Embedded Systems Conference San Francisco 2002

The SpecC System-Level Design Language and Methodology, Part 2

Class 349

Rainer Dömer*, Andreas Gerstlauer*
Paul Kritzinger**, Mike Olivarez**

*Center for Embedded Computer Systems
Universitiy of California, Irvine, USA
**Motorola Semiconductor Products Sector
Advanced Systems Architectures, Austin, TX, USA

Abstract

A well-defined design methodology supported by a system-level design language (SLDL) is the key for managing the complexity of the design flow, especially at the system level. Only with well-defined and unambiguous models and transformations can we achieve productivity gains through synthesis, verification and tool interoperability. This paper presents the SpecC system design methodology. It shows how, through gradual, stepwise refinement, a design is taken from specification down to implementation. Finally, it introduces a design example of industrial-strength that has been implemented following the methodology, including the results and productivity gains achieved.

This is the second paper in a two-part series. This part covers the SpecC methodology and its application to an industrial design example, a GSM vocoder.

1 Introduction

In general, system design is the process of implementing a given specification on a chip in silicon. It is a two-step process that gradually moves the design to lower levels of abstraction. First, a system architecture is derived from the specification. Then, the system components are implemented down to their register-transfer level (RTL) or instruction-set (IS) architecture.

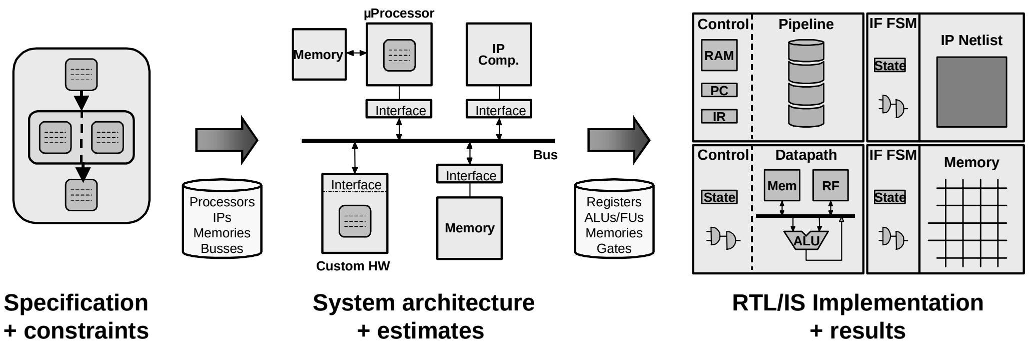

Figure 1: System-On-Chip Design.

At each level of abstraction, a behavioral description is converted into a structural description that implements the original behavior and satisfies given constraints. This process is also called *synthesis*.

During system synthesis, a system architecture is defined by allocating a set of components like processors, memories, custom hardware or IP components that communicate via a set of system busses. The functionality of the specification is then mapped onto this architecture.

During component synthesis, the components of the system architecture are implemented by designing their microarchitecture. For each component, a datapath consisting of functional units, register files, memories, and busses is defined. Finally, the desired behavior of custom hardware or software components is implemented on top of the their RTL or instruction-set microarchitecture, respectively.

1.1 Abstraction Levels

As the system design progresses through the different stages, the system specification is gradually refined from an abstract idea down to an actual implementation. This refinement is achieved in a stepwise manner through several layers of abstraction.

System design starts with a set of requirements and constraints, both in terms of functionality and quality. Initially, functional requirements are possibly captured in different models of computation (MOC) for different application domains. For example, data-flow models are frequently used to describe data-dominated parts of the system.

The actual design process, however, starts with a single, common system *specification* derived from the requirements and the different MOCs. The specification describes the system functionality in a unified way as a starting point for system synthesis. It is free of any implementation detail.

At the *architecture level*, the component structure of the system architecture is defined. The system functionality is partitioned and partitions are assigned to different components. In the process, the computational parts of the system are ordered based on execution times and a scheduling of computation on each component.

At the *communication level*, components are refined into bus-functional representations, which accurately describe the timing of events on the wires of the busses.

Finally, at the *implementation level*, the components are defined in terms of their register-transfer or instruction-set architecture. The granularity of time in the system is refined down to individual clock cycles in each component.

Eventually, the design will be further refined down to a gate-level structure with timing of system events in terms of sub-cycle delays. From there on, the design is then taken into placement and routing, physical layout, and finally manufacturing.

1.2 Design Flow

The SpecC design flow is based on four abstraction levels, namely specification, architecture, communication, and implementation level. The design starts with a specification model captured by the user based on algorithms of his/her choice.

The system synthesis process is then subdivided into two tasks: *architecture exploration* maps the computation in the specification onto system components that are instantiated out of a component library. During architecture exploration, the specification model is refined into the intermediate architecture model.

Then, *communication synthesis* refines the abstract communication in the architecture model into an implementation over actual wires of system busses. The system components are refined into bus functional models that communicate over bus wires using protocols selected from a protocol library.

The result of the system synthesis process is the communication model, which is then handed off to the backend tools for RTL or instruction-set level implementation. Hardware components are synthesized into a microarchitecture of RTL-components, software is compiled into the processor's instruction set, and interface logic and bus drivers are generated on the hardware and software side, respectively.

The final result of the system design process is the implementation model.

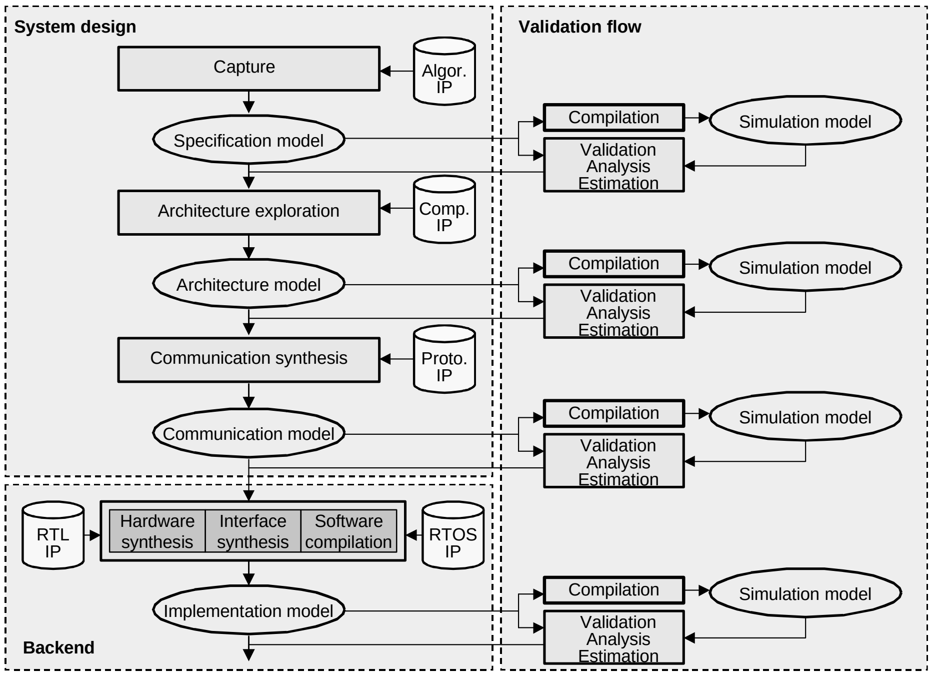

Figure 2: SpecC Methodology.

1.3 SpecC Methodology

The complete SpecC system-level design methodology is shown in Figure 2. In the SpecC methodology, all models are written in the SpecC language. The design is represented by a corresponding description in SpecC at each stage. All models are executable for validation through simulation (right), reusing the same test bench throughout the whole design process. In addition to simulation, the formal nature of the models enables the application of formal methods for verification, analysis or estimation, for example.

The well-defined nature of the whole design process is the basis for rapid design space exploration through automatic model refinement and other design automation.

Next, we will explain the different models and refinements of the SpecC methodology in detail. Based on a simple design example, we will walk through the methodology step by step.

2 Specification Model

The system design process starts with the specification model written by the user to specify the desired system functionality. It forms the input to architecture exploration, the first step of the system design process. Therefore, it defines the basis for all exploration and synthesis. For example, the specification model defines the granularity for exploration through the size of the leaf behaviors, it exposes the available parallelism, it separates communication from computation, and it uses hierarchy to group related functionality and to manage complexity.

The specification model is a purely functional, abstract model that is free of any implementation details. The hierarchy of behaviors in the specification model solely reflects the system functionality without implying anything about the system architecture to be implemented.

The specification model is also free of any notion of time. The model executes in zero simulation time. Events in the specification model are used for synchronization, which establishes a partial ordering among the behaviors based on desired causality.

In general, at each level of hierarchy the specification is an arbitrary serial-parallel composition of behaviors. Behaviors communicate through variables and synchronize through events attached to their ports. At the lowest level of hierarchy, leaf behaviors execute the algorithms in the form of C code.

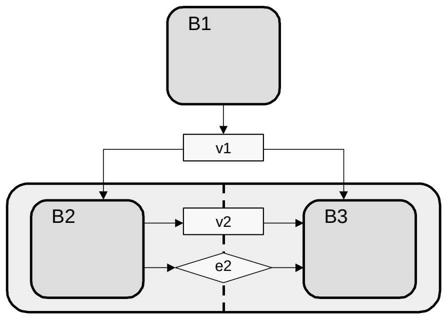

Figure 3: Specification Model Example.

Figure 3 shows an example of a simple yet quite typical specification model. Execution starts with leaf behavior *B1*, followed by the parallel composition of leaf behaviors *B2* and *B3*. *B1* produces variable *v1*, which then is consumed by both *B2* and *B3*. In addition, the concurrent behaviors *B2* and *B3* exchange data and synchronize through variable *v2* and event *e2*. *B2* writes to *v2* and notifies *B3* about the availability of data via event *e2*. After receiving event *e2*, *B3* in turn then reads the data from variable *v2*.

3 Architecture Exploration

Architecture exploration is the first part of the system synthesis process. It derives a system architecture from the specification model. The purpose of architecture exploration is to map the computational parts of the specification represented by the behaviors onto the components of a system architecture.

The main steps involved in this process are *behavior partitioning*, *variable partitioning*, and *scheduling*.

3.1 Behavior Partitioning

The first step in architecture exploration is the allocation of a set of processing elements (PEs) and the mapping of the specification behaviors onto the allocated PEs. This process determines the groups of behaviors that will define the functionality to be implemented by each PE.

For our example, we assume an allocation of two processing elements, *PE1* and *PE2*, and a mapping of leaf behaviors *B1* and *B2* onto *PE1* whereas leaf behavior *B3* will execute on *PE2*.

In the SpecC description, PE allocation and behavior mapping is modeled by inserting an additional level of hierarchy at the top of the behavior hierarchy. Here, a set of concurrent behaviors representing the PEs of the system architecture is introduced.

The leaf behaviors are grouped under those newly added PE behaviors according to the selected mapping, replicating the original behavior hierarchy in each PE as necessary. In order to preserve the execution semantics of the original specification, synchronization is added between PEs for each pair of sequential behaviors mapped to concurrent PEs.

Finally, communication between behaviors on different PEs becomes system-global communication and is moved to the top-level that contains the PE behaviors.

The refined model of our example after PE allocation and behavior partitioning is shown in Figure 4. At the top level, the design is transformed into a parallel composition of two newly inserted behaviors, *PE1* and *PE2*, which represent the components of the system architecture. Inside *PE1*, leaf behaviors *B1* and *B2* are instantiated, replicating the original behavior hierarchy of the specification model. Leaf behavior *B3*, on the other hand, is executing inside component behavior *PE2*. Behavior *B3* is the only leaf behavior grouped under *PE2*. Other empty parts of the behavior hierarchy have been collapsed and removed in this PE.

The variable *v1* and the message-passing channel *C2* for communication between behaviors *B1* and *B2* on the one hand and *B3* on the other hand have become global variables and channels, respectively. They are instantiated at the top level, connecting the behaviors inside the two PEs. These global variables and channels represent all the communication occurring between PEs.

In order to preserve the semantics of the original specification, pairs of behaviors communicating via message-passing channels must be inserted to synchronize sequential behaviors executing on concurrent PEs.

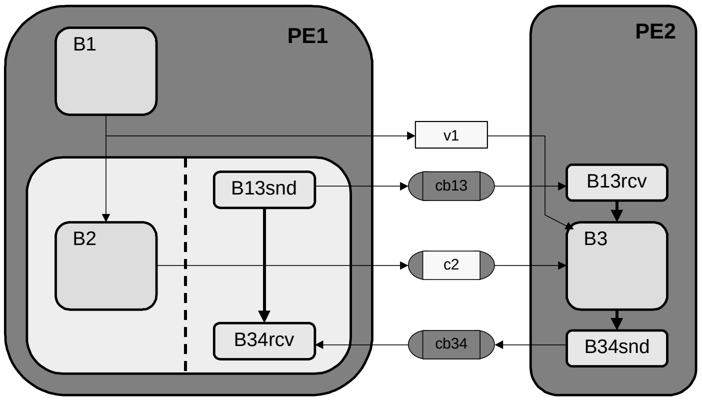

Figure 4: Model after Behavior Partitioning.

As shown for our example, two synchronization behavior pairs and two channels are inserted. Behaviors *B13Snd* on *PE1* and *B13Rcv* on *PE2* synchronize execution of behaviors *B1* and *B3* between *PE1* and *PE2* via channel *cb13*. In accordance with the original specification model, behavior *B3* is not allowed to start executing until behavior *B1* has finished. Similarly, the synchronization behavior pair *B34Snd* and *B34Rcv* ensures that *PE1* does not complete and start a new iteration until behavior *B3* has finished.

3.2 Variable Partitioning

At this point, the set of global variables instantiated between the PE behaviors represents global storage that has to be mapped to actual memories in the system architecture. In a straightforward implementation, global variables are mapped to a dedicated shared memory that is allocated together with the processing elements and included in the system architecture.

Alternatively, in a message-passing architecture shared variables are mapped to the local memories of the processing elements. A local copy of the variable is created in each component that is accessing the variable. As shown in Figure 5, a local copy of variable *v1* is created in both *PE1* and *PE2*. The behaviors inside the PEs are then operating on the data in the local memory instead of accessing a global variable.

However, in order to preserve the shared semantics of the variable and to keep the local copies inside the PEs in sync, updated data values have to be exchanged between the components at synchronization points. Therefore, updated data values are communicated over the existing channels together with behavior synchronization. In our example, the new value of *v1* produced by *B1* on *PE1* is passed over the channel *cb13* together with transferring control from *B13Snd* to *B13Rcv*.

3.3 Scheduling

The next step in the architecture exploration process is the scheduling of behavior executions on the processing elements. Processing elements have a single thread of control only. Therefore, behaviors mapped to the same PE can only execute sequentially and have to be serialized.

For example, a static scheduling of *PE1* in our design example will serialize the parallel composition of behavior *B2* and the *B3Stub* behaviors. Assuming *B13Snd* is scheduled before *B2* and *B34Rcv* after *B2*, the hierarchy is flattened and the behaviors are executed in that order in the refined SpecC model.

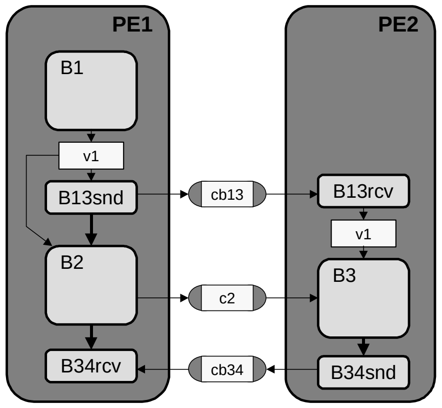

Figure 5: Architecture Model.

The final model of the design after scheduling is shown in Figure 5. At the top level, the model remains unchanged. The design is a parallel composition of component behaviors *PE1* and *PE2* communicating via message-passing channels *cb13*, *c2*, and *cb34*. However, the *PE2* behavior has been replaced with a refined model of the component that reflects the static scheduling of the behaviors inside *PE1*.

4 Architecture Model

Scheduling is the final step of architecture exploration and the resulting model is therefore the so-called architecture model. It is an intermediate model of the system design process.

The architecture model reflects the component structure of the system architecture. At the top-level of the behavior hierarchy, the design is a set of concurrent, non-terminating component behaviors. However, communication is still on an abstract level and components communicate via message-passing channels. The communication synthesis task that follows will implement the abstract communication over busses with real protocols.

The behaviors grouped under the components specify the desired functionality for the implementation of the component during later stages.

Concurrency is limited to the top-level of the design in the architecture model. All the concurrency in the design at this point is captured by the set of components running in parallel. Inside each component, behaviors execute sequentially in a certain order.

The architecture model is timed in terms of the computational parts of the design. Behaviors are annotated with estimated execution delays for simulation feedback, verification and further synthesis.

5 Communication Synthesis

Communication synthesis refines the abstract communication between components in the architecture model into an actual implementation over wires and protocols of system busses.

The steps involved in this process are *channel partitioning*, *protocol insertion*, *protocol inlining*.

5.1 Channel Partitioning

The first step in the process of implementing communication over system busses is the allocation of a set of busses and the mapping of communication channels onto those busses. This process determines the groups of channels to be implemented by each bus.

In our design example, we have only two components communicating with each other. Therefore, only one system bus, *Bus1*, is allocated connecting *PE1* and *PE2*. All communication channels are mapped onto that bus.

In the SpecC description, bus allocation and channel mapping is modeled by inserting an additional level at the top of the channel hierarchy. The new top-level channels represent the allocated system busses. The channels instantiated between the components are grouped under the bus channels according to the selected mapping.

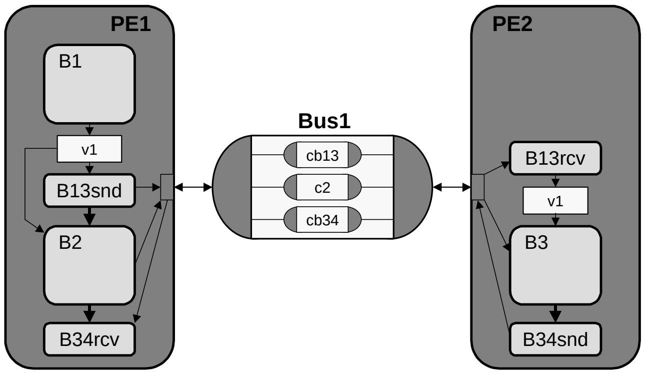

Figure 6: Model after Channel Partitioning.

Figure 6 shows the refined model of our example after bus allocation and channel partitioning. At the top level, a new channel, *Bus1*, has been inserted between the two PEs. The abstract channels *cb13*, *c2*, and *cb34* for communication between *PE1* and *PE2* are grouped as subchannels under the bus channel.

The two component behaviors *PE1* and *PE2* are connected to the bus through corresponding bus ports and bus interfaces. All inter-component connections are multiplexed over the single bus.

5.2 Protocol Insertion

The next step in communication synthesis is the insertion of actual bus protocols into the model. In the process, the abstract bus channels are replaced with an actual implementation of their semantics over the real bus protocol.

A description of the protocol is taken out of the protocol library in the form of a protocol channel. The protocol channel encapsulates the bus wires and implements the bus protocol by driving and sampling bus wires according to the protocol timing constraints. At its interface, the protocol channel provides methods for all primitive transactions supported by the protocol like read, write, burst read, burst write, and so on.

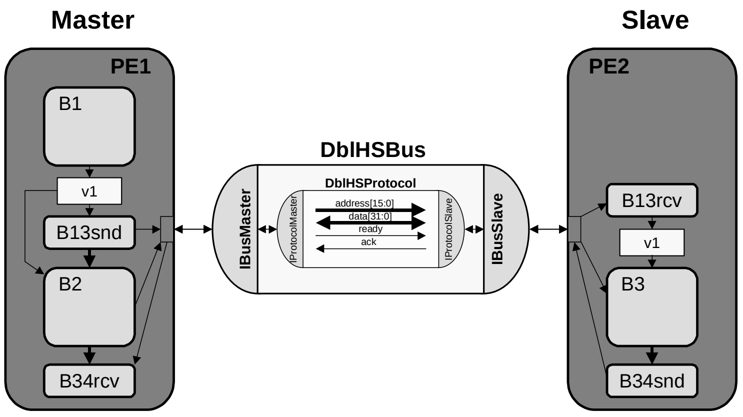

Figure 7: Model after Protocol Insertion.

On top of the protocol layer, an application layer is created that implements the abstract message-passing semantics over the bus protocol. The application layer wraps around the protocol layer and instantiates the protocol channel internally. The functionality of the application layer includes synchronization, arbitration, bus addressing, and data slicing.

Finally, the abstract bus channels in the model are replaced with their equivalent hierarchical combinations of protocol and application layers that implement the communication of each bus.

Figure 7 shows the refined model of our design example after inserting a double-handshake protocol for the bus between *PE1* and *PE2*. The bus channel has been replaced with the hierarchical combination of application layer channel *DblHSBus* and protocol layer channel *DblHSProtocol*. The PE behaviors are connected to the new bus channel via their bus ports. In this case, it was decided to make *PE1* the bus master and *PE2* the bus slave.

5.3 Protocol Inlining

After protocols have been inserted for the busses in the system, the communication is finally inlined into the components. The communication functionality is moved into the components where it will later be implemented together with the behaviors mapped onto the components.

During inlining, the application layer and protocol layer channels are split and the code is moved into the components according to their connectivity. After inlining, the bus wires internal to the protocol layer are exposed and the components are connected to the bus wires via corresponding ports. Inside the components, adapter channels containing application layer and protocol layer methods required by the component are instantiated. On the one side, the hierarchical adapters are connected to the component ports and their methods drive and sample the bus wires via the adapter ports. On the other side, the behaviors inside the PEs are connected to the interfaces of the adapter channels, calling the bus interface methods provided by the adapters.

As shown in Figure 8, the *DblHSBus* channel in our example is split into two halves, *PE1Bus* and *PE2Bus*, which are moved into the component behaviors *PE1* and *PE2*, respectively. The variables representing the bus wires for *address* bus, *data* bus and the two control lines, *ready* and *ack*, are exposed and the PEs are connected to the wires via corresponding ports.

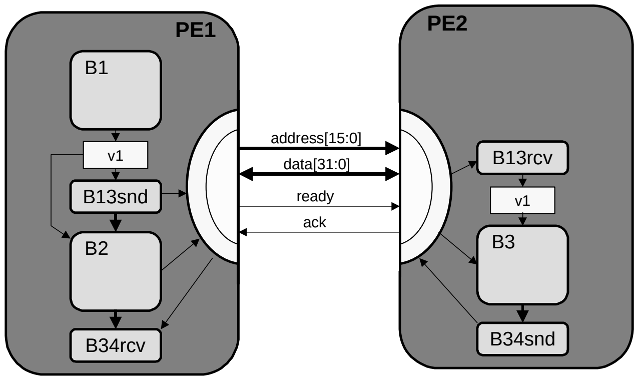

Figure 8: Model after Protocol Inlining.

6 Communication Model

The model after protocol inlining is called the communication model. It is the final result of the system synthesis process and as such defines the structure of the system architecture in terms of both components and connections. Computation has been mapped onto components and communication onto busses.

At the top-level of the hierarchy, the communication model is a parallel composition of a set of non-terminating components communicating via a set of system busses. Inside the components, a sequence of behaviors describes their functionality. The behaviors also define the timing of bus transactions as determined by the communication calls executed by the code.

At their interfaces, the components therefore provide a timing-accurate model of the component functionality down to the level of events on the bus wires. As a result, the communication model is timed in terms of both computation and communication.

7 Backend

In the backend, the behavioral views of the components in the communication model are converted into structural descriptions of each component's microarchitecture. The functionality of each component is implemented as custom hardware described by its RTL model, as processor software compiled into an instruction-set stream, or as an IP with fixed functionality. In the process, timing is refined down to the level of individual clock cycles based on each component's clock period. Therefore, the implementation model is cycle-accurate.

The backend process encompasses three parallel synthesis tasks for hardware, software, and interfaces.

7.1 Hardware Synthesis

On the hardware side, high-level synthesis (HLS) is performed. High-level synthesis of custom hardware requires scheduling of the code into clock cycles. The C code inside the leaf behaviors of the component is scheduled by drawing clock boundaries between the statements. The list of statements between clock boundaries defines the data-path operations performed in each clock cycle and the set of clock boundaries defines the states of the hardware control unit.

For example, under the assumption that the *PE2* component in our design example will be implemented as custom hardware, its leaf behavior *B3* needs to be scheduled. Given *PE2*'s clock, clock boundaries are introduced into the list of *B3*'s C code statements.

7.2 Software Synthesis

On the software side, the computation represented by the behaviors executing on the programmable processor component is implemented by compiling the code into the instruction set of the processor. For our design example, we assume that the *PE1* component will be implemented as a general-purpose microprocessor.

Software synthesis is a two-step process: code is generated from the SpecC model of the component and the generated code is compiled into the instruction-set of the target processor.

7.3 Interface Synthesis

Also, the communication functionality represented by the application and protocol layers of the bus adapter channels needs to be implemented on the target components as part of the backend process.

On the hardware side, bus interface logic is synthesized as part of the custom hardware. For example the bus adapter *PE1Bus* is refined into an FSMD model that drives and samples the bus wires in terms of the component clock.

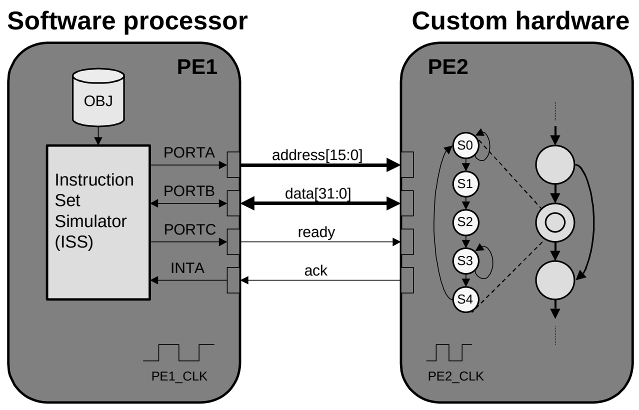

Figure 9: Implementation Model.

On the software side, bus drivers are generated which implement the application and protocol layer functionality over the processor's I/O instructions. For example, the bus adapter *PE2Bus* on the processor *PE2* is compiled into a bus driver library, which will be linked against the rest of the processor's program.

Figure 9 shows the implementation model after the refinement in the backend process. The PE behaviors are replaced with refined models of hardware, software and interfaces.

8 Implementation Model

The implementation model is the result of the backend process and as such the final end-result of the whole system design flow. It is a structural description of the system down to the component microarchitectures.

At the top-level, the system architecture is a set of non-terminating, concurrent components communicating via system busses. At the component level, computation and communication functionality is described on top of the component's microarchitecture: FSMD models for custom hardware and instruction-set models for software on programmable processors.

The implementation model is a cycle-accurate system description. The order and timing of computation and computation in the system is described in terms of component clocks. A global order is imposed among the system's components via the order of events on the common bus wires.

9 Design of a GSM Vocoder

In order to demonstrate the benefits of the SpecC methodology, we will now walk through the design process of a GSM vocoder application.

The vocoder project was done at the University of California, Irvine, in close connection with Motorola [1]. The purpose of the project was to apply the SpecC methodology to an industrial size example and to demonstrate and evaluate the methodology's effectiveness and benefits.

The voice encoding/decoding part of the GSM standard for mobile telephony was chosen as the basis for the project as a medium-size application which is beyond pure toy examples but yet small enough to be feasible in the relatively short project time.

9.1 GSM Vocoder Standard

The so-called *Enhanced Full-Rate (EFR) Speech Transcoding* is a standard that is part of the world-wide GSM system for cellular phone networks. The lossy compression scheme is an instance of a class of widely used speech encoding/decoding algorithms based on a so-called *code-excited linear predictive* (CELP) coding model. In this particular case, the GSM Vocoder encodes incoming speech samples at a rate of 104 kbit/s into an encoded bit stream with a rate of 12.2 kbit/s.

The CELP voice encoding scheme is based on a speech synthesis model which tries to emulate the way in which speech is generated in the human vocal tract. The combination of the output of a long term pitch filter and a set of residual pulses out of a fixed codebook models the buzz produced by the human vocal chords. This excitation is then fed into a short-term, *linear prediction* (LP) synthesis filter that models the modulation occurring in the human throat and mouth as a system of lossless tubes.

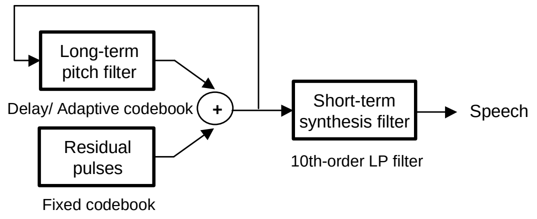

Figure 10: GSM Vocoder Speech Synthesis Model.

Instead of transmitting compressed speech samples, the filter parameters of the speech synthesis model are extracted in the encoder, transmitted, and used for driving the synthesis of speech in the decoder. In the encoder, parameters are extracted such that the mean-square error between synthesized and original speech is minimized. Parameter extraction operates on frames of 160 samples corresponding to 20 ms of speech. Each frame is further subdivided into 4 subframes and one set of parameters is transmitted per subframe.

The Vocoder standard specifies a maximal total latency of 10 ms for the first subframe when operating encoder and decoder in back-to-back mode. In addition, a complete frame has to be encoded and decoded within the 20 ms before the next frame arrives.

9.2 Specification Model

The original vocoder standard published by the European Telecommunication Standards Institute (ETSI) contains a bit-exact reference implementation of the standard in C. This reference code was taken as the basis for developing the SpecC specification model. At the lowest level, the C algorithms were directly reused by encapsulating them in SpecC leaf behaviors. However, the C function hierarchy had to be converted into a clean and efficient SpecC hierarchy by analyzing dependencies, exposing available parallelism, grouping related parts hierarchically, and so on. In contrast to the original C code, the SpecC specification describes the vocoder functionality in a clear and concise manner, which greatly eases understanding for both the user and any automated tools.

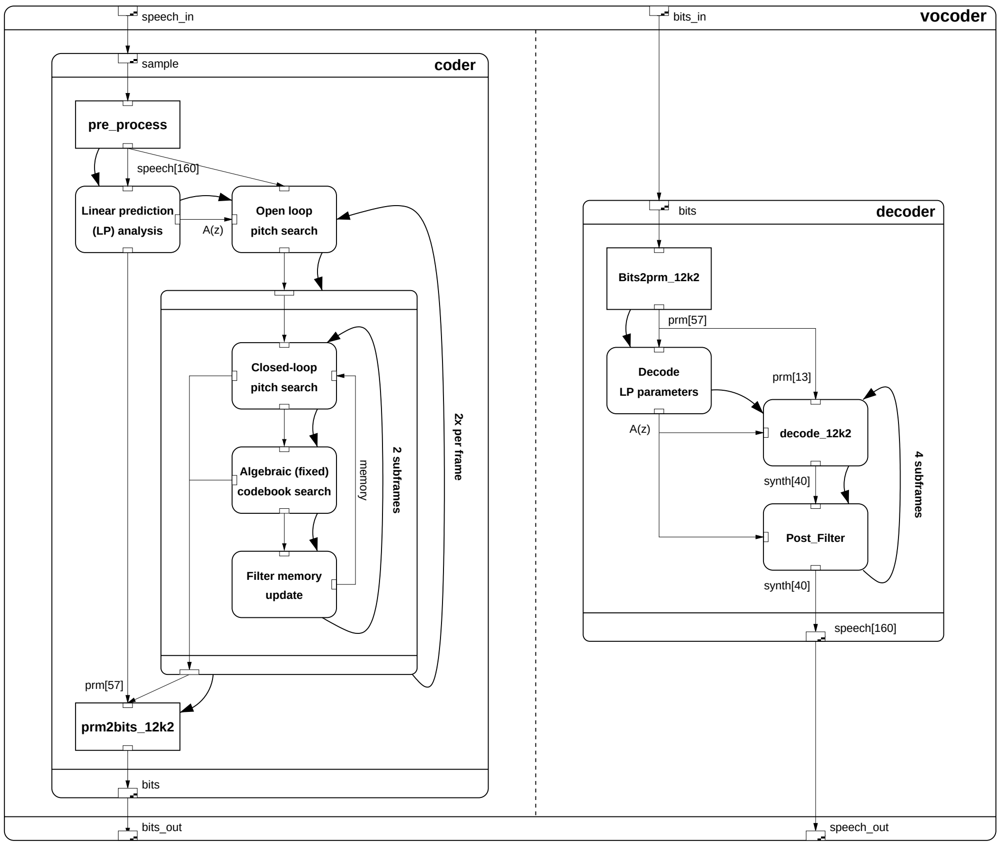

Figure 11: Specification Model of the Vocoder.

At the top level, the specification model runs encoding and decoding behaviors in parallel, as shown in Figure 11. Note that, due to space constraints, only the top levels of the hierarchy are shown here. In total, the specification model of the vocoder contains 43 leaf behaviors and consists of 13,000 lines of code.

9.3 Profiling

Before the start of the actual design process, an initial profiling of the specification was performed to extract characteristics and to analyze the code.

The chart in Figure 12 shows the profiled computational complexity of the different parts of the encoder. For each behavior, the number of operations were counted during simulation. Operation counts are multiplied with a factor according to their relative weights, summed, and combined with the timing constraints to derive a WMOPS rating for each behavior. The graph shows the result for the four different parts of the encoder: LP analysis, open-loop pitch search, closed-loop pitch search and codebook search. For each part, the total WMOPS per frame are shown for the total part and for all the leaf behaviors within. As the graph shows, the codebook search is by far the most demanding and critical part of the vocoder. Therefore, design efforts need to focus on this hotspot first and foremost.

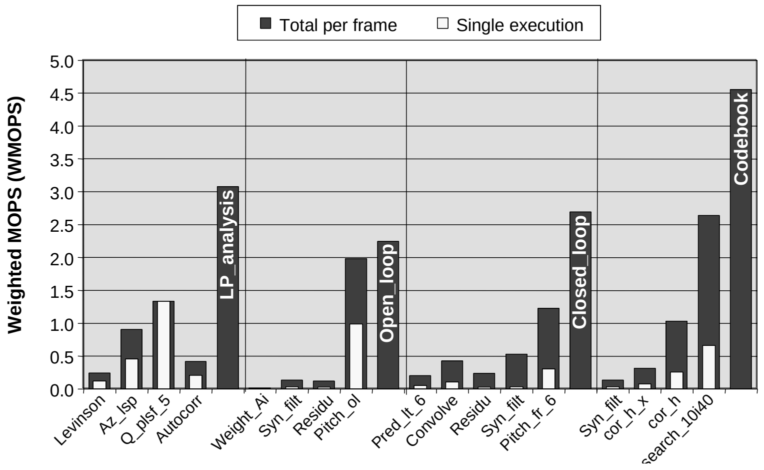

Figure 12: Relative Computational Complexity of the Encoder.

9.4 Architecture Exploration

As a first step in architecture exploration, a set of system components must be allocated. For the vocoder, a target architecture consisting of a digital signal processor (DSP) and a custom hardware co-processor was selected. The DSP was chosen due to the signal processing nature of the specification. Out of the DSPs available from Motorola, only the DSP56600 matched the 16-bit fixed-point requirements of the standard. A custom hardware co-processor assists the DSP. In case the performance of the DSP is not sufficient, computation can be accelerated by mapping it onto the co-processor.

In case of the vocoder, exploration started with a pure software solution in which both encoding and decoding tasks were executing concurrently on the DSP. The two concurrent tasks were scheduled dynamically in order to be able to adjust to changing arrival times between encoder and decoder.

(a)

	Cycles	Delay	*Constraint*
First subframe	1,146,387	19 ms	10 ms
Whole frame	2,801,488	47 ms	20 ms

(b)

	SW Cycles	HW Cycles	Delay	*Constraint*
First subframe	877,805	28,000	15 ms	10 ms
Whole frame	1,727,160	112,000	29 ms	20 ms

Figure 13: Vocoder Delay, (a) on DSP56600, (b) on DSP56600 + Custom HW.

The results of the performance estimation for the pure software solution are shown in Figure 13(a). Back-to-back encoding and decoding requires about 1.1 million and 2.8 million cycles on the DSP56600. This translates into a delay of 19 ms and 47 ms at the maximum clock frequency of 60 MHz specified for the DSP. As can be seen, these results are well beyond the constraints of 10 and 20 ms, respectively.

Thus, it became apparent that the DSP's peak processing rate of 60 MIPS would not be enough to satisfy the timing constraints. Therefore, the architecture was modified by moving the most critical part, the codebook search, into hardware. An estimation of the codebook search in hardware was performed by straightforward scheduling of the code into clock cycles. The results show that we can expect a 10x decrease in the number of clock cycles on top of the factor 1.6 improvement in clock speed.

The total estimated delays for this SW/HW solution are shown in Figure 13(b). The number of cycles for the software part is approximately cut in half. The hardware accelerated codebook search brings the total delays down to 15 ms and 29 ms. With the given accuracy, these worst-case results are within range of the

10/20 ms constraints. Therefore, this candidate architecture was chosen for further evaluation in the following implementation stages.

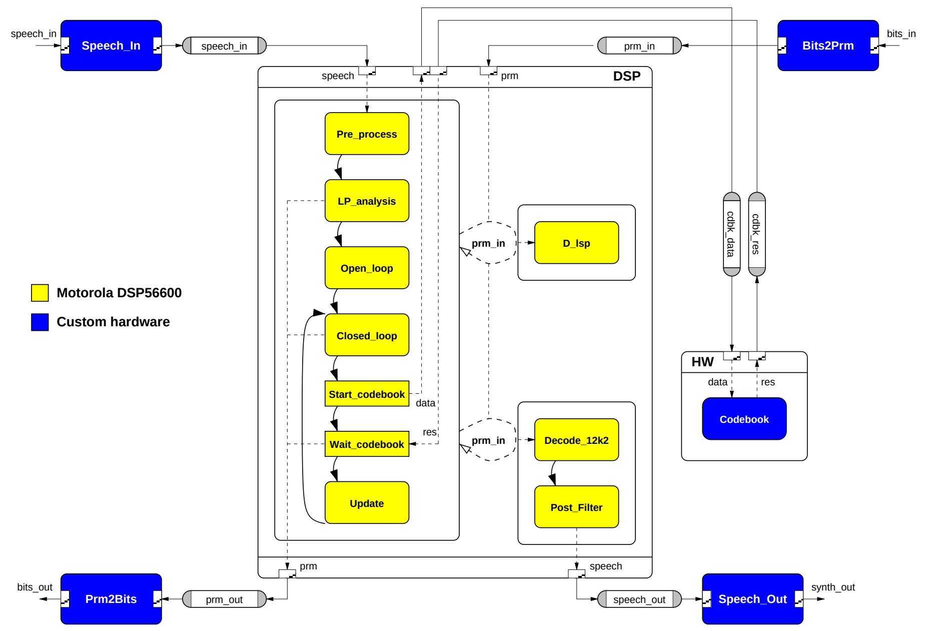

Figure 14: Vocoder Architecture Model.

9.5 Architecture Model

The resulting architecture model is shown in Figure 14. At the center of the architecture is the DSP56600 digital signal processor. The *DSP* runs encoding and decoding concurrently in a dynamic scheduling approach. The main loop of the application is formed by the encoding task reading incoming speech samples, processing them, and producing the encoded bit stream at the output. However, encoding is interrupted whenever a new packet arrives at the decoding side. Depending on the state of the decoding process, the corresponding decoding stage is executed and once the decoder has finished processing the incoming packet, control returns to the encoder.

The encoding task on the *DSP* is supported by the codebook search custom *HW* component. The encoder communicates with the codebook *HW* via message-passing channels in order to send data into the co-processor for processing and to receive the corresponding results.

In addition, the *DSP* is surrounded by four peripheral custom hardware components that handle I/O with the environment, pre-process incoming speech or bit streams, and perform the necessary framing.

9.6 Communication Synthesis

Next, synthesis of the communication between the DSP and the different hardware blocks was performed. A single system bus was chosen for all communication between the DSP and the five hardware components. The Motorola DSP defines its own fixed protocol for all communication over its built-in bus interface. Since the hardware components can be synthesized to support any protocol, the DSP56600 protocol was selected for the system bus.

Then, the inter-component message-passing communication was implemented over the DSP protocol. Bus interface FSMDs and bus drivers were synthesized for the application and protocol layers on the hardware and software side, respectively. Since the DSP is the only master on the bus, no arbitration was necessary. Synchronization was implemented by an interrupt-driven scheme between hardware and software.

9.7 Communication Model

The resulting communication model is shown in Figure 15. The five components are connected by the *address*, *data,* chip select (*MCS*), read (*nRD*) and write (*nWR*) wires of the bus. The *DSP56600* processor is the master on the bus. The *codebook HW* co-processor and the four peripheral hardware components are bus slaves, listening for transfers with matching addresses on the bus. In addition, hardware components can signal the DSP by raising interrupts in the processor, as exemplified by the connection from the codebook HW to the DSP's *intC* interrupt line.

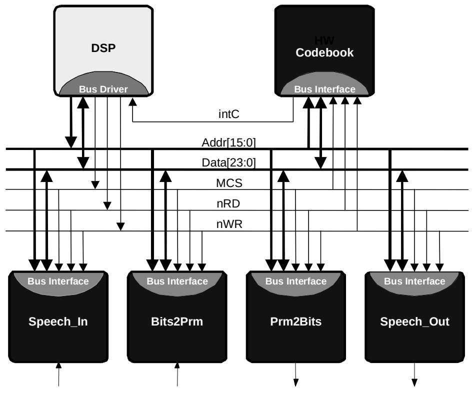

Figure 15: Vocoder Communication Model.

9.8 Backend

At the output of the system design process, the communication model is then fed into the backend tasks for implementation of the functionality on each component.

On the software side, the behaviors mapped onto the DSP were implemented by converting the SpecC hierarchy into C code and compiling the code into the DSP's instruction set. On the hardware side, the behaviors mapped onto the custom hardware co-processor and peripheral blocks were implemented down to RTL by performing high-level synthesis from the SpecC description. Finally, the interfaces on the software and hardware side were synthesized. Bus interface logic was generated as part of the custom hardware FSMDs. On the other hand, assembly code for the bus drivers and interrupt handlers was created and linked against the compiled DSP program.

The generated RTL code for custom hardware and interfaces was then further implemented and verified down to the gate-level by pushing it through logic synthesis (Synopsys DesignCompiler).

9.9 Implementation Model

The final SpecC implementation model of the vocoder design is shown in Figure 16. The implementation model performs a cycle-accurate co-simulation of hardware and software components communicating via bus wires.

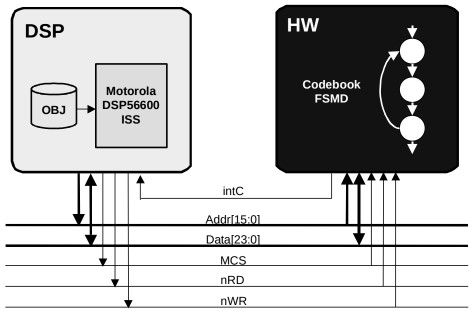

Figure 16: Vocoder Implementation Model.

The model for the DSP component runs an instruction-set simulation of the assembly code of the software generated in the backend, driving and sampling the bus wires according to the simulator's outputs and inputs.

The hardware component models are refined into FSMD models of the custom hardware RTL design. The state machines in the hardware execute the functionality and access the bus wires in a cycle-accurate manner.

With the DSP and custom hardware running at 60 Mhz and 100 MHz, respectively, the final results in Figure 17 show that the timing constraints for the transcoding delays are easily satisfied. The transcoding delays of 6 ms and 11 ms for the first subframe and the whole frame, respectively, even leave room for additional optimizations by reducing the clock frequency to lower power consumption, for example.

	SW Cycles	HW Cycles	ms	*Constraint*
First subframe	338,809	28,000	6.11	10 ms
Whole frame	530,351	112,000	10.71	20 ms

Figure 17: Results, Vocoder Performance.

In summary, the vocoder project was done by two non-expert students working part-time over a period of 6 months. Since the tools were not available at the time of the project, all design tasks were performed manually. With the availability of automated tools, the project time could have been shortened even further down to the 12 man-weeks spend on actual design.

All in all, the project demonstrated that just by following the well-defined steps of the SpecC methodology large productivity gains of 10x or more can already be achieved. With the help of upcoming design automation tools, the time-to-silicon will be reduced even further.

10 Summary and Conclusions

In this paper, we presented the SpecC system-level design methodology. The SpecC methodology defines four models and three transformations that bring an initial system specification down to an RTL-implementation.

The specification model is a purely functional description of the desired system functionality. It is free of any implementation details and there is no notion of time. The architecture model describes the component structure of the system architecture and orders computation based on estimated execution delays. The communication model refines communication into bus-functional component models. It is accurate in timing for both computation and communication. Finally, the implementation model is a cycle-accurate description of the system at the RTL/instruction-set level.

The SpecC design flow contains three major tasks: System synthesis consists of architecture exploration and communication synthesis, which map computation behaviors and communication channels in the specification onto components and busses of a system architecture, respectively. Then, in the backend, the components are implemented by synthesizing hardware, software and bus interfaces.

References

[1] A. Gerstlauer, S. Zhao, D. Gajski, A. Horak. *Design of a GSM Vocoder using SpecC Methodology*. UC Irvine, Technical Report ICS-TR-99-11, March 1999.

[2] D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, S. Zhao. *SpecC: Specification Language and Methodology*. Kluwer Academic Publishers, 2000.

[3] A. Gerstlauer, R. Dömer, J. Peng, D. Gajski. *System Design: A Practical Guide with SpecC*. Kluwer Academic Publishers, 2001.

[4] http://www.cecs.uci.edu/~specc/
[5] http://www.specc.org/