System-Level Abstraction Semantics

Andreas Gerstlauer
Daniel D. Gajski

Technical Report CECS-02-17
July 12, 2002

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-3425, USA
(949) 824-8919

gerstl@cecs.uci.edu
gajski@cecs.uci.edu
http://www.cecs.uci.edu

System-Level Abstraction Semantics

Andreas Gerstlauer
Daniel D. Gajski

Technical Report CECS-02-17
July 12, 2002

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-3425, USA
(949) 824-8919

gerstl@cecs.uci.edu
gajski@cecs.uci.edu
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.

Contents

1 Introduction **1**
1.1 System Design Process 1
1.2 Traditional Models 2

2 Abstraction Levels **2**
2.1 Specification Model 3
2.2 Multiprocessing Model 4
2.3 Architecture Model 6

3 Experiments **7**

4 Summary & Conclusions **9**

References **9**

List of Figures

1 System design tasks. 2
2 Y-Chart. 2
3 System design flow. 3
4 Specification model definition. 4
5 Specification model example. 4
6 Multiprocessing model definition. 4
7 Multiprocessing model example. 5
8 Architecture model definition. 6
9 Architecture model example. 7
10 JPEG encoder. 8
11 Simulation performance. 9

System-Level Abstraction Semantics

Andreas Gerstlauer, Daniel D. Gajski
Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-3425, USA
{gerstl,gajski}@cecs.uci.edu
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.

1 Introduction

It is a well-known fact that designers of heterogeneous multiprocessor SOCs are facing an increasing productivity gap between semiconductor technology and methodology and tool support. Technological advances allow us to put complete systems on a single chip. A system-on-chip (SOC) design integrates heterogeneous architectures consisting of multiple processors, custom hardware blocks, intellectual property (IP) components, memories, and busses. However, with current methodologies and tools we will not be able to design such systems under the given time-to-market pressures.

A lot of efforts have been focussed on raising the level of abstraction for 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.

A requirement for any design flow is a set of well-defined abstraction levels and models. The number of models and the properties of each model have to be defined such that designers and tools can optimize decisions and move between models 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 models 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. However, 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,

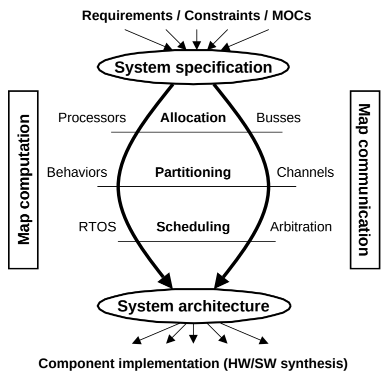


Figure 1: System design tasks.

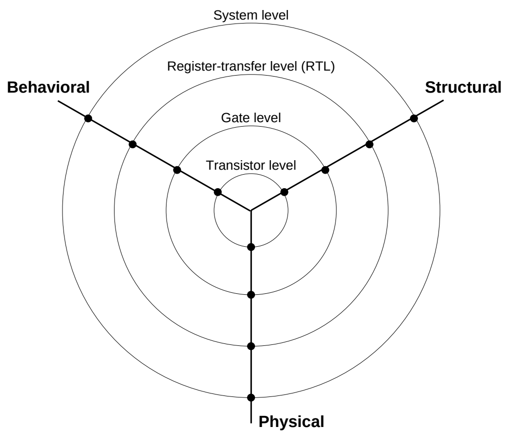


Figure 2: Y-Chart.

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 [1, 2, 3]. However, none of these defines an actual flow with models at specific points.

Traditionally, abstracted models of a 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 [5, 6]. At higher levels, different models of computation are combined into common simulation environments for specification [4]. 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 [7, 8, 10, 12]. 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.

2 Abstraction Levels

A general classification of the design process is available through the Y-Chart as shown in Figure 2 [1]. 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, the design can be described in the form of a behavioral, a structural model, or a physical 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. Finally, a physical model describes the spatial layout, i.e. the physical placement of the subcomponents on the chip.

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.

Given two events e_1 and e_2, where an event e_i is a tuple (a_i, t_i) of action a_i occurring at time t_i, e_1 and e_2 are ordered iff it can be determined that $t_1 < t_2$ or $t_2 < t_1$. A system is totally ordered if all pairs of events are ordered as is the case with real time on the chip, for example. A system is partially ordered if only subsets of all events are

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.

ordered. For example, at higher levels, a relationship between independent parts is not specified. An abstraction level employs a model of time to specify order. Real time is abstracted as discrete logical time. Two unordered events are modeled to occur at the same logical time, leaving the freedom of implementing them in any oder in real time.

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.

For each design task, the models at the input and output of the flow have to be defined such that the transformation between the models becomes possible. If the gap is too big to be done in one step, the task needs to be split, creating additional intermediate models. On the other hand, tasks should be as independent as possible in order to be able to solve them separately.

In summary, a model at a certain level of abstraction trades off accuracy and efficiency, for example in terms of simulation speed. Models have to be defined with the right amount of detail to allow rapid and meaningful exploration, synthesis, and validation.

In general, the system design process is too complex to be completed in one single step. The gap between requirements and implementation is, for all practical purposes (i.e. using non-exponential algorithms), impossible to cover. 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 dependencies. Especially, since partitioning of computation influences the amount of communication to be

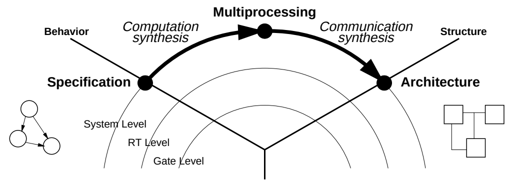


Figure 3: System design flow.

performed, computation synthesis needs to be performed before communication synthesis.

Figure 3 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 abstraction models. Table 1 summarizes the characteristics of the different models for system design.

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.

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

design	:= *computation* , *communication*
computation	:= *behavior*
behavior	:= *leaf* $\mid$ (*composition*)
composition	:= *sequential* $\mid$ *parallel* $\mid$ *pipelined* $\mid$ *alternative*
parallel	:= *behavior* $\|$ *behavior* $\|$ ...
sequential	:= *behavior* $\triangleright$ *behavior* $\triangleright$ ...
pipelined	:= *condition* : *behavior* $\mid$ *behavior* $\mid$ ...
alternative	:= *condition* : *behavior* $\vee$ *behavior* $\vee$ ...
communication	:= *message* , *message* , ...
message	:= *leaf* $\rightarrow$ *channel* $\rightarrow$ *leaf*

Figure 4: Specification model definition.

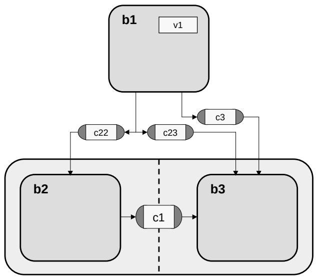


Figure 5: Specification model example.

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.

Production rules for terms describing specification models are given in Figure 4. Note that due to space constraints, definitions have been shortened and some details have been omitted. An example a simple yet typical specification model is shown in Figure 5.

In the example of Figure 5, the specification is a serial-parallel composition of *b1* followed by the concurrent execution of *b2* and *b3*. Leaves *b1*, *b2*, and *b3* contain functions over their input data. In summary, in the given notation the computation of the specification example is

$$b_1 \triangleright (b_2 \| b_3).$$

In terms of communication, behavior *b1* sends data to *b2* via channels *c22* and to *b3* via channels *b3* and *c23*. Again, using the notation defined in Figure 4, the communication part of our specification model example is

$$\begin{aligned} b_1 \rightarrow c_{22} \rightarrow b_2 \quad &, \\ b_1 \rightarrow c_{23} \rightarrow b_3 \quad &, \\ b_1 \rightarrow c_3 \rightarrow b_3 &. \end{aligned}$$

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.

design	:= *computation* , *communication*
computation	:= *PE* $\|$ *PE* $\|$ ...
PE	:= *processor* $\mid$ *IP*
processor	:= [*behavior*]
IP	:= [*IPbehavior* $\leftrightarrow$ *wrapper*]
behavior	:= *leaf* $\mid$ (*composition*)
composition	:= *sequential*
communication	:= *message* , *message* , ...
message	:= *leaf* $\rightarrow$ *channel* $\rightarrow$ *leaf*
	$\mid$ *leaf* $\rightarrow$ *wrapper* $\mid$ *wrapper* $\rightarrow$ *leaf*

Figure 6: Multiprocessing model definition.

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. The definition of the multiprocessing model is given in Figure 6. An exemplary multiprocessing model corresponding to the specification example from Figure 5 is shown in Figure 7.

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 structural objects like 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 runtime. 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 (data message) 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 a special case of IPs are dedicated memory components which do not provide any functionality apart from (read and write) access to stored data

In the case of the example of Figure 7, *b1* and *b3* are mapped onto processing elements *pe1* and *pe2*, respectively. *b2* is implemented by an existing IP component that provides the same functionality. A vendor-supplied black-box description *ip1* encapsulates a simulation, analysis and synthesis model of the IP while allowing integration into the system through a channel interface. A system memory *m1* holds variables *v1* and *v3* and provides read and write

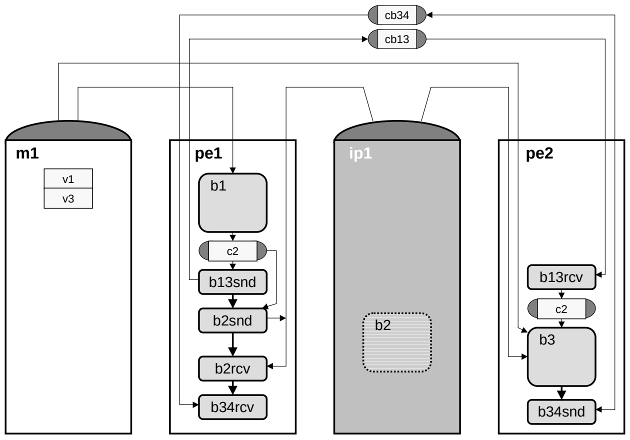


Figure 7: Multiprocessing model example.

access through its channel interface. On the other hand, local copies of the variable *v2* have been created in *pe1* and *pe2*. In addition, communication and synchronization blocks *bXXsnd* and *bXXrcv* have been inserted to preserve the original execution semantics. Execution of formerly sequential blocks mapped to concurrent PEs is synchronized, and updated variable values are communicated to keep local copies in sync. Finally, behavioral blocks inside *pe1* and *pe2* communicate via global channels *cbXX* or by accessing the channel interfaces of *m1* and *ip1* directly.

In summary:

$$\begin{aligned}
\mathit{Computation} &= m_1 \parallel pe_1 \parallel ip_1 \parallel pe_2 \\
m_1 &= [m^{ip} \leftrightarrow m_1^{wrapper}] \\
pe_1 &= [b_1, b_{13}^{snd}, b_2^{snd}, b_2^{rcv}, b_{34}^{rcv}] \\
ip_1 &= [b_2^{ip} \leftrightarrow ip_1^{wrapper}] \\
pe_2 &= [b_{13}^{rcv}, b_3, b_{34}^{snd}]
\end{aligned}$$

and

$$\begin{aligned}
\mathit{Communication} = \; & b_1 \rightarrow v_2 \rightarrow b_2^{snd} \;\& \\
& b_2^{snd} \rightarrow ip_1^{wrapper} \;\& \\
& b_1 \rightarrow v_2 \rightarrow b_{13}^{snd} \;\& \\
& b_{13}^{snd} \rightarrow cb_{13} \rightarrow b_{13}^{rcv} \;\& \\
& b_{13}^{rcv} \rightarrow v_2 \rightarrow b_2 \;\& \\
& b_1 \leftrightarrow m_1^{wrapper} \;\& \\
& M_1^{wrapper} \rightarrow b_3 \;\& \\
& ip_1^{wrapper} \rightarrow b_3 \;\& \\
& ip_1^{wrapper} \rightarrow b_2^{rcv} \;\& \\
& b_{34}^{snd} \rightarrow cb_{34} \rightarrow b_{34}^{rcv}.
\end{aligned}$$

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.

The multiprocessing model describes the implementation of the computation on the PEs of the system architecture. It is a structural view of the system's computational aspects. On the other hand, the multiprocessing model contains behavioral descriptions of the PEs that will feed into the lower parts of the design flow. Finally, at the multiprocessing level, communication between the PEs is exposed for implementation in the following steps.

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 as part of the multiprocessing model, 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 bus wires. It is obtained by adding bus protocols to all channels, splitting channels, and inlining them into each PE as bus drivers. Figure 8 shows the definition of the architecture model. The architecture model example corresponding to the previously shown multiprocessing model is shown in Figure 9.

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 bus wires, and $c : \bigcup_{p \in PE} O_p \mapsto B$ is the port mapping function connecting PE ports to bus wires. 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.

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.

$$\begin{array}{ll}
\textit{design} & := \textit{computation}\ ,\ \textit{communication} \\
\textit{computation} & := \textit{PE} \parallel \textit{PE} \parallel \ldots \\
\textit{PE} & := \textit{processor} \mid \textit{IP} \mid \textit{transducer} \mid \textit{arbiter} \\
\textit{processor} & := [\ \textit{behavior} \leftrightarrow \textit{busdriver} \leftrightarrow \textit{ports}\ ] \\
\textit{IP} & := \textit{processor} \mid \textit{structural} \\
\textit{transducer} & := [\ \textit{ports} \leftrightarrow \textit{busdriver} \leftrightarrow \textit{busdriver} \leftrightarrow \textit{ports}\ ] \\
\textit{arbiter} & := [\ \textit{busdriver} \leftrightarrow \textit{ports}\ ] \\
\textit{communication} & := \textit{connectivity}\ ,\ \textit{connectivity}\ ,\ \ldots \\
\textit{connectivity} & := \textit{port} \mapsto \textit{wire} \\
\textit{message} & := \textit{leaf} \rightarrow \textit{channel} \rightarrow \textit{leaf} \\
 & \quad \mid \textit{leaf} \rightarrow \textit{busdriver} \mid \textit{leaf} \leftarrow \textit{busdriver}
\end{array}$$

Figure 8: Architecture model definition.

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

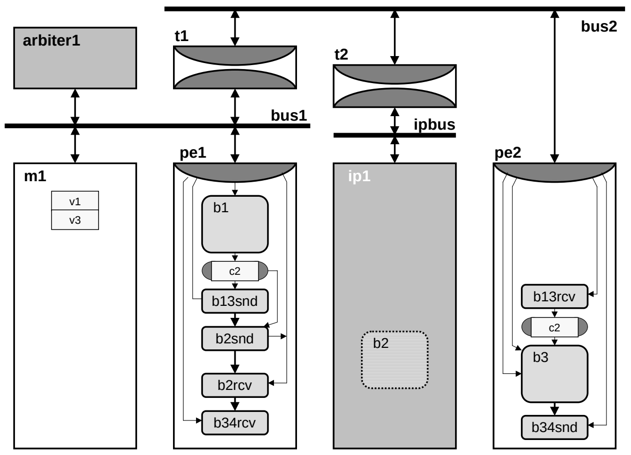


Figure 9: Architecture model example.

the form of IPs and as such can be defined as bus-functional or structural processor models.

In the example shown in Figure 9, the memory *m1* is connected to processor *pe1* via the processor's bus *bus1* while *ip1* and co-processor *pe2* are connected via *bus2*. Inside *pe1* and *pe2*, behavioral blocks connect to bus drivers that implement message-passing over the bus wires. Transducers *t1* and *t2* translate between incompatible bus protocols. *t1* acts as a bridge between busses *bus1* and *bus2*. *t2* interfaces the IP with its proprietary protocol to *bus2*. The channel interface of *ip1* in the architecture model is moved into *t2* where it implements communication with *ip1* over the exposed wires of the IP bus. Finally, an additional PE *arbiter1* that regulates conflicting accesses of *pe1* and *t1* on *bus1* is inserted.

Using the notation defined in Figure 8, the architecture model example is formalized as follows:

$$\begin{aligned}
Computation &= m_1^{ip} \,||\, pe_1^{p} \,||\, ip_1^{ip} \,||\, pe_2^{p} \,|| \\
&\quad\; arbiter_1 \,||\, t_1 \,||\, t_2 \\
pe_1^{p} &= [pe_1 \leftrightarrow pe_1^{driver} \leftrightarrow pe_1^{ports}] \\
pe_1 &= b_1, b_{13}^{snd}, b_2^{snd}, b_2^{rcv}, b_{34}^{rcv} \\
pe_2^{p} &= [pe_2 \leftrightarrow pe_2^{driver} \leftrightarrow pe_2^{ports}] \\
pe_2 &= b_{13}^{rcv}, b_3, b_{34}^{snd} \\
t_1 &= [t_1^{ports_1} \leftrightarrow t_1^{driver_1} \leftrightarrow t_1^{driver_2} \leftrightarrow t_1^{ports_2}] \\
t_2 &= [t_2^{ports_{ip}} \leftrightarrow t_2^{driver_{ip}} \leftrightarrow t_2^{driver_2} \leftrightarrow t_2^{ports_2}]
\end{aligned}$$

and

$$\begin{aligned}
Communication = \;& b_1 \rightarrow v_2 \rightarrow b_2^{snd} \;\; \& \\
& b_2^{snd} \rightarrow pe_1^{driver} \;\; \& \\
& b_1 \rightarrow v_2 \rightarrow b_{13}^{snd} \;\; \& \\
& b_{13}^{snd} \rightarrow pe_1^{driver} \;\; \& \\
& pe_2^{driver} \rightarrow b_{13}^{rcv} \;\; \& \\
& b_{13}^{rcv} \rightarrow v_2 \rightarrow b_2 \;\; \& \\
& b_1 \leftrightarrow pe_1^{driver} \;\; \& \\
& pe_2^{driver} \rightarrow b_3 \;\; \& \\
& pe_2^{driver} \rightarrow b_3 \;\; \& \\
& pe_1^{driver} \rightarrow b_2^{rcv} \;\; \& \\
& b_{34}^{snd} \rightarrow pe_1^{driver} \;\; \& \\
& pe_2^{driver} \rightarrow b_{34}^{rcv} \;\; \& \\
& m_1^{ports} \mapsto bus_1 \;\; \& \\
& pe_1^{ports} \mapsto bus_1 \;\; \& \\
& arbiter_1^{ports} \mapsto bus_1 \;\; \& \\
& t_1^{ports_1} \mapsto bus_1 \;\; \& \\
& t_1^{ports_2} \mapsto bus_2 \;\; \& \\
& pe_2^{ports} \mapsto bus_2 \;\; \& \\
& t_2^{ports_2} \mapsto bus_2 \;\; \& \\
& t_2^{ports_{ip}} \mapsto bus_{ip}.
\end{aligned}$$

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 [16, 17] and SpecC [13, 14, 15]. 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 the application of the abstraction levels to the SpecC language can be found in [18].

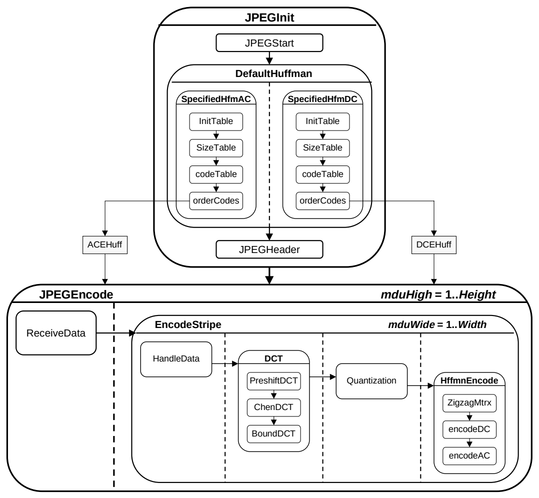
We then modeled several design examples following the presented flow. In the following, we will outline the implementation of a JPEG encoder [19, 20]. Note that the focus was on demonstrating feasibility and effectiveness of the models. Therefore, implementation decisions were made without performing elaborate design space exploration. Source code for all models in SpecC can be downloaded from our web pages [21]. 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 10 shows the three models of the JPEG encoder. At the top of the specification model (Figure 10(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 8×8 blocks and processes each block through DCT, quantization and Huffman encoding. As an example of communication, Figure 10(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.

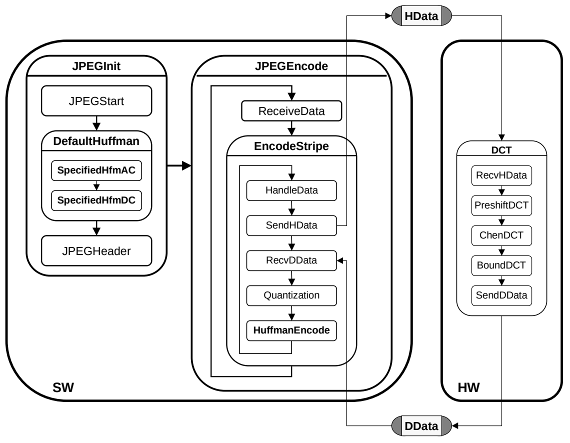
For the purpose of computation synthesis, we assumed a mapping of the encoder on a Motorola Coldfire processor (*SW*) assisted by a custom hardware co-processor (*HW*) for acceleration of the DCT (Figure 10(b)). Software and hardware communicate via two message-passing channels, sending and receiving 8×8 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.

In Figure 10(b), the software waits for the result of the DCT before continuing with any processing. By changing only a few lines of code, we were able to modify the architecture such that software and hardware operate in a pipelined fashion (i.e. while the DCT is processing a block the software continues processing of the previous block and prepares the next one), resulting in 100% utilization of the *SW* processor. Similarly, other architectural alternatives can be easily explored in a very short amount of time with minimal changes in the model.

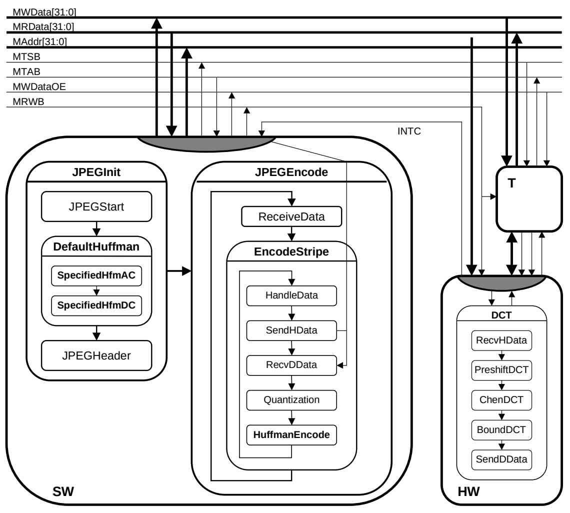
Finally, for communication synthesis, we connected the two processors via a single bus using the Coldfire bus protocol. Furthermore, it was assumed that the protocol of the DCT IP is fixed and incompatible with the Coldfire protocol, necessating the inclusion of a transducer (Figure 10(c)). The *SW* processor is the master on the bus and drives the address and control lines. The *HW* co-processor listens directly on the address bus and its associated control lines while the transducer translates between data transfer protocols. For synchronization, the hardware signals the software through the processor's interrupt line *INTC*. In-



(a) Specification



(b) Multiprocessing



(c) Architecture

Figure 10: 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.

side the two 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 models 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 11. 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 model, 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 covering the flow from specification to architecture. 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 approach at properly defining models in a formalized way. The models define a

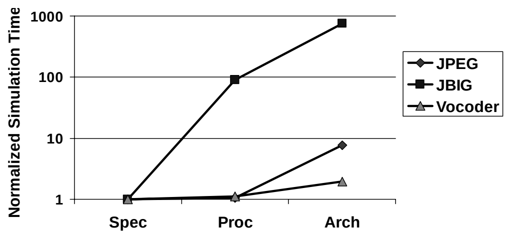


Figure 11: Simulation performance.

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 models is the enabler for interoperability and design automation. Based on the abstract definitions, we can demonstrace 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.

Acknowledgments

This work was supported by the Semiconductor Research Cooperation (SRC) under contract 2000-HJ-832 with Task ID 832.002.

References

[1] D. D. Gajski, R. Kuhn. "Guest editors introduction - New VLSI tools." *IEEE Computer*, pp. 11-14, 1983.

[2] W. Hardt et al. "The PARADISE design environment." In *Proceedings Embedded Systems Conference*, 1999.

[3] A. Jantsch et al. "The Rugby model: A conceptual frame for the study of modelling, analysis and synthesis concepts of electronic systems." In *Proceedings Design, Automation, and Test in Europe*, 1999.

[4] J. T. Buck et al. "Ptolemy: a framework for simulating and prototyping heterogeneous systems." *Int. Journal of Computer Simulation*, special issue on Simulation Software Development, vol. 4, pp. 155-182, April 1994.

[5] P. Coste et al. "Multilanguage design of heterogeneous systems." In *Proceedings International Workshop on Hardware-Software Codesign*, May 1999.

[6] P. Gerin et al. "Scalable and flexible cosimulation of SoC designs with heterogeneous multi-processor target architectures." In *Proceedings of the Asia and South Pacific Design Automation Conference*, Yokohama, Japan, February 2001.

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

[8] D. Lyonnard et al. "Automatic generation of application-specific architectures for heterogeneous multiprocessor system-on-chip." In *Proceedings 2001 Design Automation Conference*, Las Vegas, June 2001.

[9] A. Baghdadi, D. Lyonnard, N.-E. Zergainoh, A. A. Jerraya. "An efficient architecture model for systematic design of application-specific multiprocessor SoC." In *Proceedings Design, Automation, and Test in Europe*, Munich, Germany, March 2001.

[10] R. Siegmund, D. Müller. "SystemC^{SV}—An extension of SystemC for mixed multi-level communication modeling and interface-based system design." In *Proceedings Design, Automation, and Test in Europe*, Munich, Germany, March 2001.

[11] K. Svarstad, G. Nicolescu, A. A. Jerraya. "A model for describing communication between aggregate objects in the specification and design of embedded systems." In *Proceedings Design, Automation, and Test in Europe*, Munich, Germany, March 2001.

[12] K. Svarstad et al. "A higher level system communication model for object-oriented specification and design of embedded systems." In *Proceedings of the Asia and South Pacific Design Automation Conference*, Yokohama, Japan, February 2001.

[13] D. D. Gajski et al. *SpecC: Specification Language and Design Methodology*. Kluwer Academic Publishers, 2000.

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

[15] SpecC Technology Open Consortium (STOC). http://www.systemc.org.

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

[17] Open SystemC Initiative (OSCI). http://www.systemc.org.

[18] A. Gerstlauer, D. D. Gajski. *SpecC Modeling Guidelines*. Technical Report CECS-TR-02-16, Center for Embedded Computer Systems, UC Irvine, April 2002.

[19] L. Cai et al. *Design of a JPEG Encoding System*. Technical Report ICS-TR-99-54, Information and Computer Science, UC Irvine, November 1999.

[20] H. Yin, H. Du, T.-C. Lee, D. D. Gajski. *Design of a JPEG Encoder using SpecC Methodology*, Technical Report ICS-TR-00-23, Information and Computer Science, UC Irvine, July 2000.

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