, such as jobs, processes, subroutines, loops, blocks of statements, statements, arithmetic-level operations or boolean expressions. Higher granularity means fewer objects, which in turn enables easier interaction, faster runtime for partitioning algorithms, and faster estimations, but fewer possible partitions.

Second, we must select the **design metrics** that will be used to define a good partition. Common metrics include monetary cost, performance, communication rates, power consumption, silicon area, package size, testability, reliability, program size, data size, and memory size.

Third, we must select a model with which we will **estimate** metric values. Estimation is necessary because we can't spare the hours or days necessary to build a design for each possible partition, especially if we wish to examine hundreds or thousands of possibilities.

Fourth, we need to combine multiple estimated metric values into a single *cost* value that defines a partition's "goodness", by using an **objective function**. Because those metric values often compete with one another (i.e., when one value increases, another value decreases), we usually need to weigh each value in the objective function by its relative importance to the overall design. An objective function thus gives us a way to compare two partitions, and to select one that satisfies constraints.

Fifth, we need **partitioning algorithms** to efficiently explore a subset of the huge number of possible partitions. Commonly used classes of algorithms include clustering algorithms [30], iterative-improvement algorithms [31, 32], genetic algorithms [33], and custom algorithms [26, 34]. Some algorithms are fast, such as clustering, while other algorithms are slower but often find better solutions, like genetic algorithms.

A variety of techniques have evolved to assist the designer perform functional partitioning. We can form three categories of techniques: hardware partitioning, hardware/software partitioning, and interactive partitioning environments. The hardware partitioning techniques aim to partition functionality among hardware modules, such as among ASICs or among blocks on an ASIC. Most such techniques partition at the granularity of arithmetic operations, differing in the partitioning algorithms employed. Clustering algorithms are used in [35, 36], integer-linear programming in [37, 38], manual partitioning in [39], and iterative-improvement algorithms in [40]. Other techniques for hardware partitioning operate at a higher-level of granularity, such as in [41], where processes and subroutines are partitioned among ASICs using clustering and iterative-improvement algorithms.

Hardware/software partitioning techniques form the second functional partitioning category. These techniques focus on partitioning functionality among a hardware/software architecture. In [29] and [42], overviews of the hardware/software partitioning problem are provided, including discussion on the issues of granularity and estimation. The technique in [43] partitions at the statement-level

of granularity using clustering algorithms, while the approaches in [26], [27], and [34] partition at the statement, statement sequence, and subroutine levels, respectively, using iterative improvement algorithms.

In the third category are general environments that support interactive or automated partitioning of all three types of specification objects (variables, behaviors, and channels) among a variety of systems components (such as processors, ASICs, memories, and buses). An example of such an environment is found in [9]. Such an environment can be used for hardware partitioning as well as for hardware/software partitioning.

Figure 6 shows an example partition of the main functional objects from Figure 4(b) among the given system component allocation. Since audio must be stored and generated at very high speeds, the functions responsible for such storing and generation are to be implemented with an ASIC. Since audio and video can be generated simultaneously, the audio and video data are to be stored in separate memories, thus preventing contention for access to a single memory. The remaining functions are to be implemented on the processor, since they don't require the speed offered by an ASIC and hence can be implemented more cheaply on the processor.

3.3 Transformation

Up to this point, we have assumed that the specification consisted of functions that could be implemented one-to-one on system components. However, those functions are derived from a specification intended for readability, so implementing them directly may not lead to the best design. For example, we may have introduced a procedure in the specification for readability, but implementing a distinct hardware module to implement that procedure may produce a performance bottleneck. Instead, we may prefer to inline that procedure so that each caller implements the functionality of that procedure internally, resulting in better performance. As another example, we may create a specification consisting of two concurrent processes, but implementing a separate controller for each process might be too costly. Instead, we may prefer to merge those two processes into one process, so that they will execute serially in a single controller.

Inlining procedures and merging processes are common examples of specification transformations. A transformation reorganizes the specification, thus changing the organization of any subsequent implementation. Other transformations include flattening hierarchy, splitting processes, grouping statements into procedures, and merging variables into arrays.

Some approaches have evolved to help automate transformations. In [44], several transformations are proposed, including procedure inlining and process splitting, to allow a designer to trade-off area and performance. In [45], a transformation technique is described for merging two processes into one. The technique provides for a fine-grained scheduling of operations from the two processes, meaning that good performance can be achieved, while reducing hardware size from two controllers down to one controller. A variety of optimizing transformations with origins in software compilation are also applied to the internal representation of behavior in many design tools [46, 47].

3.4 Estimation

We would like to evaluate metric values, such as performance and area, for a large number of system-level designs, in order to find a design that best satisfies constraints. We can derive those metric values from the system's implementation, but that requires far too much time if we wish to examine more than a few designs. Instead, we can estimate metric values, by creating a rough (and thus quick) hardware and software implementation for each system component.

Accuracy and speed are competing factors in the development of an estimator. Accuracy comes from creating a more complete implementation, while speed comes from creating a less-detailed implementation. For example, we could rapidly estimate hardware size by allocating and then counting functional units, and then using quick statistical estimates for the number of registers, multiplexors, and controller gates. Clearly, the time saved over generating a complete implementation is at the expense of less accuracy. In general, only rough estimates are needed during system design. For example, if we wish to see if a set of functions can be implemented using a gate array with 100000 gates, we only need to get an idea of whether the functions require much more or much less than 100000 gates.

We describe methods for estimating the common metrics of hardware size, software size, and performance. As we will see, software/hardware size and performance estimation techniques are not completely accurate because the mapping of a behavioral description into hardware or software is not straightforward (one-to-one). The complexity is introduced by optimization on different abstraction levels. Since algorithms used in optimizing compilers are not known during estimation, it is difficult to predict code reduction or performance enhancement due to the optimization. Also, it is difficult to predict performance due to architectural features such as caching, pipelining, and multiple instruction issue, since estimators do not compute dynamics of the code and data. Similarly, in hardware it is difficult to predict performance and size due to logic optimization of control, library mapping in datapaths, state minimization, etc.

The hardware size of a given set of functions can be estimated by roughly synthesizing a controller and datapath to implement those functions, by applying algorithms for scheduling operations into control steps, allocating functional, storage, and interconnect units, and binding data values and operations to units. In other words, we need to determine the number and type of register-transfer objects required, including: registers, register files, functional units, multiplexors, buses, wires, state registers and control logic. Unfortunately, the algorithms for determining the number and type of objects required are computationally expensive, so estimators usually only generate a subset of objects. Once the required objects have been determined, we can estimate size for a variety of technologies. For an FPGA implementation, we would estimate the total number of combination-logic blocks (CLB) by summing CLBs used for each object. For a gate array, we would sum the equivalent gates needed for each object. For a custom implementation, we would sum the transistors for each object, or we would compute the bounding box area after performing object placement and routing.

The software size of a given set of functions may be estimated by compiling the functions into the instruction set of a given processor. If such compilers are unavailable, then we could alternatively compile into a generic instruction set. If we previously tabulate the number of the given processor's instructions needed to implement each generic instruction, then we could estimate software size by summing tabulated numbers for all generic instructions in the compiled generic code.

There are generally two types of performance metrics that we are interested in estimating: execution-time of a function, and communication bitrates of a bus. For each type, we may be interested in minimum, maximum, or average values. Such performance metrics can be estimated at various levels of accuracy. For coarse but quick estimates we can use queueing models. In this approach, we (manually) associate statistical numbers relating to the execution time and communication frequencies of each function on a given system component type, and then we use queuing models to determine statistical execution-times and communication rates for the overall system [16, 48].

For somewhat more accurate performance estimates, we can use program-level models. In this case, determining minimum and maximum performance requires analysis of the possible paths through each function in the specification, which is hard for all but very simple functions. Determining average performance requires dynamic profiling, in which we simulate the specification with typical input stimuli and determine the branch probabilities. Once we have determined the possible paths or the branch probabilities, we must determine the performance for the given set of system components. For functions assigned to hardware components, estimating performance requires that we map the functions to RT-level units, and then determine the minimum/maximum or average frequency of execution of each control step from the paths or branch probabilities, respectively. Then, the expected number of control steps times the clock cycle produces the execution time, and the frequency of each control step informs us of the communication rates. On the other hand, for functions assigned to software components, estimating performance requires that we compile the functions into the instruction set of the given processor, and then determine the minimum/maximum or average frequency of execution of each instruction. Then, the expected number of executions of each instruction times the execution time of each instruction produces the total execution time, and the frequency of communication instructions informs us of the communication rates. Note that, as was the case during software size estimation described above, we can again use generic instructions and tabulation to estimate software performance when a compiler for a given processor is unavailable.

Software performance estimation for some processors requires even more effort to account for pipelining, caching and interrupts. For a pipeline, the rate of execution depends heavily on the way that instructions are paired. We might therefore seek additional information on the execution time of each instruction based on what statement follows or preceeds it, in order to obtain more accurate estimates. For caching, each memory access may take a different amount of time depending on whether or not the data being accessed is found in the cache. We might use statistical hit/miss ratios to determine average access time, or we can assume for worst-case estimates that the data is never in the cache, or we can analyze the data-replacement policy in use to possibly determine if the data will be in the cache. For interrupts, accuracy might be improved if we somehow determine the frequency of interrupts and the time to service each.

Finally, software performance estimation may include the case when there are multiple concurrent tasks assigned to a single processor. In this case, we need to take into account the fact that each task will only be able to execute on the processor for particular intervals of time.

A variety of estimation tools and techniques have been suggested. For hardware estimation, several techniques estimate the size and performance of a group of arithmetic operations. In [40], the estimates are obtained by summing previously-assigned weights associated with each operation. In [35] and [36], the estimates are obtained by roughly synthesizing hardware to implement the operations. In [39], multiple groups of operations are considered; first, a set of possible rough

implementations are determined for each group, and then a global analysis picks one implementation for each group such that global constraints on size and performance for all the groups are satisfied. Other hardware estimation techniques estimate for a group of coarse-grained functions, rather than arithmetic operations. In [9], estimates are obtained by roughly synthesizing hardware for each group of functions, and a special data structure is used that permits rapid, incremental modification of the hardware as functions are moved between groups.

For software performance estimation, techniques in [49] and [50] use dynamic profiling to estimate execution time during hardware/software partitioning. Techniques in [51] and [52] perform path analysis to determine minimum or maximum execution times, the latter with the help of user annotations. In [53], methods are described for reasoning about program execution time. A summary of software performance estimation techniques can be found in [54].

Figure 7 illustrates estimated values for several design metrics for the ITVP system-level design of Figure 6(b). The designer (or automated algorithms) can use this information to decide how to improve the design. For example, noting that the program-memory size for the processor is currently violated and that there are 2000 gates available on the ASIC, the designer may try moving a function from the processor to the ASIC.

Metric	Estimate	Constraint
Size(ASIC1)	8000 gates	<10000
Size(Processor)	5500 bytes	<4000
Size(Memory1)	100k bytes	<100k
Size(Memory2)	200k bytes	<500k
Bitrate(audio_out)	10 Mb/s	>8 Mb/s
...		

Figure 7: ITVP estimates example

4 Specification refinement

After creating a specification of system functionality, and exploring alternative system-level designs, we must refine the initial functional specification by incorporating the implementation style and details that we have selected. We call such a refined specification a system-level description, since it is a mixture of structural and functional parts. Such a description will consist of interconnected system components, where each component will itself be functionally specified. Such refinement is an important concept; in past approaches, only one description was generated, close to the point when the design was ready for manufacturing. This practice is presently abandoned in favor of hierarchical modeling, in which successively more detailed descriptions are derived during the design process from more abstract descriptions.

In order to create a system-level description, several details must first be added to the system's functionality, including details related to memories, interfacing, and arbitration, which we shall now describe.

4.1 Memories

During the exploration stage of system design, we may have grouped variables for storage in a particular memory. These variables are no longer directly accessible by each process. Instead, we must create a memory description, move the variable declarations to that memory description, and then insert the memory access protocol into every part of the system description that accesses a variable in the memory. Other details, such as specific memory addresses for each variable, may also be added to the newly created memory description.

4.2 Interfacing

Partitioning functions among system components usually introduces the need to communicate data between components. For example, a specification may include a function that reads a variable. If the function and the variable are assigned to different components, then the variable's value will need to be transferred over a bus. The addition of such specification details that describe communication between components is called interfacing. There are several problems that must be solved when interfacing, including bus-size generation, protocol generation, and protocol matching.

Bus-size generation determines the width of the bus that will implement a group of communication channels, given a set of bitrate and buswidth constraints. Although we assigned a width to each bus during the allocation step of system design, during this refinement step we can optimize the bus width to use as few wires as possible while still satisfying performance constraints. Approaches to bus generation are described in [55, 56].

Protocol generation determines the exact mechanism for transferring data over a bus of a fixed width. We must determine the type of control to be used, such as a full handshake, a half handshake, or a fixed-time access. We must also determine how to distinguish data destined for different locations, perhaps by sending an address over the data lines first, or by adding additional address wires. Finally, we must determine how to decompose the data for serial transmission, in case the bus width is narrower than the number of bits of the data that we wish to transfer.

Protocol matching enables communication between components in which one component uses a fixed protocol. Such a case arises when we implement certain functions in software running on an off-the-shelf processor. If the other component is an ASIC, then that ASIC must implement a protocol that is the complement of the fixed protocol. If the other component also uses a fixed but different protocol, then we need to insert hardware between the two components that can receive and send data with each protocol (such hardware is called a transducer).

There are several techniques developed that address the problems of interfacing. In [57, 58], techniques for specifying protocols are described that extend traditional timing diagrams. In [59, 60], techniques are described for creating transducers. In [61, 62], an approach is introduced in which the detailed I/O structure and protocols of library modules are hidden from a designer, who can simply interconnect those modules using high-level primitives. Interface controllers are then synthesized automatically to permit communication between modules.

4.3 Arbitration

When concurrently-executing processes access the same resource, such as a bus or a memory, we need to ensure that only one process accesses that resource at a given time. Arbitration resolves simultaneous access requests by granting permission to only one process at a time. During refinement, we must insert new arbiter processes into the specification where needed. There are two types of schemes for determining priority during arbitration. A fixed-priority scheme assigns a priority to each process statically; the process's priority never changes. A dynamic-priority scheme determines the priority of a process at run-time, based on the pattern of accesses of the processes. A round-robin dynamic-priority scheme assigns the lowest priority to the process that was most recently granted access. A first-come-first-served dynamic-priority scheme grants access to processes in the order that they requested access. Fixed-priority schemes have simple implementations, but may leave a low-priority process waiting for very long periods of time, even forever if higher-priority processes continuously request access. Dynamic-priority schemes have more complex implementations, but ensure fair access for all processes.

4.4 Generation

After introducing the above refinement details, we need to refine the functional specification into a system-level description. In doing so, we must ensure that the new description is readable, modifiable, and modularized, and that different designers can implement different parts. We must also ensure that the description is suitable for further processing by synthesis or compilation tools. Finally, we must ensure that the description is simulatable, so that we can continue to verify system functionality.

For example, Figure 8 shows a SpecCharts system-level description for the ITVP, reflecting the allocation and partition of Figure 6. Note that we now include the memory, ASIC, and processor components, as well as declarations of buses and control signals among those components. We also describe the functionality to be implemented on each component. For example, ASIC1 is to implement the *StoreAudio* and *GenerateAudio* functions. Note that the *GenerateAudio* function has been modified: the read of *audio1* has been replaced by a procedure call that executes a protocol (*ReadMemory1* that reads *audio1* from *Memory1* over *bus1*).

Algorithms for generating a system-level description after partitioning are found in [3].

```
entity ItvE is port (
        audio_in : in integer;
        audio_out : out integer
        ....  ); end;
architecture ItvA of ItvE is begin

behavior Itv type
        concurrent subbehaviors is
    signal  bus1 : bustype;
    signal bus1req, bus1ack : bit;
    signal  bus2 : bustype;
begin
    Memory1 : ;
    Memory2 : ;
    ASIC1 : ;
    Processor : ;

    behavior Memory1 type code is
        signal audio1 : audiomemtype;
    begin
        -- code to control access to
             audio1 over bus1
        ....
    behavior ASIC1
        type concurrent subbehaviors is
    begin
        StoreAudio : ;
        GenerateAudio : ;

        behavior GenerateAudio
              type code is begin
            wait until gen_audio;
            for i in 1 to num_audio loop
                a := ReadMemory1(bus1)
                audio_out <= a;
            ....
    behavior Processor
        type concurrent  subbehaviors is
    begin
        OverlayCharacters : ;
        MainControl : ;

        behavior
```

Figure 8: ITVP refined specification example

5 Software synthesis

A system-level description usually possesses complex features not found in traditional programming languages, such as the C language. A typical compiler cannot usually compile these features. Software synthesis is the task of converting a complex description into a traditional software program that can be compiled by traditional compilers.

One such complex feature of system-level descriptions is the definition of concurrent tasks. If two concurrent tasks are mapped to a single processor, then the tasks must be scheduled to execute sequentially [63, 64]. In such scheduling, it is important to ensure that every task has a chance to execute, or in other words, that no task is "starved." Another issue is minimizing the amount of "busy-waiting": the time the processor spends waiting for some external event. A third issue is ensuring that timing constraints for each task are satisfied. For example, data may be arriving at a specific rate and must be captured and processed by a given task, or a task may have to output data at a certain rate to ensure satisfactory system performance. Such tasks must be guaranteed a minimal rate of execution. Several techniques exist for performing such scheduling [54, 26, 65, 64]. One uses a global task scheduler, which activates each task (or portion thereof) by calling each as a subroutine. This technique may require overhead to maintain the state of each task as it switches from one to the other. Another technique reduces this overhead by maintaining data locally within each task, and modifying each task to relinquish control of the processor whenever it must wait for an event or an interrupt occurs. Choosing a technique usually involves a tradeoff between performance and program size.

6 Hardware synthesis

After generating a refined system-level description, we need to create hardware for the parts of the description that are to be implemented on custom components, such as on ASIC's or FPGA's. Such hardware synthesis incorporates high-level synthesis, FSM synthesis, logic synthesis and technology mapping. High-level synthesis transforms a system component's functional description into a structure of register-transfer components such as registers, multiplexors, and ALU's. Such a structure usually consists of two parts: a controller implementing a finite-state machine, and a datapath executing arithmetic operations. We refer to such a structure as a finite-state machine with datapath, or FSMD [1, 66]. The controller controls register transfers in the datapath and generates signals for communication with the external world.

There are several interdependent tasks that make up high-level synthesis. The input executable specification is first compiled into an internal representation, such as one described in [14, 67, 68, 69]. The internal representation exposes control and data dependencies between arithmetic operations, such as between additions and comparisons. Allocation selects, from a register-transfer component database, the storage, function and bus units to be used in the design. Scheduling maps operations to control steps, each of which usually represents one clock period or clock phase. Scheduling is necessary since all operations usually cannot be executed at once due to data dependencies or due to a limited number of units capable of executing particular operations. Binding maps scalar and array variables to registers and memories, operations to function units, and transfers to buses. A variety of algorithms, tools and environments for high-level synthesis are described in [1, 70, 71].

FSM and logic synthesis transforms a finite-state machine (FSM) to a hardware structure consisting of a state-register, and of a combinational circuit, which generates the next state as well as the controller's outputs. The tasks involved in creating such a structure include state minimization, state encoding, logic minimization and technology mapping. State minimization reduces the number of states in an FSM by replacing equivalent states with a single state. Two states are equivalent if the sequence of outputs for any sequence of inputs does not depend on which of the two states we start in. State minimization is important since the number of states determines the size of the state register and control logic. State encoding assigns binary codes to symbolic states. The goal is to obtain a binary code that minimizes the controller's combinational logic. Logic minimization is used after encoding to reduce the size or delay of the combinational logic. Technology mapping transforms a technology-independent logic network produced by the logic minimizer into a netlist of standard gates from a particular gate library. A variety of techniques for sequential and logic synthesis can be found in [2, 72, 73, 71].

7 Simulation and Cosimulation

We need to somehow validate that our initial specification is complete and correct. A specification is complete if it includes all possible sequences of inputs that the environment might provide to our system. A specification is correct if it generates expected output for every such input sequence. To validate the correctness and completeness of our specification, we can apply formal verification techniques or simulation techniques. Formal verification techniques usually involve making assertions about the specification, and then proving that those assertions hold. For example, we may assert that all states of an FSM are reachable, and we may then use a theorem prover to prove that assertion. Simulation involves executing the specification, and then comparing the generated output sequence with the sequence of expected values. Today, simulation is the most common verification technique. Presently, neither verification or simulation techniques entirely validate the completeness of a specification, because there are far too many possible input sequences for even moderately-sized systems.

As an example of simulation, Figure 4(c) shows a simple test case for the ITVP, written in VHDL. After instantiating an ITVP component, we input a sequence of audio data, provide the ITVP with a command to output that data, and then we check that the output data is the same as the input data; if not, then we generate an error message.

Simulation is not only useful to verify the initial functional specification, but also to verify the more detailed design descriptions generated throughout the design process. In particular, we need to: (a) ensure that a design's functionality is consistent with the initial specification, (b) detect possible performance bottlenecks arising from the mapping of the abstract specification to real components with limited resources, and (c) ensure that detailed timing constraints for communication and synchronization are satisfied.

Simulation of the more detailed descriptions may take place at various levels of abstraction. In the design process shown in Figure 3, we used four different models of the system. The functional specification described only functionality without any implementation. This model is used for product definition, customer contract, and marketing. The system-level description is the bus model of the system and can be used for performance estimation and detection of bottlenecks. The RT-

level description gives the hardware design on clock-cycle level and the processor model on the instruction-set level. It can be used for checking application software as well as correctness of the ASIC architecture. The last model is the physical description that allows checking of detailed electrical and timing properties of ASICs and standard chips. In order to investigate different issues during the design process, we model different parts of the system on different abstraction levels. This hierarchical modeling typically requires different simulators, Integrating simulations of a variety of models is called cosimulation. A common example of cosimulation is that of simulating interconnected register-transfer or logic level components (hardware) along with instructions running on a processor (software), i.e., hardware-software cosimulation.

There are two competing goals of hardware-software cosimulation: speed and correctness. Speed refers to the rate at which simulated time proceeds. Because simulations usually proceed orders of magnitude slower than a real implementation, speed is crucial if we are to simulate for a reasonable number of input sequences. Correctness refers to the generation of proper output values by the simulation. Incorrect values may arise when different parts of the system are being simulated separately at different speeds, and care is not taken to ensure that shared data is accessed in the proper order by the various parts (e.g., ensuring that one part does not read a memory location before another part was supposed to have updated that location). A third goal that usually competes with speed is interactive debugging, or the ability to step through execution of the system, examine intermediate values, backtrack, etc., for debugging purposes.

Speed for the hardware and software parts varies depending on the chosen simulation technique. For software, the slowest but most debug-amenable approaches are simulation approaches. In one simulation approach, we execute the software on a model of the target processor using a hardware simulator. We can write such a model on one of several abstraction levels, including instruction-set, register-transfer, and gate levels; higher abstraction levels provide shorter simulation time at the expense of detailed timing accuracy. A faster simulation approach would be to execute the software on a custom-built simulator for the target processor. Instead of simulating, we could use faster, execution approaches. In one execution approach, we could execute the software on our development processor (e.g., our workstation), assuming that the description is written in a high-level language like C that can be compiled to the processor's instruction set). Alternatively, we could just execute the software on the target processor. A common hybrid approach is called in-circuit emulation. An emulator consists of a package with the same input/output configuration as the target processor, along with a tool on which we can not only run the software, but also interactively debug it from our workstation.

For hardware, the most common simulation technique uses an event-driven hardware simulator. An alternative is to use a hardware emulator. In some cases where speed is extremely important, we can create an FPGA implementation.

Correctness is maintained only if we ensure that the software and hardware simulations access shared data in the proper order. The most straightforward method is to create a hardware model of the target processor, and then simulate the hardware and software parts in synchrony, using the same hardware simulator; the hardware simulator thus serves as a "supervisor" that ensures that data is accessed in the proper order. However, to speed up the simulation, we would like to use one of the software execution options in the previous paragraph. In such cases, one approach to ensure correctness is to explicitly synchronize all data transfers between the hardware part and the software

part, so that no supervisor is necessary to ensure correct data access ordering. One common method for describing such transfers is to use message passing. In message passing, all data transfers occur by a process explicitly *sending* data to another process that explicitly *receives* that data. Alternatively, if it turns out that the software part is the only initiator of data transfers, then the executing software serves as the supervisor. Conversely, when the hardware is the only initiator of data transfers, then the hardware simulator serves as the supervisor.

Several techniques have been published for cosimulation. In [74], a categorization of various methods of cosimulation is introduced, and the applicability of each method for a variety of applications is discussed. In [29] and [75], software is executed on the development processor, and that software communicates with a hardware simulator through Unix interprocess communication mechanisms, using message passing communication. In [76], the software is executed on a processor simulator, which is connected with a hardware simulator. Facilities exist between the simulators to support message passing communication between the hardware and software parts. In [27, 49], cosimulation can be performed using one of three techniques. One technique, used primarily for timing analysis, uses estimators to predict the performance of parts of the specification (another cosimulation estimator is used in [9]). A second technique simulates a cycle-accurate processor model in conjunction with the hardware, using a hardware simulator. A third technique uses a prototyping board that contains a RISC processor and several FPGA's. In [42], a multiparadigm simulation environment (Ptolemy) is described, which supports co-simulation of different domains such as synchronous dataflow and digital hardware, and defines mechanisms for transfering data and synchronizing timing between those domains. The environment can thus be used to support a variety of hardware/software cosimulation techniques. One method discussed uses a cycle-accurate functional processor model, a digital-hardware domain representation of the RTL-level hardware components, and the synchronous dataflow domain for functional abstraction of some analog hardware components. Thus, the mixed hardware/software system can be simulated within an integrated environment. Another method is discussed in [77], in which a new Ptolemy domain is defined for simulating processes that communicate via message passing. This domain is used for simulation of hardware and software parts, and techniques are also described for integrating physical implementations of some processes with this simulation.

8 Conclusions

In the past, the design methodology was quite informal, with design capture and simulation late in the design cycle, just before physical design was to start. Such a methodology was tolerable since design complexity was low to medium, and since new generations of products were introduced only every 2-3 years. With increased complexity and shorter time-to-market, the old methodology is no longer acceptable. We argued in this article that for embedded software/hardware systems, a new methodology, based on a hierarchy of models on different levels of abstraction, is necessary. In such a methodology, we start with a formal functional specification, and derive the next lower-level model by exploring certain implementation issues and then refining the higher-level model with implementation selections made through exploration. Such a specify-explore-refine methodology can help managers and designers cope with today's product development requirements. It can lead to a substantial productivity gain due to the early detection of functional errors, thus reducing costly iterations

when errors are detected late in the design cycle. A productivity gain is also obtained by speeding exploration through automatic estimation of different alternatives. The proposed methodology leads to excellent documentation of the system's desired functionality as well as the design decisions that were made, making redesign a much easier task. It encourages concurrent engineering, since the various system components possess precise functional descriptions derived from an overall system specification, which simplifies integration as well as design changes during implementation. It enables marketing departments to rapidly predict a system's size, performance and design time, to determine the feasibility of a given product. Such rapid prediction also helps an engineering manager allocate human resources to a design.

We have tested the above methodology on a medium complexity (50,000 gates) fuzzy logic controller and succeeded to reduce design cycle from conceptualization to manufacturing to approximately 100 hours [78]. Such design is estimated to take about six months using standard methodology. As tools and techniques for the specify-explore-refine methodology improve, we believe that the design cycle of high-complexity systems can be reduced to several hundred hours instead of the present 12-18 month design cycle. In other words, if the goal is precisely defined and the design process well understood, then the design should be straightforward and easily manageable.

9 Acknowledgements

We would like to thank Sanjiv Narayan of Viewlogic and Jie Gong of UC Irvine for their substantial contributions to the ideas and techniques described in this article. We would also like to thank Joerg Henkel of Technische Universitaet Braunschweig, Asawaree Kalavade of UC Berkeley, and Rajesh Gupta of the University of Illinois for their helpful discussions and comments.

References

[1] D. Gajski, N. Dutt, C. Wu, and Y. Lin, *High-Level Synthesis: Introduction to Chip and System Design.* Boston, Massachusetts: Kluwer Academic Publishers, 1991.

[2] G. DeMicheli, A. Sangiovanni-Vincentelli, and P. Antognetti, *Design Systems for VLSI Circuits: Logic Synthesis and Silicon Compilation.* Martinus Nijhoff Publishers, 1987.

[3] D. Gajski, F. Vahid, S. Narayan, and J. Gong, *Specification and design of embedded systems.* New Jersey: Prentice Hall, 1994.

[4] D. Gabel, "Software engineering," *IEEE Spectrum*, pp. 38–41, January 1994.

[5] T. DeMarco, *Structured Analysis and System Specification.* New York: Yourdon Press, 1979.

[6] W. S. Davis, *Tools and Techniques for Structured Systems Analysis and Design.* Reading, Massachusetts: Addison-Wesley, 1983.

[7] D. Harel, "Statecharts: A visual formalism for complex systems," *Science of Computer Programming 8*, 1987.

[8] C. Hoare, "Communicating sequential processes," *Communications of the ACM*, vol. 21, no. 8, pp. 666–677, 1978.

[9] D. Gajski, F. Vahid, and S. Narayan, "A system-design methodology: Executable-specification refinement," in *Proceedings of the European Conference on Design Automation (EDAC)*, 1994.

[10] J. L. Peterson, *Petri Net Theory and the Modeling of Systems*. Englewood Cliffs, New Jersey: Prentice-Hall, 1981.

[11] J. Sodhi, *Computer Systems Techniques: Development, Implementation and Software Maintenance*. Blue Ridge Summit, Pennsylvania: TAB Professional and Reference Books, 1990.

[12] T. J. Teorey, *Database Modeling and Design: The Entity-relationship Approach*. San Mateo, California: Morgan Kaufman Publishers, 1990.

[13] A. Sutcliffe, *Jackson System Development*. New York: Prentice-Hall, 1988.

[14] A. Orailoglu and D. Gajski, "Flow graph representation," in *Proceedings of the Design Automation Conference*, pp. 503–509, 1986.

[15] G. Booch, *Object-oriented Design with Applications*. Redwood City, California: Benjamin/Cummings, 1991.

[16] W. Giffin, *Queueing: Basic Theory and Applications*. Columbus, Ohio: Grid Inc., 1978.

[17] IEEE Inc., N.Y., *IEEE Standard VHDL Language Reference Manual*, 1988.

[18] D. Thomas and P. Moorby, *The Verilog Hardware Description Language*. Kluwer Academic Publishers, 1991.

[19] N. Halbwachs, *Synchronous Programming of Reactive Systems*. Kluwer Academic Publishers, 1993.

[20] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, and A. Shtul-Trauring, "STATEMATE: A working environment for the development of complex reactive systems," in *Proceedings of the International Conference on Software Engineering*, 1988.

[21] S. Narayan, F. Vahid, and D. Gajski, "System specification with the SpecCharts language," in *IEEE Design & Test of Computers*, Dec. 1992.

[22] F. Belina, D. Hogrefe, and A. Sarma, *SDL with Applications from Protocol Specifications*. Prentice Hall, 1991.

[23] P. Hilfinger and J. Rabey, *Anatomy of a Silicon Compiler*. Kluwer Academic Publishers, 1992.

[24] J. Praet, G. Goossens, D. Lanneer, and H. DeMan, "Instruction set definition and instruction selection for ASIPs," in *Proceedings of the International Workshop on High-Level Synthesis*, pp. 11–16, 1993.

[25] S.Prakash and A. Parker, "Synthesis of application-specific multiprocessor architectures," in *Proceedings of the Design Automation Conference*, pp. 8–13, 1991.

[26] R. Gupta and G. DeMicheli, "Hardware-software cosynthesis for digital systems," in *IEEE Design & Test of Computers*, pp. 29–41, October 1993.

[27] R. Ernst, J. Henkel, and T. Benner, "Hardware-software cosynthesis for microcontrollers," in *IEEE Design & Test of Computers*, pp. 64–75, December 1994.

[28] M. Srivastava and R. Brodersen, "Rapid-prototyping of hardware and software in a unified framework," in *Proceedings of the International Conference on Computer-Aided Design*, pp. 152–155, 1992.

[29] D. Thomas, J. Adams, and H. Schmit, "A model and methodology for hardware/software codesign," in *IEEE Design & Test of Computers*, pp. 6–15, 1993.

[30] S. Johnson, "Hierarchical clustering schemes," *Psychometrika*, pp. 241–254, September 1967.

[31] B. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *Bell System Technical Journal*, February 1970.

[32] S. Kirkpatrick, C. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, no. 4598, pp. 671–680, 1983.

[33] J. Filho and P. Treleaven, "Genetic-algorithm programming environments," *IEEE Computer*, vol. 27, pp. 28–43, June 1994.

[34] F. Vahid, J. Gong, and D. Gajski, "A binary-constraint search algorithm for minimizing hardware during hardware-software partitioning," in *Proceedings of the European Design Automation Conference (Euro-DAC)*, 1994.

[35] M. McFarland and T. Kowalski, "Incorporating bottom-up design into hardware synthesis," *IEEE Transactions on Computer-Aided Design*, September 1990.

[36] E. Lagnese and D. Thomas, "Architectural partitioning for system level synthesis of integrated circuits," *IEEE Transactions on Computer-Aided Design*, July 1991.

[37] C. Gebotys, "An optimization approach to the synthesis of multichip architectures," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 2, no. 1, pp. 11–20, 1994.

[38] Y. Chen, Y. Hsu, and C. King, "MULTIPAR: Behavioral partition for synthesizing multiprocessor architectures," in *IEEE Transactions on Very Large Scale Integration Systems*, pp. 21–32, 1994.

[39] K. Kucukcakar and A. Parker, "CHOP: A constraint-driven system-level partitioner," in *Proceedings of the Design Automation Conference*, 1991.

[40] R. Gupta and G. DeMicheli, "Partitioning of functional models of synchronous digital systems," in *Proceedings of the International Conference on Computer-Aided Design*, pp. 216–219, 1990.

[41] F. Vahid and D.Gajski, "Specification partitioning for system design," in *Proceedings of the Design Automation Conference*, pp. 219–224, 1992.

[42] A. Kalavade and E. Lee, "A hardware/software codesign methodology for DSP applications," in *IEEE Design & Test of Computers*, 1993.

[43] X. Xiong, E. Barros, and W. Rosentiel, "A method for partitioning UNITY language in hardware and software," in *Proceedings of the European Design Automation Conference (EuroDAC)*, 1994.

[44] R. Walker and D. Thomas, "Behavioral transformation for algorithmic level IC design," *IEEE Transactions on Computer-Aided Design*, October 1989.

[45] J. Hagerman and D. Thomas, "Process transformation for system level synthesis." Technical Report CMUCAD-93-08, 1993.

[46] A. Nicolau and R. Potasman, "Incremental tree height reduction for high level synthesis," in *Proceedings of the Design Automation Conference*, pp. 770–774, 1991.

[47] M. Girkar and C. Polychronopoulos, "Automatic extraction of functional parallelism from ordinary programs," in *IEEE Transactions on Parallel and Distributed Systems*, pp. 166–178, 1992.

[48] E. D. Lazowska, *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*. Englewood Cliffs, New Jersey: Prentice-Hall, 1984.

[49] W. Ye, R. Ernst, T. Benner, and J. Henkel, "Fast timing analysis for hardware-software co-synthesis," in *Proceedings of the International Conference on Computer Design*, pp. 452–457, 1993.

[50] J. Gong, D. Gajski, and S. Narayan, "Software estimation from executable specifications," in *Journal of Computer and Software Engineering*, 1994.

[51] C. Park and A. Shaw, "Experiments with a program timing tool based on source-level timing scheme," *IEEE Computer*, vol. 24, pp. 48–57, May 1991.

[52] P. Puschner and C. Koza, "Calculating the maximum execution times of real-time programs," *Journal of Real-Time Systems*, vol. 1, pp. 159–176, 1989.

[53] A. Shaw, "Reasoning about time in higher-level language software," *IEEE Transactions on Software Engineering*, vol. 15, July 1989.

[54] W. Wolf, "Hardware-software co-design of embedded systems," *Proceedings of the IEEE*, vol. 82, no. 7, pp. 967–989, 1994.

[55] S. Narayan and D. Gajski, "Synthesis of system-level bus interfaces," in *Proceedings of the European Conference on Design Automation (EDAC)*, 1994.

[56] D. Filo, D. Ku, C. Coelho, and G. DeMicheli, "Interface optimization for concurrent systems under timing constraints," in *IEEE Transactions on Very Large Scale Integration Systems*, pp. 268–281, September 1993.

[57] G. Borriello, "Specification and synthesis of interface logic." In R. Camposano and W. Wolf, Editors, *High-Level VLSI Synthesis*, Kluwer Academic Publishers, Boston, 1991.

[58] P. Moeschler, H. Amann, and F. Pellandini, "High-level modeling using extended timing diagrams," in *Proceedings of the European Design Automation Conference (EuroDAC)*, 1993.

[59] G. Borriello and R. Katz, "Synthesis and optimization of interface transducer logic," in *Proceedings of the International Conference on Computer-Aided Design*, 1987.

[60] J. Akella and K. McMillan, "Synthesizing converters between finite state protocols," in *Proceedings of the International Conference on Computer Design*, 1991.

[61] J. Sun and R. Brodersen, "Design of system interface modules," in *Proceedings of the International Conference on Computer-Aided Design*, pp. 478–481, 1992.

[62] J. Sun, M. Srivastava, and R. Brodersen, "SIERA: A CAD environment for real-time systems," in *3rd Physical Design Workshop*, May 1991.

[63] K. Hwang and F. Briggs, *Computer Architecture and Parallel Processing*. McGraw-Hill, 1985.

[64] S. Levi and A. Agrawala, *Real-Time System Design*. McGraw-Hill, 1990.

[65] G. Andrews and F. Schneider, "Concepts and notations for concurrent programming," *ACM Computing Surveys*, vol. 15, pp. 3–44, March 1983.

[66] D. Gajski and L. Ramachandran, "Introduction to high-level synthesis," in *IEEE Design & Test of Computers*, 1994.

[67] D. Thomas, E. Dirkes, R. Walker, J. Rajan, J. Nestor, and R. Blackburn, "The system architect's workbench," in *Proceedings of the Design Automation Conference*, 1988.

[68] J. Van-Eijndhoven and L. Stok, "A data flow graph exchange standard," in *Proceedings of the European Conference on Design Automation (EDAC)*, pp. 193–199, 1992.

[69] V. Chaiyakul and D. Gajski, "High-level transformations for minimizing syntactic variances," in *Proceedings of the Design Automation Conference*, 1993.

[70] R. Camposano and W. Wolf, *High-Level VLSI Synthesis*. Boston, Massachusetts: Kluwer Academic Publishers, 1991.

[71] G. DeMicheli, *Synthesis and Optimization of Digital Circuits*. McGraw Hill, 1994.

[72] R. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang, "MIS: A multiple-level logic optimization system," *IEEE Transactions on Computer-Aided Design*, vol. 6, pp. 1062–1080, November 1987.

[73] S. Devadas, H. Ma, A. Newton, and A. Sangiovanni-Vincentelli, "MUSTANG: State assignment of finite state machines targeting multilevel logic implementations," *IEEE Transactions on Computer-Aided Design*, vol. 7, no. 12, pp. 1290–1299, 1988.

[74] K. ten Hagen and H. Meyr, "Timed and untimed hardware/software co-simulation: Application and efficient implementation," in *International Workshop on Hardware-Software Co-Design*, 1993.

[75] D. Becker, R. Singh, and S. Tell, "An engineering environment for hardware/software co-simulation," in *Proceedings of the Design Automation Conference*, pp. 129–134, 1992.

[76] R. Gupta, C. Coelho, and G. DeMicheli, "Synthesis and simulation of digital systems containing interacting hardware and software components," in *Proceedings of the Design Automation Conference*, pp. 225–230, 1992.

[77] S. Lee and J. Rabaey, "A hardware-software cosimulation environment," in *International Workshop on Hardware-Software Co-Design*, 1993.

[78] L. Ramachandran, D. Gajski, S. Narayan, F. Vahid, and P. Fung, "Towards achieving a 100-hour design cycle: A test case," in *Proceedings of the European Design Automation Conference (EuroDAC)*, 1994.