# System-Level Abstraction Semantics

Andreas Gerstlauer
gerstlauer@cecs.uci.edu

Daniel D. Gajski
gajski@cecs.uci.edu

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-3425
http://www.cecs.uci.edu

## ABSTRACT

Raising the level of abstraction is widely seen as the solution for closing the productivity gap in system design. They key for the success of this approach, however, are well-defined abstraction levels and models. In this paper, we present such system level semantics to cover the system design process. We define properties and features of each model. Formalization of the flow enables design automation for synthesis and verification to achieve the required productivity gains. Through customization, the semantics allow creation of specific design methodologies. We applied the concepts to system languages SystemC and SpecC. Using the example of a JPEG encoder, we will demonstrate the feasibility and effectiveness of the approach.

## Categories and Subject Descriptors

I.6.4 [**Simulation and Modeling**]: Model Validation and Analysis; B.m [**Hardware**]: Miscellaneous—*Design management*

## General Terms

Design, Theory

## Keywords

System-level design, modeling, design semantics, abstraction levels, methodology

## 1. INTRODUCTION

It is a well-known fact that designers of SOCs are facing an increasing productivity gap between semiconductor technology and methodology and tool support. A lot of efforts have been focussed on raising the level of abstraction in the design process. With higher levels of abstraction, the number of objects in the design decreases exponentially. This allows the designer and tools to focus on the critical aspects and explore a larger part of the design space without being overwhelmed by unnecessary details. Tools will then help the designer in gradually refining the design to lower and lower levels.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
*ISSS'02,* October 2–4, 2002, Kyoto, Japan.
Copyright 2002 ACM 1-58113-576-9/02/0010 ...$5.00.

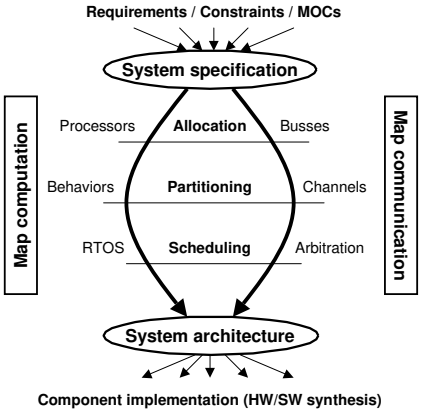



**Figure 1: System design tasks.**

A requirement for any design flow is a set of well-defined abstraction levels. The number of levels and the properties of each level have to be defined such that designers and tools can optimize decisions and move between levels efficiently. The aim is to decrease the number of objects to deal with at higher levels while providing enough detail to direct exploration at each step, trading off accuracy and efficiency, e.g. in terms of simulation speed. Furthermore, a clear and unambiguous definition of these levels is then needed to enable design automation for synthesis and verification. In addition, such a formalized definition is a necessity for interoperability across tools and designers.

The rest of this paper is organized as follows: after an introduction to the system design process and an overview of traditional modeling approaches, we will define the abstraction levels and models for system design in Section 2. In Section 3, we outline application of these definitions to different system-level languages. We present a specific design example and experimental results. The paper concludes with a summary and a brief outlook on future work in Section 4.

### 1.1 System Design Process

System design starts with a set of requirements where different parts are possibly captured in different ways. How-

| Level | Computation | Communication | Structure | Order | Validate |
|---|---|---|---|---|---|
| **Requirements** | Concepts | Tokens | Attributes | Constraints | Properties |
| **Specification** | Behaviors | Messages | Behavioral | Causality | Functionality |
| **Multiprocessing** | Processes | Messages | Processors | Execution delays | Performance |
| **Architecture** | Processes | Busses/Ports | Bus-functional | Timing-accurate | Protocols |
| **Implementation** | FSMDs | Signals | Microarchitecture | Cycle-accurate | Clock cycle |

**Table 1: System design models.**

ever, in order to feed a global design and synthesis flow, requirements have to be combined into a single, unambiguous system specification. As shown in Figure 1, the actual design process then consists of two analogous flows: (a) mapping of the computational parts of the specification onto processing elements (PEs) of a system architecture and (b) mapping of the communication in the specification onto system busses. Each flow requires allocation of components (PEs or busses), partitioning of the specification onto components, and scheduling of execution on the inherently sequential components. The result is the system architecture of PEs connected via busses. From there on, each of the PEs is then further implemented through software and hardware synthesis.

## 1.2 Traditional Models

There are several approaches dealing with classification and structuring of the design process [3, 8, 9]. However, none of these defines an actual flow with models at specific points.

Traditionally, abstracted models of a system design are used mainly for simulation purposes. In such simulation-centric approaches, the designer is responsible for manually rewriting the model at a fixed level of abstraction to adjust to changes in the design. There has been a lot of work done on horizontal integration of different models for simulation. At lower levels, different languages or implementations are integrated for co-simulation [2, 4]. At higher levels, different models of computation are combined into common simulation environments for specification [1]. However, none of these approaches attack the vertical integration of models that is needed for a synthesis-centric design flow with refinement of higher-level models into lower-level ones.

Recently, some research has focussed on abstracting communication for the purpose of specification and possibly automatic generation of communication implementations from such higher-level specifications [16, 10, 12, 15]. Although they are the motivation for our intermediate processor model, these approaches focus on abstracting communication and don't provide as high abstractions for the computational aspects. For example, in all cases the system is described as a netlist of concurrent processes, and computational units of hierarchy can only be composed in a parallel fashion, i.e. all blocks are active all the time and it is cumbersome to describe a sequential composition of computation.

# 2. ABSTRACTION LEVELS

A general classification of the design process is available through the Y-Chart [3]. It defines system, register-transfer (RT), gate, and transistor levels where each level is defined by the type of objects and where higher level objects are hierarchically composed out of lower level ones. At each level,

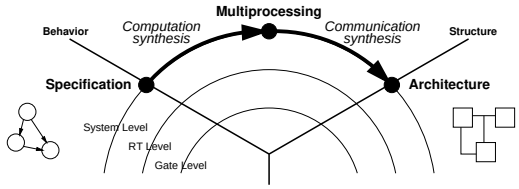



**Figure 2: System design flow.**

the design can be described in the form of a behavioral or a structural model. A behavioral model describes the desired functionality as a composition of abstract functional entities. Behavioral objects are pieces of functionality that get activated, process input data, produce output data, and terminate. In a behavioral description, those pieces are then arranged to model data and control dependencies between them. A structural model, on the other hand, describes the netlist of physical components and their connectivity. Structural objects represent real, non-terminating components and wires that are actively processing data at all times.

Models are points in the Y-Chart. A model is defined by the amount of implementation detail in the description of the design at that point. Together with the amount of structure as defined by the Y-Chart, a model determines the amount of order in the system. A behavioral description is partially ordered based on causality, i.e. dependencies only. In contrast, in a structural description order is increased by creating a total order in time on the physical objects.

In the Y-Chart, design is the process of moving from a behavioral description to a structural description under a set of constraints where the structural objects are each designed at the next lower level. This process is also called *synthesis*, especially when automated. At the system level, design is therefore the process of deriving a structural description of the system, the system architecture, from a behavioral system description, the system specification. Behavioral objects at the system level are general functions and algorithms that communicate by transferring data through global variables. Structural objects are processing elements (PEs), e.g. general purpose processors, custom hardware, IPs, and memories that communicate via busses.

In general, the system design process is too complex to be completed in one single step. The gap between requirements and implementation is too large for non-exponential algorithms. Hence, we need to divide the process into a sequence of smaller, manageable steps. As explained in the introduction, computation and communication refinement are largely orthogonal. Therefore, it is beneficial to subdivide the design process into the two separate tasks of computation and communication design. However, although interactions between tasks are minimized, there are still strong

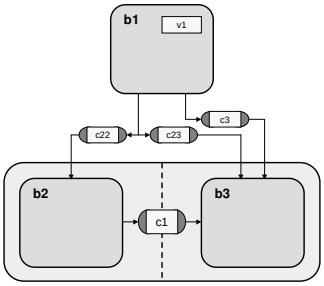



**Figure 3: Specification model example.**

dependencies. Especially, since partitioning of computation influences the amount of communication to be performed, computation synthesis needs to be performed before communication synthesis.

Figure 2 shows the resulting flow and models. System design starts with the behavioral specification model. In a first step, computation is implemented on PEs, resulting in the intermediate multiprocessing model. The multiprocessing model is a mixed behavioral/structural description. It defines the computation structure but leaves communication at a behavioral level. Finally, communication synthesis completes the design flow and creates the structural system architecture model.

In the following sections we will define those three models. Table 1 summarizes the characteristics of the different models for system design. Due to space constraints, definitions are limited in detail. For more information, please refer to [6].

## 2.1 Specification Model

The specification is a behavioral description of the system. It describes the desired functionality free of any implementation details. The specification is composed without any implications about the structure of the implementation. Objects in the specification model are abstract entities that perform computation on data and terminate. Apart from timing constraints, there is no notion of time, i.e. behavioral objects execute in zero time. Objects are ordered only based on their dependencies. An example of a simple yet typical specification model is shown in Figure 3.

At the specification level, a design consists of computation and communication. Computation is described by a hierarchical composition of behaviors. Behaviors communicate by transferring data messages over channels. More formally, a specification model is a triple

$$\langle B, C, R \rangle$$

consisting of a set of behaviors $B$, a set of channels $C$, and a connectivity relation $R \subseteq B \times C$ that defines connections of behaviors to channels.

Behaviors form a semigroup $(B, \circ)$ under the composition operation $\circ \in \{\triangleright, \|, |, \vee\}$. Behaviors $b1, b2 \in B$ can be composed sequentially ($b1 \triangleright b2$), concurrently ($b1 \| b2$), in a pipelined loop ($c : b1|b2$), or in a mutually exclusive way ($c : b1 \vee b2$) where the pipelined and alternative compositions are guarded by additional conditions $c$. Blocks at the leaves of the hierarchy contain basic algorithms that perform computations. Such leaf behaviors contain a description of the algorithm using, for example, a standard programming language like C. Hence, the code in the leaves describes how the behavior processes its input data to produce its output data using expressions over variables with different data types as supported by the programming language. Throughout the system design process, leaf behaviors will remain untouched, forming indivisible units for the purpose of exploration and refinement. In general, models describe how the system is composed out of the basic building blocks—the leaf behaviors—on top of any underlying language.

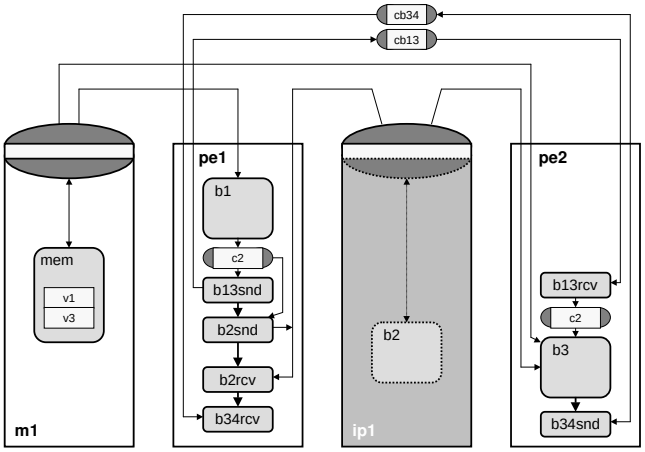



**Figure 4: Multiprocessing model example.**

In summary, the purpose of the specification model is to clearly and unambiguously describe the system functionality. The system is composed of self-contained blocks with well-defined interfaces enabling easy composition, rearrangement, and reuse. All dependencies are explicitly captured through the connectivity between behaviors and no hidden side effects exist. The parallelism available between independent blocks is exposed through their concurrent or pipelined composition. Computation and communication are abstracted as a composition of functions over data. They are separated into behaviors and channels, respectively, allowing for a separate implementation of both concepts.

## 2.2 Multiprocessing Model

The multiprocessing model is the result of mapping computation onto actual processing elements (PEs). It represents the allocation and selection of PEs and the mapping of behaviors onto PEs. It is a mix of a structural description of system computation and a behavioral description of system communication. An exemplary multiprocessing model corresponding to the specification example from Figure 3 is shown in Figure 4.

The multiprocessing model redefines the computational part of the design. Formally, a multiprocessing model is a triple

$$\langle PE, C, R \rangle$$

where computation is described as a set of concurrent PEs. PEs are structural objects representing physical components and as such are non-terminating. In general, the set of PEs in the system, $PE = P \cup IP \cup M$, consists of a set of general-purpose processors, a set of IPs, and a set of memories, respectively. Communication as the set of channels $C$ and the connectivity relation $R$ between leaf behaviors and channels remains essentially untouched.

A processor $p \in P$ is defined as a triple $\langle B_p, C_p, R_p \rangle$ that executes the set of behaviors $B_p$ mapped onto it. Behaviors inside processors communicate via a set of local channels $C_p$ as defined by the connectivity relation $R_p \subseteq B_p \times C_p$. Due to the inherently sequential nature of processing elements, behaviors inside a processor have to be serialized. In a static scheduling approach, the order of behaviors is fixed and represented as artificial control dependencies of a purely sequential composition of behaviors inside the PE, i.e. processor behaviors form a semigroup $(B_p, \triangleright)$ under sequential composition only. In a dynamic scheduling approach (not shown in this paper), the order of behaviors is determined at run-time. Behaviors are composed into tasks and an abstraction of the operating system scheduler in the multiprocessing model dispatches tasks dynamically.

In contrast to general purpose PEs, an $ip \in IP$ is defined as a pair $\langle B_{ip}, W_{ip} \rangle$ where the pre-defined, fixed functionality $B_{ip}$ is encapsulated in a wrapper $W_{ip}$. The wrapper abstracts the IP's internal communication interface and provides a set of canonical channel interfaces for communication with the IP at the behavioral level. At the system level, behaviors then can communicate directly with those wrappers, i.e. the system connectivity relation $R \subseteq B \times (C \cup W)$ connects processor behaviors $B = \bigcup_{p \in P} B_p$ to channels $C$ or IP wrappers $W = \bigcup_{ip \in IP} W_{ip}$. Note that dedicated memories are a special case of IPs which do not provide any functionality apart from reading and writing of data.

In summary, the multiprocessing model refines computation by grouping behaviors and mapping them onto a PE structure while largely preserving the original behavioral communication. PEs contain a behavioral description of their functionality. Behaviors inside PEs execute in order through static or dynamic scheduling. In addition, the multiprocessing model introduces the notion of time for the computation mapped onto the PEs, further increasing the partial order among PEs. Based on estimated execution times on the target PEs, behaviors are annotated with delay information. Therefore, true parallelism at the multiprocessing level is only available through the set of concurrent PEs.

## 2.3 Architecture Model

The architecture model is a structural description of the complete system for both computation and communication. In addition to allocation and selection of PEs, the architecture model represents the allocation and selection of busses and the mapping of global channels onto busses. As a result, the system is modeled as a netlist of PEs connected via busses. It is obtained by adding bus protocols to all channels, splitting channels, and inlining them into each PE as bus drivers. Figure 5 shows the architecture model example corresponding to the previously shown multiprocessing model.

Based on the multiprocessing model definition, the architecture model redefines the global communication part of the system. An architecture model is defined as a triple

$$\langle PE, B, c \rangle$$

where $PE$ is the set of PEs, $B$ is the set of busses, and $c : \bigcup_{p \in PE} O_p \mapsto B$ is the port mapping function connecting PE ports to busses. In general, the set of architecture model PEs, $PE = P \cup IP \cup M \cup T \cup A$, is a combination of the sets of general-purpose processors, IPs, memories, transducers, and arbiters, respectively.

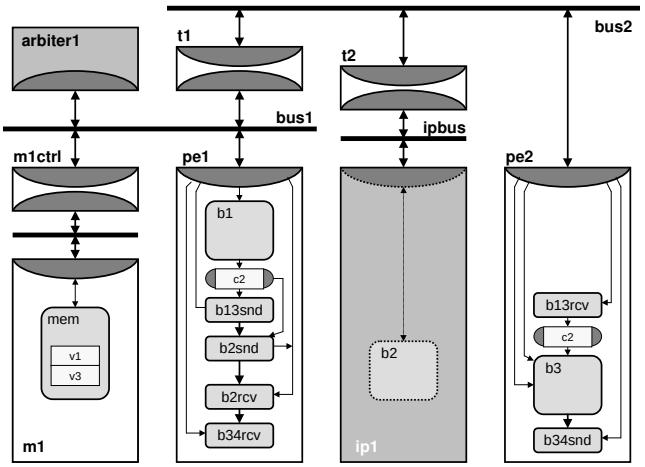



**Figure 5: Architecture model example.**

Behavioral processor descriptions are transformed to bus-functional models by adding bus drivers. A processor $p \in P$ in the architecture model is a quintuple $\langle B_p, C_p, D_p, O_p, R_p \rangle$ where $B_p$ is the scheduled set of behaviors executing on the processor, $C_p$ is the set of local channels, $D_p$ is the set of bus driver channel interfaces, $O_p$ is the processor's set of ports, and $R_p \subseteq B_p \times (C_p \cup D_p)$ is the connectivity relation that has been extended to define the connection of behaviors to channels and bus drivers. Bus drivers describe a processor's implementation of the data messages over the bus protocols on the processor's ports. Inside the processor, bus drivers provide a behavioral message interface to its behaviors and the behaviors connect to those channel interfaces for all bus communication.

For IP components, bus-functional or structural IP models can be directly integrated into the architecture model. Bus-functional IP models are equivalent to the definition of bus-functional processor models shown above. Structural IP models, on the other hand, are defined as netlists of RTL components. A structural $ip \in IP$ is a quadruple $\langle U_{ip}, B_{ip}, O_{ip}, c_{ip} \rangle$ where $U_{ip}$ is the set of RTL units, $B_{ip}$ is the set of local busses, $O_{ip}$ is the set of ports, and $c_{ip}$ is the connectivity function mapping ports of RTL units to busses and external IP ports. In the architecture model, bus-functional and structural IP models can be used interchangeably allowing, for example, mixed-level co-simulation. Again, note that memory components can be treated as a special case of IPs.

If necessary, special transducer PEs that translate between incompatible protocols need to be inserted into the architecture model. A transducer interfaces to two busses via two sets of ports and contains bus drivers for each protocol. Hence, a transducer is defined as a processor with two sets of ports and two sets of bus driver channel interfaces.

Finally, the architecture model can contain arbiters which mediate conflicting bus accesses in case of multiple masters on a bus. Arbiters implement a certain arbitration protocol on their bus ports through internal bus drivers. Therefore, equivalent to scheduling of computation on PEs in the multiprocessing model, arbiters serialize accesses to the inherently sequential busses. Arbiters usually come in the form of IPs and as such can be defined as bus-functional or structural processor models.

In summary, the architecture model refines communication into an implementation over busses, ports, drivers, and transducers. Computation inside the PEs, on the other hand, remains largely untouched. The structural nature imposes a total order on the communication over each bus. Furthermore, the partial order between busses is refined by introducing bus protocol timing. Therefore, the architecture model is timing-accurate in terms of both computation and communication.

# 3. EXPERIMENTS

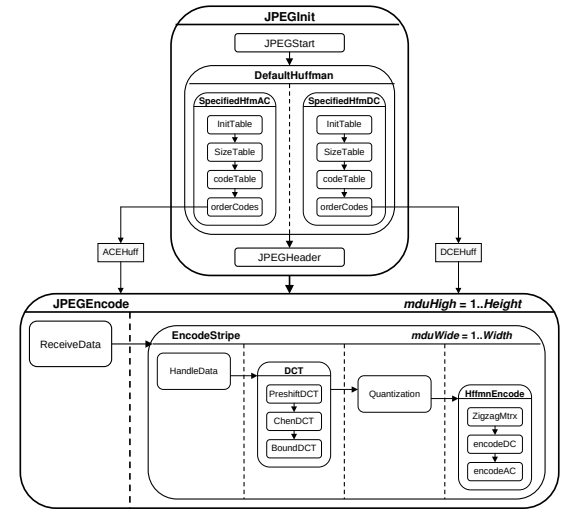
We have applied the system-level abstraction semantics to several system-level design languages including SystemC [7, 11] and SpecC [5, 14]. In order to represent the different models in a language, each model concept was translated into one or more language constructs. For example, specification behaviors map to processes in SystemC or behaviors in SpecC. Ideally, the mapping of model concepts to language constructs should be unambiguous in order to ease understanding of models written in a language for both humans or tools. Details of this application to different languages are, however, beyond the scope of this paper.

We then modeled several design examples following the presented flow. In the following, we will outline the implementation of a JPEG encoder. Note that the focus was on demonstrating feasibility and effectiveness of the models. Therefore, implementation decisions were made without performing elaborate design space exploration. Source code for all models in SpecC can be downloaded from our web pages [13]. In this case, we chose SpecC as modeling language since, at the time of development, SpecC supported the most concepts explicitly through dedicated constructs.

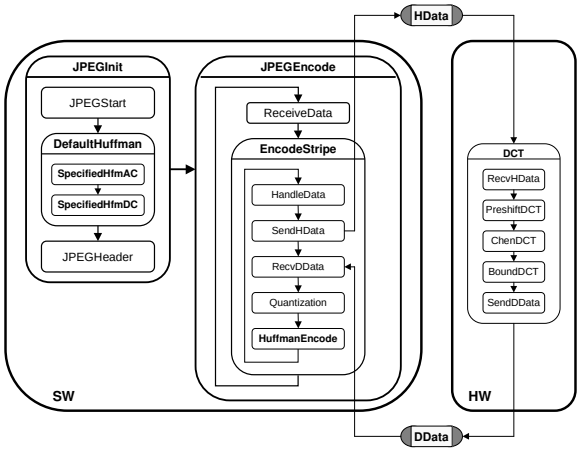
Figure 6 shows the three models of the JPEG encoder. At the top of the specification model (Figure 6(a)), the encoder consists of two sequential behaviors, *JPEGInit* followed by *JPEGEncode*. *JPEGInit* performs initialization of the two Huffman tables in two parallel subbehaviors, and writes the output header. Then, the actual encoding is done in two nested, pipelined loops. The outer pipeline splits the image into stripes of 8 lines each. The inner pipeline then splits the stripes into $8 \times 8$ blocks and processes each block through DCT, quantization and Huffman encoding. As an example of communication, Figure 6(a) shows the two Huffman tables *ACEHuff* and *DCEHuff* that are sent from *JPEGInit* to *JPEGEncode*. Note that since the two behaviors are composed sequentially, channels can degenerate to simple variables.

For the purpose of computation synthesis, we assumed a mapping of the encoder on a Motorola Coldfire processor ($SW$) assisted by a custom hardware co-processor ($HW$) for acceleration of the DCT (Figure 6(b)). Software and hardware communicate via two message-passing channels, sending and receiving $8 \times 8$ blocks from software to the DCT processor and back. Behaviors inside the $SW$ processor are statically scheduled and serialized. The two nested pipelines are converted into two nested, sequential loops.

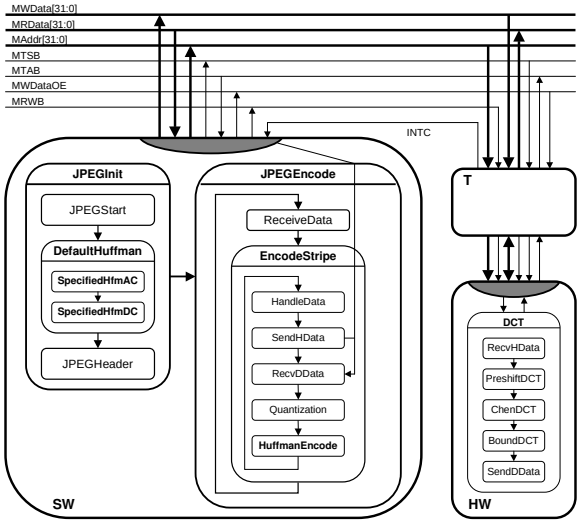
Finally, for communication synthesis, we connected the two processors via a single bus using the Coldfire bus protocol. Furthermore, it was assumed that the protocol of the DCT IP is fixed and incompatible with the Coldfire protocol, necessating the inclusion of a transducer (Figure 6(c)). The $SW$ processor is the master on the bus and drives the address and control lines. The transducer $T$ listens on the




(a) Specification




(b) Multiprocessing




(c) Architecture

**Figure 6: JPEG encoder.**

| | Lines of Code | Lines Changed | Simul. Time |
|---|---|---|---|
| Specification | 1,811 | | 3.8 s |
| Multiprocessing | 2,000 (+10%) | 235 (13%) | 4 s |
| Architecture | 2,493 (+25%) | 545 (27%) | 48 s |

**Table 2: JPEG encoder statistics.**

bus and translates between bus and *HW* co-processor protocol. For synchronization, the hardware signals the software through the processor's interrupt line *INTC*. Inside the PEs, bus drivers and interrupt handlers translate the message-passing calls of the behaviors into bus transactions by driving and sampling the PE's bus ports according to the protocol.

Characteristics of the JPEG encoder models in SpecC are listed in Table 2. The table shows both the lines of code and the number of lines added or changed when moving from one model to the next. As can be seen, refinement between levels is localized and leaves most of the original code untouched. Most of the changes result from additions to represent increased implementation detail.

To validate the models, we performed simulations at all levels. The simulation performance at different levels for the JPEG encoder (Table 2) and two additional examples, a JBIG encoder for facsimile applications and a voice encoder/decoder for mobile telephony, are shown in Figure 7. As we move down in the level of abstraction, more timing information is added, increasing the accuracy of the simulation results. However, simulation time increases exponentially with lower levels of abstraction. As the results show, moving to higher levels of abstraction enables more rapid design space exploration. Through the intermediate multiprocessing level, valuable feedback about critical computation synthesis aspects can be obtained early and quickly.

## 4. SUMMARY & CONCLUSIONS

In this paper, we presented a division of the system-level design process into three well-defined system-level models. The three models define a comprehensive approach at raising the level of abstraction in embedded systems design, supporting both computation and communication abstraction. The definition of models is based on a separation of concerns that minimizes interactions between levels, reduces refinement between models, and supports easy exploration with a variety of components and IPs. The two-step approach to the design flow supports rapid design space exploration by focusing on critical decisions at early stages while providing quick feedback.

To our knowledge, this is the first attempt at properly defining models in a formalized way. The models define a framework on top of which system-level languages and design methodologies can be developed. For example, platform based design predefines the sets of PEs and busses within the framework of multiprocessing and architecture models. The formalization of levels is the enabler for interoperability and design automation. Based on the above definitions, we can demonstrate automatic model refinement between levels. In the future, we want to extend the formalization to a general algebra with axioms based on which proofably correct transformations can be defined. Such a formalized framework of models and transformations based on the definitions presented in this paper is the foundation for the vertical integration of models through synthesis and verification.

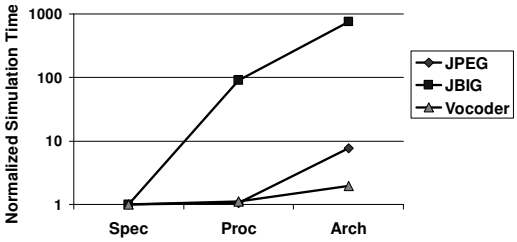



**Figure 7: Simulation performance.**

## 5. ACKNOWLEDGMENTS

This work was supported by the Semiconductor Research Cooperation (SRC), Task ID 832.002.

## 6. REFERENCES

[1] J. Buck et al. Ptolemy: A framework for simulating and prototyping heterogeneous systems. *Journal of Computer Simulation*, 4, 1994.

[2] P. Coste et al. Multilanguage design of heterogeneous systems. In *CODES*, 1999.

[3] D. D. Gajski and R. Kuhn. Guest editors introduction: New VLSI tools. *IEEE Computer*, pages 11–14, 1983.

[4] P. Gerin et al. Scalable and flexible cosimulation of SoC designs with heterogeneous multi-processor target architectures. In *ASPDAC*, 2001.

[5] A. Gerstlauer et al. *System Design: A Practical Guide with SpecC*. Kluwer Academic Publishers, 2001.

[6] A. Gerstlauer and D. D. Gajski. System-level abstraction semantics. Technical Report CECS-02-17, CECS, UC Irvine, 2002.

[7] T. Grötker et al. *System Design with SystemC*. Kluwer Academic Publishers, 2002.

[8] W. Hardt et al. The PARADISE design environment. In *ESC*, 1999.

[9] A. Jantsch et al. The Rugby model: A conceptual frame for the study of modelling, analysis and synthesis concepts of electronic systems. In *DATE*, 1999.

[10] D. Lyonnard et al. Automatic generation of application-specific architectures for heterogeneous multiprocessor system-on-chip. In *DAC*, 2001.

[11] Open SystemC Initiative. http://www.systemc.org.

[12] R. Siegmund and D. Müller. SystemC$^{SV}$: An extension of SystemC for mixed multi-level communication modeling and interface-based system design. In *DATE*, 2001.

[13] SpecC home page. http://www.cecs.uci.edu/˜specc.

[14] SpecC Technology Open Consortium. http://www.specc.org.

[15] K. Svarstad et al. A higher level system communication model for object-oriented specification and design of embedded systems. In *ASPDAC*, 2001.

[16] K. van Rompaey et al. CoWare: A design environment for heterogeneous hardware/software systems. In *EURO-DAC*, 1996.