EDP 2002
9th IEEE/DATC Electronic Design Processes Workshop

SpecC Methodology for High-Level Modeling

Rainer Dömer
Daniel D. Gajski
Andreas Gerstlauer

Center for Embedded Computer Systems
Universitiy of California, Irvine, USA

Abstract

The key for managing the complexity of embedded system design is a well-defined methodology supported by a clearly structured system-level design language. The SpecC methodology described in this paper is based on the SpecC language and consists of a set of well-defined and unambiguous design models and a set of well-defined transformations that refine one model to the next. Given these models and transformations, the generic SpecC methodology can be customized to produce a system design framework that can be easily integrated with a given design flow and environment.

This paper describes four SpecC models at different levels of abstraction, namely at the specification, architecture, communication and implementation level. It also defines the refinement transformations between them, namely architecture exploration, communication synthesis, and software and hardware implementation. Both, the models and the transformations, are sufficiently formalized to allow automatic model refinement, synthesis, and verification.

1 Introduction

The generic SpecC methodology follows a top-down approach. A high-level, abstract specification of the intended system is step-wise refined down to a clock-cycle accurate implementation at the register transfer level (RTL). In other words, the SpecC framework will transform a pure behavioral system description into a fully structural netlist of system components such as processors, custom hardware units, memories, and intellectual property (IP) units, interconnected by system busses. Here, the behavioral input is basically plain C code, whereas the output is basically a hierarchical block diagram.

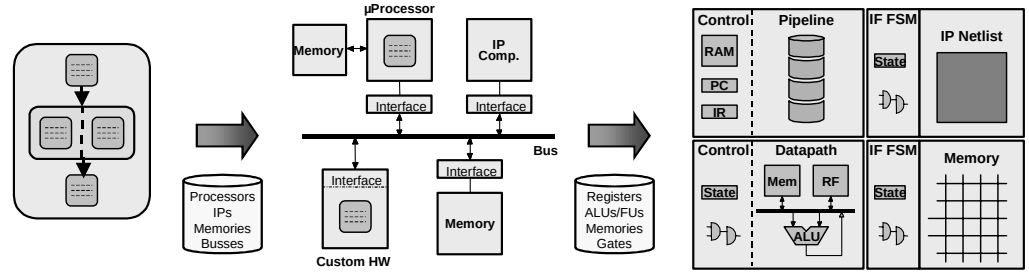

Figure 1: System-On-Chip Design.

Within the SpecC framework, the transformations on the models will be performed semi-automatically, that is, tedious model refinement tasks are performed by automated tools, whereas actual design decisions are made by the designers based on their experience and guided by data obtained from analysis tools such as profilers and static code analyzers.

To overcome the difficulties involved with the complexities of this system design task, the SpecC framework structures the synthesis process into a set of well-defined models that are refined gradually, in a straightforward and step by step manner.

Figure 1 shows this task divided into two major steps. First, the 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.

During the first step, the 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.

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

As described later, the first major step can be subdivided into smaller refinement steps. Chapter 2 covers this in detail and defines the four main models used in the SpecC design flow, namely the specification, architecture, communication and implementation models. In addition, the exact transformations used throughout the system refinement process are defined as well.

Since, in the real world, only very few systems are designed fully from scratch, the SpecC methodology puts emphasis on the reuse of legacy and intellectual property (IP) components. These can be easily integrated with the system description at any time by use of the "plug-and-play" feature of the SpecC language [3]. In addition, mixed levels of abstraction may be used freely in the same design model. This allows independent refinement of different parts of the system. Also, mixed levels of abstractions in the same design model may be used to speed up simulation by using low-level models only in the area of interest, while other parts are kept at the fast-executing high level.

1.1 SpecC Language

The SpecC methodology is based on the SpecC language [1]. The SpecC language was specifically developed to address the challenges of system-on-chip (SOC) design. It is based on ANSI-C and offers special extensions (keywords) to cover the needs of embedded designs including hardware. As such, the SpecC language provides a minimal, well-defined set of orthogonal constructs that precisely covers the concepts identified in embedded systems in a one-to-one fashion.

It should be emphasized that the concepts supported by the SpecC language cannot be extended by the user. While this intuitively sounds like a limitation, it is actually an important feature of the language. Having a fixed and well-defined set of constructs in the language is of crucial importance for the development of CAD tools because tools need to "understand" the semantics of each construct in order to be able to support the particular concept.

Note that this is contrast to other approaches, such as SystemC [4], where the extendability of the language (adding user-defined classes with special functionality) is part of the methodology. While such language extendability can be easily supported in simulation (after all, it is just C++ code to be compiled), this leads to significantly higher complexity. Moreover, support of such extensions by general synthesis and verification tools becomes impossible.

In that kind of methodology, the support of CAD tools will always be restricted to only a (typically small) *subset* of the language (remember VHDL!). As a result, the typical system specification may simulate nicely, but cannot be used with automated refinement and synthesis tools without tedious and error-prone manual modification by the user.

For more information on the SpecC language and its features and benefits, please refer to [2,3].

2 SpecC Design Flow

The SpecC design flow is based on four abstraction levels, namely *specification*, *architecture*, *communication*, and *implementation* level. As shown in Figure 2, the design starts with a specification model captured by the user based on algorithms of his/her choice.

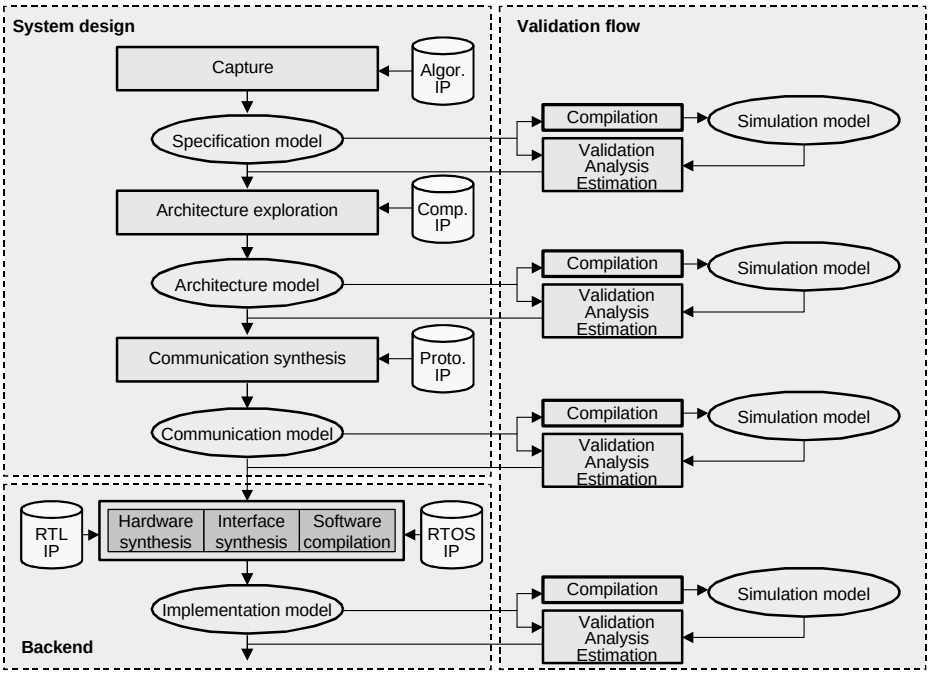

Figure 2: SpecC Design Flow.

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 micro-architecture 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.

At any abstraction level, the design models are represented by a corresponding description written in the SpecC language. Thus, all models are executable for validation through simulation and can reuse 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 and estimation. Also, the well-defined nature of the whole design process is the basis for rapid design space exploration through automatic model refinement and synthesis.

The following sections explain the different models and refinement steps of the SpecC design flow in detail. Using a simple design example, we will walk through the methodology step by step.

2.1 Specification Model

The SpecC design flow 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. As such, the specification model defines the basis for all exploration and synthesis. In particular, 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.

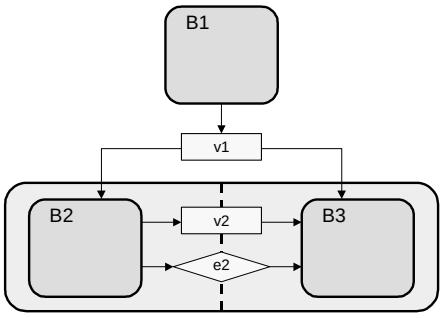

Figure 3: Specification Model Example.

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*.

2.2 Architecture Exploration

Architecture exploration derives the 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 the system architecture.

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

2.2.1 Behavior Partitioning

Behavior partitioning starts with 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.

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.

2.2.2 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. 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.

2.2.3 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.

In general, scheduling can be performed statically or dynamically. For space reasons, however, we simply assume a static scheduling for our example and refer to [3] for more details.

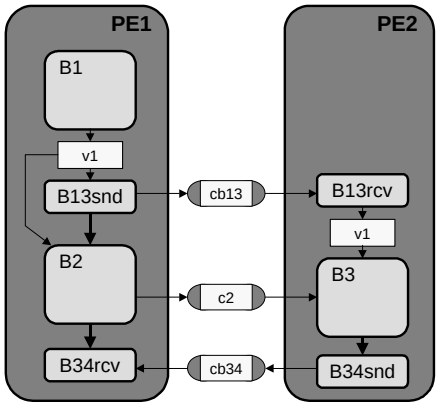

Figure 4: Architecture Model Example.

The final model of the design after scheduling is shown in Figure 4. At the top level, the model consists of the two PEs allocated for our example. The design is a parallel composition of component behaviors *PE1* and *PE2* communicating via message-passing channels *cb13*, *c2*, and *cb34*.

2.3 Architecture Model

After architecture exploration has been performed, the resulting model is therefore 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.

2.4 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*.

2.4.1 Channel Partitioning

The first step in communication synthesis 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.

2.4.2 Protocol Insertion

The next step is the insertion of actual bus protocols into the model. Here, 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.

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.

2.4.3 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 5, the single channel in our example is split into two halves that are moved into the component behaviors *PE1* and *PE2*, respectively. The variables representing the bus wires are exposed and the PEs are connected to the wires via corresponding ports.

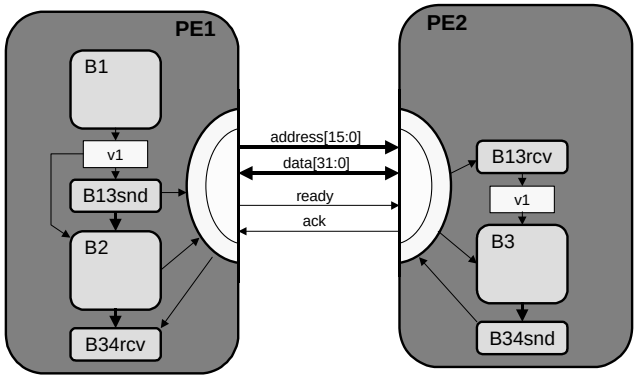

Figure 5: Communication Model Example.

2.5 Communication Model

The model after communication synthesis 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.

2.6 Backend

In the backend, the behavioral views of the components in the communication model are converted into structural descriptions of each component's micro-architecture. 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.

2.6.1 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.

2.6.2 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.

2.6.3 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.

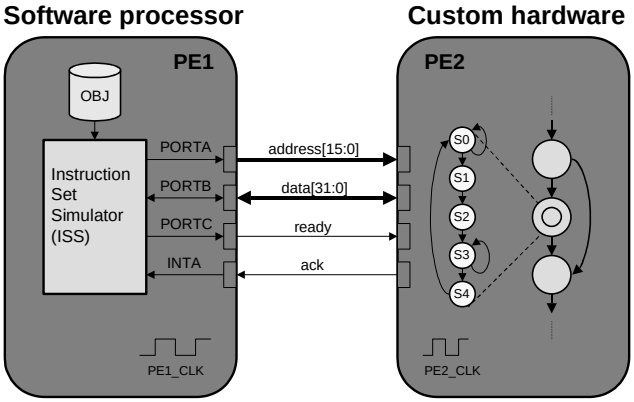

Figure 6: Implementation Model Example.

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

2.7 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 micro-architectures.

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 micro-architecture: 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.

3 Summary and Conclusions

In this paper, we presented the SpecC system-level design methodology. The customizable SpecC design flow defines four major models and three major transformations that bring an initial, abstract system specification down to a cycle-accurate 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.

The models and transformations are sufficiently formalized to allow automatic refinement, synthesis and verification. The SpecC design flow can also be easily customized to fit an existing design environment.

Based on the SpecC language, the SpecC framework seamlessly integrates IP components into the system at any level, and supports mixed levels of abstractions as well.

Today, the SpecC methodology is supported by the SpecC Technology Open Consortium (STOC) [5] and is backed by more than 30 companies and 30 universities worldwide. STOC promotes the SpecC technology by presentations and seminars. It also offers an Open Source reference implementation of the SpecC compiler and simulator [6] and further works on research and development to streamline the system design process by use of SpecC.

References

[1] R. Dömer, A. Gerstlauer, D. Gajski: *"SpecC Language Reference Manual, Version 1.0"*. SpecC Technology Open Consortium, *2001*.
[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] T. Grötker, S. Liao, G. Martin, S. Swan. *"System Design with SystemC"*. Kluwer Academic Publishers, 2002.
[5] http://www.specc.org/
[6] http://www.cecs.uci.edu/~specc/