A System-Design Methodology: Executable-Specification Refinement †

Daniel D. Gajski, Frank Vahid and Sanjiv Narayan
Department of Information and Computer Science
University of California, Irvine, CA, 92717

Abstract

As methodologies and tools for chip-level design mature, design effort becomes focused on increasingly higher levels of abstraction. We present a methodology and tool for system-level specification, design and refinement that result in an executable specification for each system component. The specification for each component can then be synthesized into hardware or compiled to software. We highlight advantages of the proposed methodology compared to current practice.

1 Introduction

The focus of design effort on higher levels of abstraction has led to the need for a system-level methodology and supporting tools. There are two main steps in system-level design. The first is *functionality specification*, which is the task of describing the desired system behavior in some form. The second is *system design*, which is the task of implementing this functionality with system components such that design constraints are satisfied. Example system components include standard processors and microcontrollers, memories, buses, and custom ASICs. The domain of these two steps is shown in Figure 1. The result of system design is a set of system components, each with its own functional specification. Implementation of each component follows. A standard component requires software compilation of the functional specification into machine code, whereas custom components require synthesis of the specification into register-transfer structure. The first task is accomplished with standard compilers while the second one uses high-level and logic synthesis.

There are two very different system-level design approaches in current practice. In one approach, the system's functionality is first implemented with interconnected register-transfer or gate-level objects, and

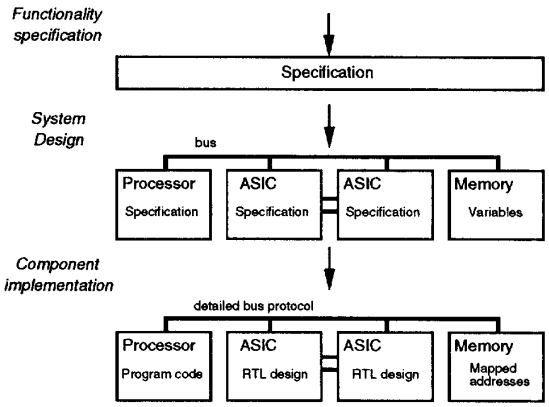

Figure 1: System-level domain

this structure is then partitioned among system components. However, once structure is obtained only minor changes can be made to the system's performance through introduction of redundant objects or through repartitioning. More substantial changes require knowledge of high-level functional and timing information, but such information can not be discerned from the structure. A second drawback to this approach is it doesn't consider software implementations.

In the other approach, the system's functionality is first partitioned among system components, and each component is then implemented as structure or as software, depending on the component type. While overcoming the drawbacks of the structure-first approach, current practice of this approach involves mostly informal and manual techniques. The functionality is informally specified using a natural language such as English, and system design is done manually using mental or hand-calculated estimations for quality metrics such as performance, size, and power. Drawbacks of such techniques include the lack of early functional verification, the lack of good feedback with regards to quality metrics that result from design decisions, the lack of automated tools to reduce design time, and

†This work was supported by the SRC (grant #91-DJ-146), NSF(grant #MIP 8922851-01) and California Micro(grant #91-040, #91-041).

the lack of good documentation of functionality and design decisions to aid in concurrent design and in redesign.

Several research efforts have focused on overcoming one or more of these drawbacks. Simulation environments have been developed to encourage early system simulation of hardware and software components for functional verification [1, 2]. An architectural template and tools environment for rapid prototyping have also been suggested [3]. Functional partitioning approaches have been introduced for multiple custom chips [4, 5], and for multiple processors [6]. Issues for functional partitioning among hardware and software components have been discussed [7], and prototype partitioning systems have been developed [5, 8]. Frameworks have been proposed to support the process of controlling and interfacing various system-design tools [9].

The methodology and tool we present can be used in conjunction with the simulation, prototyping, and framework environments described above. Our work differs from other previous efforts in several key points.

First, we handle exploration of various implementations of the *three* aspects of functionality, namely behavior, data, and communication, rather than focusing on behaviors alone as in most previous work.

Second, our technique is applicable to a variety of system-component technologies, not just a fixed set of hardware or software components of one technology. The above two points provide a seamless exploration of system-design options, which includes hardware/software codesign.

Third, the end result of our approach is a *refined specification* in which interconnected system-components are functionally specified, permitting further verification and encouraging concurrent design.

Fourth, we use a new model (PSM) for specification that can describe both hardware and software functionality, at varying levels of abstraction, in a uniform manner. This model differs from previous ones which were well suited for either software or hardware but not both.

In this paper, we present our system-level methodology. It is highlighted in Figure 2, where the boxed items represent a replacement of informal and manual techniques with well-defined and automatable ones. We first describe system specification and introduce our PSM model. We then define and describe the major system-design tasks, as well as a suggested ordering of those tasks. We provide a real example, and describe an environment to support the methodology.

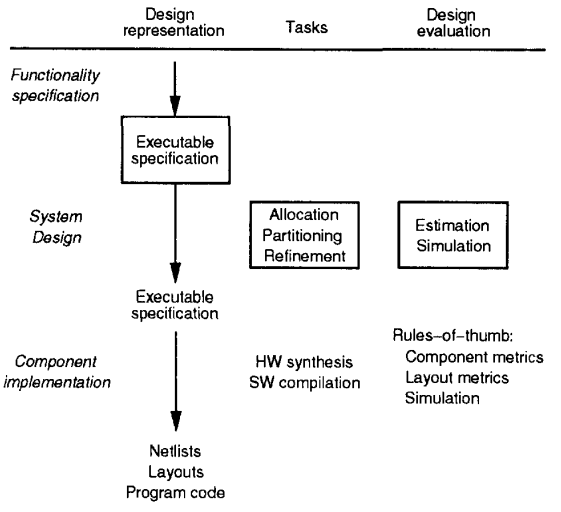

Figure 2: Proposed system-level methodology

2 Executable specification and PSM

We use an executable specification rather than a natural language to specify system functionality. An executable specification is one which captures the functionality of the system in a machine-readable and simulatable form. It has several advantages over a natural language. First, simulation enables early verification of the correctness of the system's intended functionality. Second, the specification may serve as an input to synthesis tools, resulting in significantly reduced design times. Third, the specification serves as documentation by providing a precise description of intended functionality.

Many languages have been used for executable specification, including VHDL, Verilog, C, and Statecharts. Such languages are used to capture common conceptual models such as finite-state machines (FSM), FSMs with complex expressions, hierarchical and concurrent FSMs, dataflow diagrams, Petri-nets, and communicating sequential processes. Unfortunately, these conceptual models are not adequate for concisely describing all of the characteristics of a common class of systems referred to as **embedded systems**. These characteristics include: programming constructs, state-transitions, sequential and concurrent behavior decompositions, and exceptions. To overcome this limitation, we developed a new model called program-state machines.

Program-state machines (PSM) [10] are essentially a combination of the hierarchical finite-state machine and programming language paradigms. A sys-

tem is specified as a hierarchy of *program-states*, where each program-state represents a mode of computation and may include standard programming declarations such as variables, types, and subroutines. At any given time only a subset of program-states are active, i.e. are actively carrying out their computations. A single root program-state represents the entire system and is always active.

A program-state may either be a **composite** program-state or a **leaf** program-state. A composite program-state may be hierarchically decomposed either into a set of **concurrent** program-substates (all program-substates are active when the program-state is active), or into a set of **sequential** program-substates (only one of the program-substates is active at a time when the program-state is active). A sequentially decomposed program-state contains a set of transition arcs to represent the sequencing between the program-substates. There are two types of transition arcs. A *transition-on-completion arc (TOC)* is traversed when the source program-substate has completed its computation and the associated arc condition evaluates to true. A *transition-immediately arc (TI)* is traversed immediately when the arc condition becomes true, regardless of whether or not the source program-substate has completed its computation. A leaf program-state is at the bottom of the behavioral hierarchy and has its computation described using **programming language** statements.

The PSM model supports all embedded-system characteristics in an elegant manner. In addition, as the system is refined, the programming constructs can be used to describe portions destined for software implementation while the state-transitions describe portions destined for hardware implementation, all with a single uniform representation that eliminates the need for multiple languages. Since no language currently exists that supports all the PSM characteristics, we developed a VHDL front-end language called SpecCharts [11].

Our subsequent system-design methodology is not strictly dependent on use of the PSM model and SpecCharts. Other languages may be used to capture the PSM model, with some extra effort. However, PSM and SpecCharts are in closest accord with our system-design methodology, and yield the most concise and readable specifications.

2.1 System design

System design is the task of mapping the functionality, as captured in an executable specification, to some set of system components such that design constraints on parameters such as monetary cost, performance, and power are satisfied. Our approach to system design consists of three well-defined tasks on three classes of functional objects, as summarized in Figure 3. The three classes of functional objects that comprise any executable specification are **variables**, **behaviors**, and **channels**. Variables store data, behaviors transform data, and channels transfer data between behaviors. In our terminology, a behavior is a non-trivial algorithmic-level computation that together with other behaviors describe all system actions (identical to the "task" concept described in [7]). It corresponds to a block of statements in the specification such as a loop body, procedure, or process. For each of these objects there are three tasks to be performed: allocation, partitioning, and refinement.

Functional objects	*System-design tasks*		
	Allocation	Partitioning	Refinement
Variables	Memories	Variables to memories	Address assignment
Behaviors	Processors	Behaviors to processors.	Interfacing
Channels	Buses	Channels to buses	Arbitration/protocols

Figure 3: System-design tasks

Allocation adds system components to the design. One class of system component consists of memories, such as RAMs, ROMs, register-files, and registers. Memories are used to store scalar and array variables. Another class of component consists of standard processors and microcontrollers as well as custom ASIC "processors". These standard/custom processors are used to implement behaviors. A third class of "component" consists of physical buses. Buses are used to implement communication channels.

Partitioning maps each class of functional objects to allocated components. Variables are mapped to memories, behaviors are mapped to standard/custom processors, and channels are mapped to buses. Each mapping is many to one. Standard partitioning algorithms, such as clustering or simulated annealing, can be applied. Clustering may use any of various closeness criteria [12]. For behaviors, common criteria include interconnection, communication, sequentiality, and hardware sharability. For variables and for channels, common criteria include sequential access, common accessors, and width similarity.

Refinement adds new behaviors to maintain correct functionality for a given allocation and partitioning. Variables partitioned among memories require memory address translation. Behaviors separated among components must be modified to maintain correct communication. Channels mapped to buses require interface synthesis to determine communication protocols, and arbiter synthesis to resolve any simultaneous bus requests. A refined specification is then generated consisting of a set of interconnected system-components, each functionally specified.

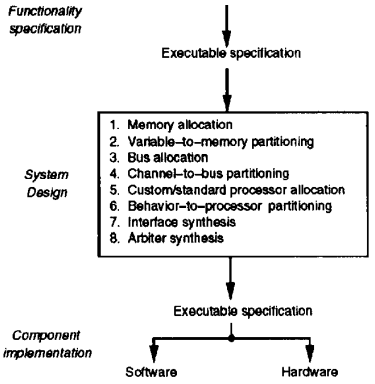

Figure 4: One possible ordering of tasks

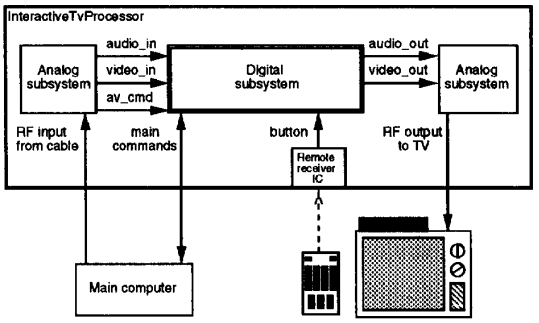

Figure 5: The ITVP's environment

There is no fixed ordering of the system-design tasks. One ordering which we have found leads to good results is shown in Figure 4. After the functionality is specified, large variables are mapped to memories such that variables which satisfy closeness criteria are mapped to the same memory. Channels are mapped to buses in a similar manner. Then processor or ASIC components are allocated and behaviors are partitioned among those components. Variable or channel repartitioning may follow in order to further improve the design. Interface and arbiter synthesis are performed to complete the functional specification of each component.

Throughout the entire process, allocation and partitioning tasks are guided by **estimations** of design quality metrics such as area, performance, pins, and power. Accurate yet fast estimation techniques are a subject of intense research; some results for both hardware and software implementations are described in [13, 14, 15, 16]. The estimations are incorporated into an objective function used for design evaluation. A common objective function is one which favors designs with the smallest amount of constraint violations.

To further evaluate design decisions, a refined specification can be generated and simulated between any tasks during the design process.

The resulting system components can be implemented with either automated or manual techniques. Since the components are defined as an executable specification, synthesis or compilation follow very naturally.

3 Example

Let us consider as a design example an interactive television processor (ITVP). The system captures video frames and displays them as still pictures while audio is played. A user can interact by selecting menu items via the TV remote control, resulting in an appropriate new video frame and accompanying audio. Such systems are common in hotels, stores, and more recently in homes with cable TV. They can be used to take a video tour of a hotel, shop through a video catalog, or even to perform banking transactions or make airline reservations. The system resides in a box adjacent to a television set, similar to a box for cable TV. A diagram of the overall system is found in Figure 5, with only certain data flows shown. The core of the ITVP is a digital subsystem, the design of which will be our focus.

The main behaviors and data objects for the digital subsystem are shown in Figure 6. The actual system consists of 32 behaviors and 69 data objects derived from 860 lines of VHDL code, but only the large or important objects are shown. The system contains behaviors called *CaptureAudio* and *GenerateAudio* to capture and then generate several thousand successive audio bytes (i.e. a frame), using two arrays *audio1* and *audio2*. Likewise, the *CaptureGenerateVideo* behavior and *video* array capture and generate video frames. A *fonts* array indicates which of the 16x16 pixels should be illuminated for each of the 128 supported ASCII characters. A *screen_chars* array indicates which ASCII character, if any, should be displayed in each of the 30x30 screen positions. *OverlayCharacters* reads the screen-characters and fonts arrays and indicates to the video generator when to override a video pixel with a white pixel so that a white character will appear on the screen. Behavior *CaptureAVCmd* and variable *av_cmd* capture an encoded command that appears at the start of every audio or video frame, where the command indicates how to handle the frame. Finally, there is a behavior *ProcessRemoteButtons* to respond to buttons pressed on the remote control, a behavior *ProcessAVCmd* to handle the encoded audio/video command, and a behavior *ProcessMainCmds* to respond to commands issued by the main computer.

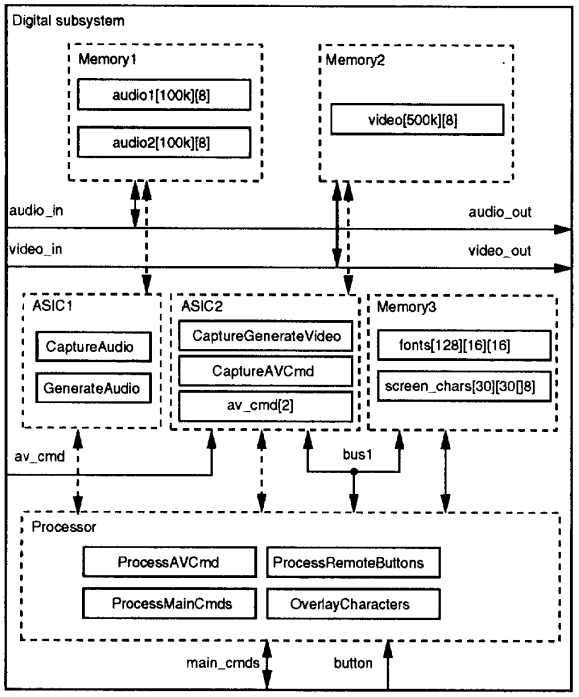

Figure 6: One possible design for the ITVP

After the specification and system-design steps of our methodology, the system components are defined and the system may be summarized in "block-diagram" form, a form familiar to most designers. Such a block-diagram is shown in Figure 6 for our example. The system is implemented with six components: three memories, two ASICs, and a processor. Each of the behaviors and variables is mapped to exactly one of these components. The *Memory*1 component stores both the *audio*1 and *audio*2 arrays, while *Memory*2 stores the *video* array. *Memory*3 stores both the *fonts* array and the *screen_chars* array. *ASIC*1 implements the *CaptureAudio* and *GenerateAudio* behaviors, *ASIC*2 implements *CaptureGenerateVideo* behavior as well as the *CaptureAVCmd* behavior and the *av_cmd* variable, and *Processor* implements the *ProcessAVCmd*, *ProcessMainCmds*, *ProcessRemoteButtons* and *OverlayCharacters* behaviors. Note that there is a single bus, *bus*1, to which several channels have been mapped, including behavior accesses to *av_cmd*, *fonts* and *screen_chars*.

Let us consider the decisions made in the example design. The audio and video arrays were stored in separate memories because they must be output simultaneously. Storing them in the same memory would have required multiplexing accesses to them which in turn would have violated minimum audio/video output-rate constraints. The audio and video capture and generate behaviors were implemented on ASICs because software implementations would not have met input and output rate constraints. The audio and video behaviors appear on different ASICs because they would not both fit on a single ASIC containing 20,000 gates. An ASIC was used to store the *av_cmd* variable because the command must be captured immediately, and such immediate capture would have been difficult to implement on the processor. The variable could have been stored on either ASIC. The *fonts* and *screen_chars* arrays are not accessed concurrently so they were mapped to a single memory without loss of performance. The behaviors in the processor do not require very fast performance, so they can be implemented in software without violating constraints, even though the behaviors are specified as executing concurrently but are implemented sequentially (actually they are interleaved with one another).

The above block diagram represents just one of many possible ITVP designs. For example, the two ASIC components can be replaced by a single, larger ASIC. Alternatively, we can use an ASIC technology with different cost and performance characteristics. We can even use a microcontroller rather than a processor to lower costs. For each possible set of components, there are numerous alternative mappings of behaviors and variables to those components.

To generate a specification for each component, the processor behaviors must be refined into a single sequential behavior, and interfaces must be synthesized between each behavior and variable which require data transfer between components. The example demonstrates the need and usefulness of the allocation, partitioning, and refinement tasks as defined in our methodology.

4 The SpecSyn environment

A set of tools to support our methodology is being implemented as the **SpecSyn** system-design environment. It consists of partitioners [17] which support several algorithms including clustering, group migration, and simulated annealing; estimators [13, 14] for the quality metrics of execution-time in hardware or software, hardware area, software instruction and data memory size, and pins; and prototype tools to synthesize arbiters and interfaces [18]. The partitioners, estimators, and interface/arbiter tools are implemented in C with 16000, 19000, and 8000 lines of code, respectively. Routines for internal representation of the specification require 30,000 lines of code. SpecSyn has been released to over 10 companies and a new version is due for distribution in the first quarter of 1994.

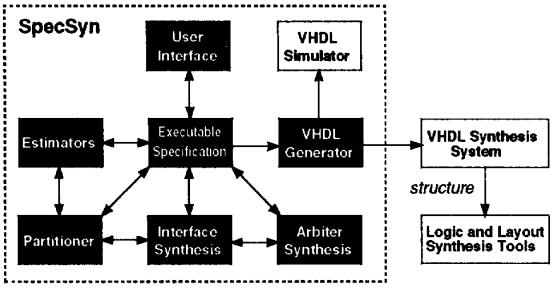

Figure 7: The SpecSyn system-design environment

5 Conclusion

We have applied the methodology with the SpecSyn environment on several examples, including a medical instrument for measuring bladder volume, a fuzzy-logic controller, a RISC signal processor, an interactive TV system, a microwave-transmitter controller, and an answering machine. Design quality is comparable with manual designs and design-time is up to an order of magnitude less. Numerous manual allocations and partitionings and hundreds of automatically generated ones can be evaluated in just minutes.

There are several advantages to using a system-level methodology that refines an executable specification through the system-design tasks of allocation, partitioning, and refinement. First, early functional verification through simulation is possible, which minimizes the occurrence of time-consuming functional changes later in the design process. Second, the resulting system-component executable specifications are precise so they enable concurrent design and minimize integration problems. Third, the specification is machine-readable so automation tools can be used to reduce overall design time and estimators used to rapidly explore many more possible designs. Fourth, the well-defined system tasks combined with the executable specification lend themselves to excellent documentation of system-design decisions and intended functionality, which is especially important for redesign.

We envision a conceptualization environment that allows designers to quickly explore and evaluate potential designs. This requires work on three parts of such an environment: (1) a component base with VHDL models for different technologies, (2) fast estimators of quality metrics for different architectural styles and technologies (e.g. custom layout, gate-arrays, FPGA's, and standard components), and (3) an appropriate human interface for display of quality metrics and different user views. The SpecSyn environment is a first step in this direction.

References

[1] A. Kalavade and E. Lee, "A Hardware/Software Codesign Methodology for DSP Applications," in *IEEE Design & Test*, 1993.

[2] R. Gupta, C. Coelho, and G. DeMicheli, "Synthesis and Simulation of Digital Systems Containing Interacting Hardware and Software Components," in *DAC*, pp. 225–230, 1992.

[3] M. Srivastava and R. Brodersen, "Rapid-Prototyping of Hardware and Software in a Unified Framework," in *ICCAD*, pp. 152–155, 1992.

[4] E. Lagnese and D. Thomas, "Architectural Partitioning for System Level Synthesis of Integrated Circuits," *IEEE Trans. on CAD*, July 1991.

[5] R. Gupta and G. DeMicheli, "Hardware-Software Cosynthesis for Digital Systems," in *IEEE Design & Test*, pp. 29–41, October 1993.

[6] S.Prakash and A. Parker, "Synthesis of Application-Specific Multiprocessor Architectures," in *DAC*, pp. 8–13, 1991.

[7] D. Thomas, J. Adams, and H. Schmit, "A Model and Methodology for Hardware/Software Codesign," in *IEEE Design & Test*, pp. 6–15, 1993.

[8] R. Ernst, J. Henkel, and T. Benner, "Hardware-Software Cosynthesis for Microcontrollers," in *IEEE Design & Test*, pp. 64–75, December 1994.

[9] M. Jacome and S. Director, "Design Process Management for CAD Frameworks," in *DAC*, pp. 500–505, 1992.

[10] D. Gajski, F. Vahid, and S. Narayan, "SpecCharts: A VHDL Front-End for Embedded Systems." UC Irvine TR 93-31,1993.

[11] S. Narayan, F. Vahid, and D. Gajski, "System Specification with the SpecCharts Language," in *IEEE Design & Test*, Dec. 1992.

[12] D. Gajski, J. Gong, F. Vahid, and S. Narayan, "The SpecSyn Design Process and Human Interface." UC Irvine TR 93-3,1993.

[13] S. Narayan and D. Gajski, "Area and Performance Estimation from System-Level Specifications." UC Irvine TR 92-16,1992.

[14] J. Gong, D. Gajski, and S. Narayan, "Software Estimation from Executable Specifications," in *Journal of Computer and Software Eng.*, to appear.

[15] W. Ye, R. Ernst, T. Benner, and J. Henkel, "Fast Timing Analysis for Hardware-Software Co-Synthesis," in *ICCD*, pp. 452–457, 1993.

[16] F. Vahid, S. Narayan, and D. Gajski, "Constant-Time Cost Evaluation for Behavioral Partitioning." UC Irvine TR 92-29,1992.

[17] F. Vahid and D. Gajski, "Specification Partitioning for System Design," in *DAC*, 1992.

[18] S. Narayan and D. Gajski, "Synthesis of System-Level Bus Interfaces," in *EDAC*, 1994.