SCE: System-on-Chip Environment

31

Gunar Schirner, Andreas Gerstlauer, and Rainer Dömer

Abstract

The constantly growing complexity of embedded systems is a challenge that drives the development of novel design automation techniques. System-level design can address these complexity challenges by raising the level of abstraction to jointly consider hardware and software as well as by integrating the design processes for heterogeneous system components. In this chapter, we present a comprehensive system-level design framework, the System-on-Chip Environment (SCE), which is based on the influential SpecC language and methodology. SCE implements a top-down digital system design flow based on a specify-explore-refine paradigm with support for heterogeneous target platforms consisting of custom hardware components, embedded software processors, and complex communication bus architectures. Starting from an abstract specification of the desired system, models at various levels of abstraction are automatically generated through successive stepwise refinement, ultimately resulting in a final pin- and cycle-accurate system implementation. The seamless integration of automatic model generation, estimation, and validation tools enables rapid Design Space Exploration (DSE) and efficient implementation of

G. Schirner (✉)
Department of Electrical and Computer Engineering, Northeastern University, Boston, MA, USA
e-mail: schirner@ece.neu.edu

A. Gerstlauer
Department of Electrical and Computer Engineering, The University of Texas at Austin, Austin, TX, USA
e-mail: gerstl@ece.utexas.edu

R. Dömer
Center for Embedded and Cyber-Physical Systems, Department of Electrical Engineering and Computer Science, The Henry Samueli School of Engineering, University of California, Irvine, CA, USA
e-mail: doemer@uci.edu

© Springer Science+Business Media Dordrecht 2017
S. Ha, J. Teich (eds.), *Handbook of Hardware/Software Codesign*,
DOI 10.1007/978-94-017-7267-9_31

Multi-Processor Systems-on-Chips (MPSoCs). This article provides an overview and highlights key aspects of the SCE framework from modeling and refinement to hardware and software synthesis. Using a cellphone-based example, our experimental results demonstrate the effectiveness of the SCE framework in terms of system-level exploration, hardware, and software synthesis.

Acronyms

API	Application Programming Interface
AST	Abstract Syntax Tree
BFM	Bus-Functional Model
BLM	Block-Level Model
CE	Communication Element
DB	Database
DCT	Discrete Cosine Transform
DSE	Design Space Exploration
ESL	Electronic System Level
FCFS	First-Come First-Serve
FIFO	First-In First-Out
FPGA	Field-Programmable Gate Array
GUI	Graphical User Interface
HAL	Hardware Abstraction Layer
HDS	Hardware-Dependent Software
HLS	High-Level Synthesis
HW	Hardware
IDE	Integrated Development Environment
IP	Intellectual Property
ISS	Instruction-Set Simulator
MAC	Media Access Control
MIPS	Million Instructions Per Second
MoC	Model of Computation
MPSoC	Multi-Processor System-on-Chip
OOO PDES	Out-of-Order Parallel Discrete Event Simulation
OS	Operating System
PDES	Parallel Discrete Event Simulation
PE	Processing Element
PIC	Programmable Interrupt Controller
PSM	Program State Machine
RTL	Register Transfer Level
RTOS	Real-Time Operating System
SCE	System-on-Chip Environment
SLDL	System-Level Description Language
SW	Software
TLM	Transaction-Level Model

Contents

31.1 Introduction

Designing modern Multi-Processor Systems-on-Chips (MPSoCs) becomes increasingly difficult. Challenges arise from both an increasing heterogeneity of the execution platform to meet stringent performance and power requirements, as well as from the growing application complexity demanding to integrate more, increasingly interrelated functions. Today's MPSoCs are highly heterogeneous compositions of general purpose processors, digital signal processors, graphics processors, and custom accelerators, all connected through flexible and heterogeneous interconnect systems. Designing and programming such platforms are a tremendous challenge due to heterogeneity in programming paradigms, differences in exposed parallelism, and tool suite compositions. The productivity gap between design capability and the potential in chip complexity/capacity is increasing [15]. Traditional approaches of manual implementation are tedious and error-prone as well as too time consuming to meet the shortened time-to-market demands.

One key aspect to increase productivity is to raise the level of abstraction for system design to an algorithmic level, irrespective of a later Hardware (HW)/Software (SW) split, hiding the complexity of low-level implementation details. Moving to the system level of abstraction reduces the complexity during development, enabling designers to focus on important algorithmic properties and design decisions without being overwhelmed by the burden of low-level implementation issues. However, raising the level of design abstraction requires tool suites that are vertically integrated across all levels so as to enable a seamless codesign of software and hardware.

In this chapter, we present the System-on-Chip Environment (SCE), a vertically integrated digital system design framework based on the SpecC language and methodology [18]. SCE realizes a top-down refinement-based system design flow with support of heterogeneous target platforms consisting of custom hardware components, embedded software processors, dedicated Intellectual Property (IP) blocks, and complex communication bus architectures.

Starting off with a high-level, abstract, formal, and sound parallel programming model in SpecC language to capture the desired application behavior, the designer can define various architecture and mapping alternatives. SCE then generates Transaction-Level Models (TLMs) that realize the architecture and mapping decisions. The generated TLMs allow for detailed, simulation-based validation and performance analysis. After identifying suitable architecture and mapping candidates for an application, SCE's back-end synthesis aids in generating a cycle- and pin-accurate implementation as an interconnected set of synthesized hardware and software components down to final RTL descriptions and binary code images.

This chapter first introduces relevant related work in Sect. 31.2. It then provides a general overview of the SCE design flow in Sect. 31.3, followed by highlighting key features of the flow in more detail. Section 31.4 covers model validation with performance estimation and simulation. Section 31.5 then describes the successive model refinement for system-level architecture and communication aspects. Next, Sect. 31.6 describes the SCE software synthesis highlighting target optimization potentials, and Sect. 31.7 covers the hardware synthesis capabilities. Finally, Sect. 31.8 demonstrates the benefits of SCE with experimental results and Sect. 31.9 concludes this chapter.

31.2 Related Work

Supporting the design process has been the aim of significant research efforts with a wide range of approaches. To name a few, they range from high-level analysis and synthesis approaches that are based on specialized models of computation. Examples include POLIS [1] (Codesign Finite State Machine), DESCARTES [32] (ADF and an extended SDF), and Cortadella et al. [9] (petri nets). Integrated Development Environments (IDEs), at another end of the spectrum, typically provide limited automation support but aim to simplify manual development (e.g., Eclipse IDE [16] with its wide range of plug-in modules). Several comprehensive Electronic System Level (ESL) synthesis methodologies and tools [20] have been developed for the system-level design of heterogeneous MPSoCs. Examples include Deadalus [30], Koski [27], Metropolis [2], PeaCE/HoPES [25], SystemCoDesigner [28], and OSSS [24].

Abstract models are an important means for early prototyping and performance estimation. System-Level Description Languages (SLDLs), such as SystemC [23] and SpecC [18], are often used for modeling of systems. At lower levels, virtual platforms allow for a detailed analysis of the system before availability of real hardware, often revealing details not available on the target [26]. While these approaches focus on modeling, simulation, and validation, they do not offer an integrated solution to generate the final implementation.

31.3 Design Flow Overview

Figure 31.1 outlines the design flow realized by the System-on-Chip Environment (SCE) [13]. The SCE flow focuses on two major steps: refinement-based system-level design space exploration in the front-end and software/hardware back-end synthesis. In the front-end exploration phase, the input specification is refined into

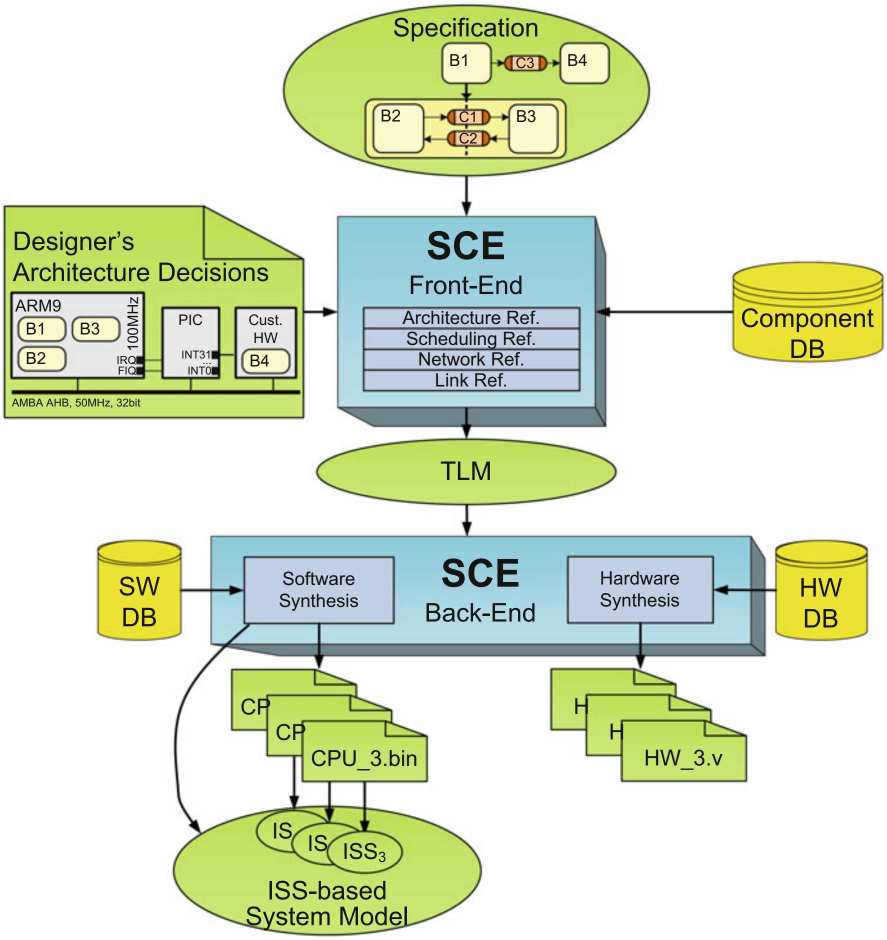


Fig. 31.1 Overall system design flow in the System-on-Chip Environment (SCE)

a TLM realizing the designer's architecture and mapping decisions for detailed performance estimation and early validation. The TLM then serves as an input for back-end synthesis, including both *software synthesis*, which automatically generates the final target software implementations for each system processor, and *hardware synthesis*, which generates the Register Transfer Level (RTL) for each custom hardware component. Both software and hardware synthesis generate matching communication stacks to realize the application distributed across the heterogeneous processing platform.

The input to the system design flow is a *specification model* containing the application captured in SpecC [18]. The specification is an abstract, parallel, platform-agnostic description of application algorithms. SpecC allows capturing a wide range of application models with an arbitrary serial-parallel composition of behaviors that communicate through abstract communication primitives for synchronous or asynchronous message passing, shared variable access, or event transfers (see Sect. 31.3.1 for an overview).

A second type of input contains the *designer's architecture decisions* including platform definition and specification mapping. For this, the designer defines the number and type of processors in the system and the topology of their communication and connectivity. In addition, the designer defines the mapping of application computation and communication onto the target platform. This includes decisions on the target software architecture (e.g., how to realize multitasking) and communication refinement decisions, such as the routing of channels over busses and the definition of essential communication parameters for each channel. For example, the user can select the synchronization scheme, such as polling or interrupt-based synchronization.

Based on the *specification model*, and the designers' *architecture decisions*, SCE then automatically generates a Transaction-Level Model (TLM) that realizes these decisions. The generation process is aided by a rich *component database* containing Processing Elements (PEs) (e.g., processors, DSPs), Communication Elements (CEs) (e.g., bridges, routers), memory components, and interconnects. In the generation process, the selected component models are instantiated and connected to create the envisioned platform. On top of this, the application (as defined in the specification) is distributed to the PEs and CEs according to the mapping decisions. Communication between PEs is refined from the standardized abstract channels in the specification down to distributed set of channels realizing the specified communication semantics on the selected platform. In order to deal with the tremendous complexity in the front-end exploration, the TLM generation is subdivided into four successive refinement steps. Each step focuses on a particular architecture aspect and by realizing, i.e., *refining*, that aspect, uncovers the next set of architecture decisions to be made (see Sect. 31.3.3).

Overall, the generated TLM captures a model of the application mapped to the envision platform realizing the architecture decisions. The application together with generated communication stacks executes behaviorally with timing back annotation on top of the timed abstract models of PEs, CEs, memories, and interconnects.

Section 31.5 illustrates the modeling in more detail. The TLM supports rapid and accurate system simulations, providing the basis for exploration, performance analysis, and debugging.

Once the designer has identified a suitable platform, the TLM also serves as input for the back-end synthesis of both software and hardware. *Software synthesis* produces a final binary image for each processor in the platform. The binary includes the application code, all drivers for communication in a heterogeneous system, as well as an off-the-shelf Real-Time Operating System (RTOS), if selected. The produced binaries can directly execute on the target processor(s) of the final hardware. For early binary validation, before availability of the real hardware, Instruction-Set Simulator (ISS)-based system models (i.e., virtual platforms) can be used. Section 31.6 discusses more details about SW synthesis. *Hardware synthesis* produces RTL for the mapped portion of the application code and the necessary communication stacks. This RTL can then be synthesized further down to a gate-level netlist and final physical realization using standard logic and back-end synthesis flows. More details about HW synthesis are described in Sect. 31.6.

The SCE flow vertically integrates from specification down to implementation through a set of consistent models, all captured in the SpecC language. This vertical integration lends itself for development of plugins at varying abstraction levels. One such example is Algo2Spec [41, 42] which automatically synthesizes a Simulink algorithm model into a SpecC specification. With this, Algo2Spec creates an extended codesign flow offering additional opportunities for algorithm designers to explore the platform suitability of algorithms and to tune algorithms to better match platform requirements (e.g., in terms of parallelism).

31.3.1 SpecC Language and PSM Model of Computation

The SCE framework is based on and closely integrated with the SpecC language and methodology [18]. It should be emphasized that this is a unique setting in which the tools, the methodology, *and* the language have been specifically created and designed together to address the needs of designing digital systems consisting of both hardware and software. In fact, the SpecC language [12] has been specifically designed to support the essential requirements for describing embedded system models at different abstraction levels. In particular, SpecC features explicit constructs and keywords for behavioral and structural hierarchy, concurrency and pipelining, synchronization and communication, exception handling, timing, and explicit state transitions. Moreover, SpecC precisely covers these requirements in an orthogonal and thus minimal manner [19].

The benefit of SpecC's language approach (in contrast to the library approach of SystemC) is the ability to parse SpecC models and unambiguously recognize the captured system modeling features. Notably, SpecC covers multiple abstraction levels, from the abstract specification model in which only functionality and design constraints are represented down to the cycle- and pin-accurate implementation models at RTL abstraction. In contrast to commercial multi-language tools in both the hardware and software domains, the *single language* approach is a benefit since only one compiler and run-time engine must be built and maintained. In other words, the SCE framework can rely on a single-core data structure to represent the model from the beginning to the end of the design flow.

Such a homogeneous methodology does not suffer from language interfacing problems or cumbersome translations between languages with different semantics. Instead, all models are consistent, and one set of tools can be used for all models at all stages. Also, refinement tasks are merely transformations from one model into a more detailed one specified with the same language. Using a single language throughout the design process is beneficial for reuse of IP as well. Design models from the component library can be reused in the system without modification ("plug-and-play") and a new design can be inserted immediately as a library component.

The SpecC language used in the SCE design flow is based on the Program State Machine (PSM) Model of Computation (MoC). Computation and communication are separately captured using distinct language constructs. This separation enables an automatic refinement for mapping of computation to separate processing elements and establishing the communication between PEs. Computation is captured in the form of so-called behaviors, and communication is expressed in channels.

Figure 31.1 contains a graphical representation of a simple *specification model*. Boxes with rounded corners (*B1-B4*) symbolize behaviors. Each leaf behavior basically contains ANSI-C code, which is omitted for brevity. Behaviors can also be composed hierarchically to allow for complex structures. They can be behaviorally composed to execute in sequence, parallel, pipelined, or as state machine [19].

Behaviors are statically connected and communicate through direct point-to-point channels (*C1*, *C2*, *C3*). These channels are selected from a feature-rich set of standardized channel types, which allow for a wide range of communication mechanisms similar to what is found in an operating system. Communication primitives include synchronous and asynchronous message passing, blocking and non-blocking communication (e.g., FIFO), as well as synchronization only (e.g., semaphore, mutex, barrier).

31.3.2 Target Platform Description

The designer's target architecture decisions, as shown on the left of Fig. 31.1, describe the digital target platform. These architecture decisions include the allocation of PEs such as processors and HW components and the mapping of behaviors to PEs. The example in Fig. 31.1 shows the allocation of an ARM9 processor and one custom hardware component. The behaviors *B1*, *B2*, and *B3* are mapped to the processor. These behaviors are later wrapped into tasks, and the designer can select important task parameters, such as scheduling policies and priorities.

More generally, SCE supports distributed Multi-Processor System-on-Chip (MPSoC) target architectures as conceptually illustrated in Fig. 31.2. Target

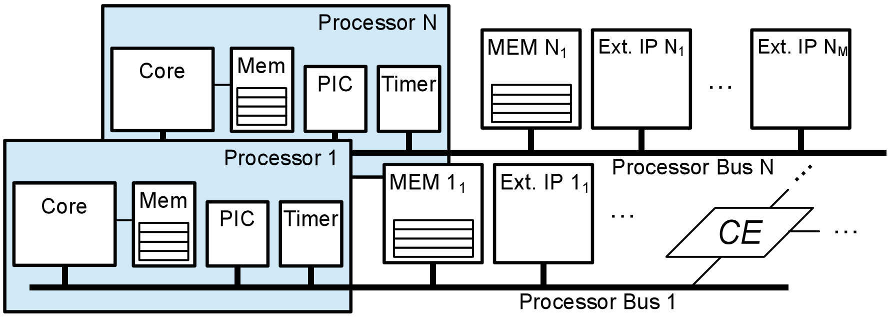


Fig. 31.2 Generic MPSoC platform targeted in SCE

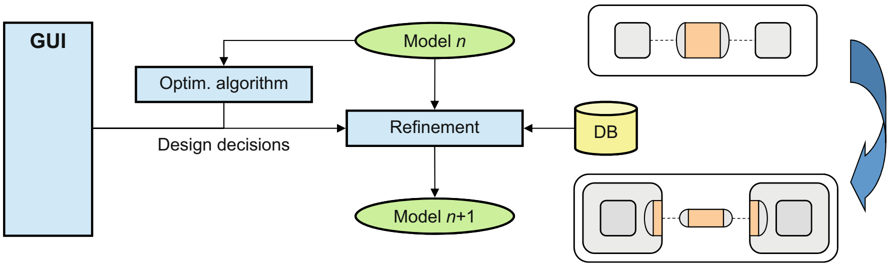


Fig. 31.3 General stepwise refinement approach in SCE

platforms can consist of a set of processors, where each processor is connected to a processor specific main bus. We assume that each processor has an associated memory, which stores the execution binaries and local variables. Additionally, we associate a customizable Programmable Interrupt Controller (PIC) and a timer with each processor. Each processor communicates with external memory (holding globally shared variables) and with hardware blocks over the processor main bus. A processor also can communicate with other processors and external IPs or memories connected to other busses through one or more Communication Elements (CEs), such as a bridges or a routers. In extension to what is shown in Fig. 31.2, we assume that the busses may be arranged as a hierarchy of busses.

31.3.3 Stepwise Refinement

The SCE framework implements a top-down design flow. SCE vertically integrates from an abstract behavioral specification down to a detailed implementation through a series of successive refinement steps. The general principle of this stepwise refinement is outlined in Fig. 31.3.

SCE's stepwise refinement separates decision-making and model refinement. Design *decisions* are primarily made by the user, entered through a GUI, or automatically determined by an optimization algorithm. Conversely, the realization of the design decision, i.e., the *refinement*, is automated. For this, a dedicated refinement tool at a given abstraction level reads the input model ($Model_n$) and refines it following the given architecture decisions, producing the refined model ($Model_{n+1}$) which then exhibits additional implementation details that realize and represent these decisions. Each refinement tool utilizes a Database (DB) of components (such as a processor model, operating system model, etc.) to implement the decisions (such as behavior mapping or task mapping). With this separation, the tedious and error-prone part of model refinement is automated.

The SCE-internal refinement tools exchange design models in the form of a Syntax Independent Representation (SIR) [10, 39], a binary representation of a SpecC model. The SpecC compiler converts between the SpecC source code and its binary representation (the SIR). In addition, the SpecC compiler suite provides a rich Application Programming Interface (API) for convenient model transformation (such as traversing the model hierarchy, adding behaviors, or manipulating their port connectivity). Standardizing model manipulation dramatically simplifies building a refinement tool.

Four refinement levels are distinguished within SCE's front-end (Fig. 31.1). Each refinement (i.e., model transformation from $Model_n$ to $Model_{n+1}$) realizes one set of orthogonal architecture/mapping decisions and by that uncovers the set of design decisions that have to be made to guide the next refinement step (which would then create $Model_{n+2}$).

Starting from the *Specification Model*, *Architecture Refinement* realizes the mapping of behaviors to PEs and adds the necessary synchronization logic to maintain the specified execution order. Mapping parallel executing behaviors to the same PE necessitates dynamic scheduling. This prompts the user to specify scheduling parameters and priorities. *Scheduling refinement* then implements these decisions, integrating the selected RTOS model into the design, converting behaviors into tasks and setting their scheduling parameters.

The next steps then focus on communication refinement. The user is prompted to define the overall interconnect topology (bus interfaces, interconnects, communication elements). *Network refinement* realizes these decisions by inserting appropriate models from the database and ensuring proper mapping. Given the interconnect network, the final decision is mapping of channels to the interconnect structure, defining addresses and synchronization principles. *Link refinement* realizes these decisions generating the final TLM or a more detailed pin- and cycle-accurate Bus-Functional Model (BFM). Section 31.5 covers the individual refinement steps and resulting models in more detail.

31.4 Model Validation

As mentioned above, the design models used throughout the SCE flow are all represented as executable models in the SpecC language [12], regardless of the abstraction level they are specified at or refined to (The only exception is the final design model at RTL abstraction which, in addition to SpecC, can also be exported in VHDL or Verilog HDLs for hardware synthesis and the program code generated

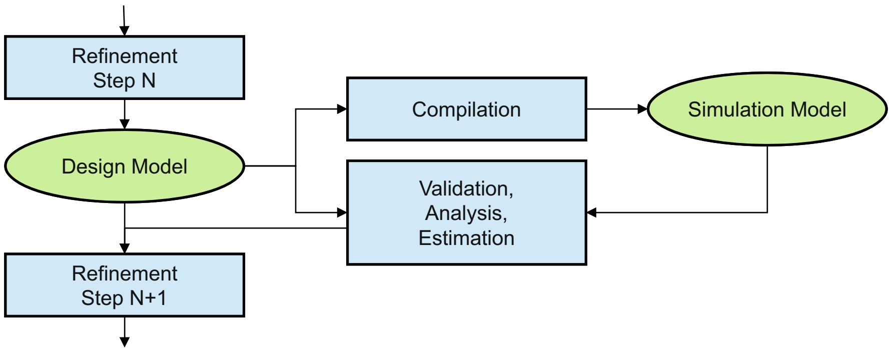


Fig. 31.4 SCE validation flow at any abstraction level: model compilation and simulation for validation, analysis, and performance metrics estimation

in ANSI-C for software synthesis.). Therefore, all models are readily executable for functional validation and evaluation through simulation. In addition, the formal nature of the models and the availability of a compiler with a comprehensive internal representation and corresponding API [10] also enables the application of formal methods for model analysis, verification, and estimation.

At any stage, as shown in Fig. 31.4, the design model together with its testbench can be fed into the compiler to generate an executable simulation model for dynamic validation (see Sect. 31.4.1 below). Alternatively, *static analysis* can be applied to the model for estimation and formal verification purposes. In contrast to dynamic simulation, where the model is executed and actual input data and specific dynamic behavior is observed, static analysis relies solely on the information stored in the SLDL source code. Here, a compiler front end reads the model code, analyzes it, and builds an Abstract Syntax Tree (AST) with control flow and typed symbol information. Based on that, static information can be extracted that is generally valid (and not just valid for a specific input). For example, where dynamic simulation shows that a model works fine for a given test set, static model analysis may prove the existence of potential deadlocks (or their absence) for *any* data input and thus show a stronger property.

In general and in practice, both static and dynamic methods are typically used together for the analysis and estimation of model metrics, in order to guide the following design decisions so that the application's requirements and goals are achieved as needed.

Note that the tasks performed in the SCE validation flow are virtually identical at any abstraction level and therefore can be performed by the same set of tools [11]. At the same time, the system designer can rely on the same methods for model validation and performance estimation, which significantly eases the learning curve for system design in SCE. Last but not least, any model can be compared against its predecessor or the golden specification model such that functional correctness and meeting of critical design goals are maintained.

31.4.1 Simulation

In system-level design, simulation is the most common form of design validation. In contrast to static analysis, simulation is dynamic and requires the design model to be executable.

In SCE, simulation is performed in two steps. First, the design model is compiled into a corresponding simulation model. More specifically, the SpecC compiler takes the design model, together with a corresponding testbench, and generates an executable program that is linked against the simulation library. The simulation library implements the execution semantics of the simulation. In particular, it maintains an event queue, advances the simulation time, and also takes care of concurrent execution and required synchronization. The generated simulation model can be run on a host computer, simulating the execution of the corresponding model. Typically, the testbench included in the simulation model supplies test vectors, checks the computed output values, and reports any problems to the user. If problems occur, a debugger can be used to set break points, interrupt the simulation, and inspect intermediate values, so that the system designer can locate and fix the errors in the model.

It should be noted that there is generally a trade-off between the run time and the accuracy of the simulation. For example, compared to the initial specification model, the refined communication model will need longer time for execution, because it may perform the communication accurately in a clock-cycle manner.

Recently, the SCE compiler and simulator have been extended to support the parallel execution of design models on multi- and many-core hosts [5, 7]. Parallel simulation, known as Parallel Discrete Event Simulation (PDES) [17] in the literature, exploits the parallelism exposed in the design model for parallel execution and thus can gain up to an order of magnitude increased simulation speed. This topic is discussed in detail in the ▸ Chap. 17, "Parallel Simulation" in this book. The advanced PDES technique specifically designed for and implemented in SCE is called Out-of-Order Parallel Discrete Event Simulation (OOO PDES) [6, 8].

31.4.2 Profiling

SCE also includes profiling tools to obtain feedback about design quality metrics. Based on a combination of static and dynamic analysis, a retargetable profiler measures a variety of metrics across various levels of abstraction [4].

Initial dynamic profiling derives design characteristics through simulation of the input model. The system designer chooses a set of target PEs, CEs, and busses from the database, and the tool then combines the obtained profiles with the characteristics of the selected components. Thus, SCE profiling is retargetable for static estimation of complete system designs in linear time without the need for time consuming re-simulation or re-profiling.

The profiling results can also be back-annotated into the model through refinement. By simulating the refined model, accurate feedback about implementation effects can then be obtained before entering the next design stage. Since the system is only simulated once during the exploration process, the approach is fast yet accurate enough to make high-level decisions, since both static and dynamic effects are captured. Furthermore, the profiler supports multi-level, multi-metric estimation by providing relevant design quality metrics for each stage of the design process [3]. Therefore, profiling guides the user in the design process and enables rapid and early Design Space Exploration (DSE).

31.4.3 Estimation

The task of estimation is to calculate quality metrics from a design model. Although these metrics should be accurate, the main emphasis of estimation is to deliver these values quickly.

In SCE, estimated quality metrics are, for example, used for the task of architecture exploration. For instance, the trade-off between a software or a hardware solution for each behavior in the design model requires metrics for performance and cost. More specifically, the execution time and the area of each behavior are estimated for a potential hardware implementation. Also, the execution time, code size, and data size will be determined for a potential implementation in software, for each allocated processor. In addition, metrics, such as bit width and throughput, need to be determined for all channel and bus models, since these are needed for the task of communication synthesis. All these estimation results are annotated in the design model at the particular behaviors and channels. Thus, they are fed back into the synthesis flow so that this data is immediately available when needed by the synthesis algorithms.

Estimation is typically performed in form of static analysis of the design model. However, by use of profiling, estimation data can also be obtained dynamically during simulation. In SCE, profiling is used, for example, to count the execution frequency of each behavior. Based on these counter values, branching probabilities are determined, for example, for the conditional transitions in FSM behaviors. These branching probabilities are then used to estimate the average execution time for such behaviors.

Recently, the SCE profiling and estimation tools have been extended to support energy dissipation and power consumption for processing elements [33, 34]. which is essential for battery-powered mobile embedded systems. Dedicated power monitors can be inserted into the design model to observe and monitor power dissipation during the simulation. The system designer can then take these power characteristics into account when making design decisions.

31.5 Modeling and Refinement

Modeling and refinement are at the core of the SCE framework. The SCE exploration front end follows a successive, stepwise, and layer-based refinement process that employs a series of consecutive implementation and optimization passes as

Table 31.1 Summary of SCE system-level refinement steps

Ref.		Design decisions	Modeling layers
Comp.	Arch.	• Number and type of PEs and memories • Behaviors/variable to PE/memory mapping	PEs and memories (*App.*) Basic channels and memory i/fs
	Sched.	• Static behavior execution order • Dynamic scheduling policy and parameters	Operating system (*OS*) Basic channels and memory i/fs
Comm.	Net.	• Number and type of CEs and busses • Connectivity and channel routing	PE drivers (*HAL*) CE/PE transfers (*Pres./Net.*)
	Link	• Bus addressing and bus transfer modes • Bus arbitration and synchronization	PE hardware (*HW*) Bus transactions (*Link/MAC*)

described in Sect. 31.3.3. Each refinement step (see summary in Table 31.1) generates a SpecC-based Transaction-Level Model (TLM) with a level of abstraction appropriate to the refinement step. Each TLM captures an increasing amount of layered implementation detail and thus covers a different point in the simulation speed versus accuracy trade-off. These TLMs allow validation of the generated implementations while simultaneously serving as input to the back-end synthesis (see Sects. 31.7 and 31.6).

Following the general separation of computation and communication [18, 20], *computation design* is performed before *communication design* in SCE. Computation design is split into two smaller steps named *architecture refinement* and *scheduling refinement*. Communication design, similarly, is split into *network refinement* and *link refinement* (see Sect. 31.3.3). The output of the computation design, an intermediate scheduled architecture model, allows validation of the main computation mapping while exposing all required inter-PE interactions as input for further communication refinement. The final TLMs as an output of the communication design combines all computation and communication aspects of a system design in the form of PEs (modeled as SpecC behaviors) connected via busses and CEs (modeled as TLM channels and behaviors, respectively). Table 31.1 summarizes the different refinement steps including decisions made and modeling layers inserted in each step. In the following sections, we describe the computation and communication modeling and refinement steps in more detail.

31.5.1 Computation Modeling and Refinement

On the computation side, refinement transforms application behaviors in the specification into tasks and blocks partitioned and scheduled to execute in corresponding PE implementations, which are generated through a series of layer-based refinement steps. Functionality is organized into layers according to inherent conceptual dependencies. Generated output models are equally organized into layers of increasing detail. Each individual refinement step thereby introduces an additional layer of modeling and implementation detail.

SCE generally follows a host-compiled layering and modeling flow as described in ▶ Chap. 19, "Host-Compiled Simulation". For software PEs, complex, SpecC-based Operating System (OS) and processor models are automatically generated [38]. Within SCE, these models have been extended to support simulation and code generation for state-of-the-art single- and multi-core OSs and processors [31] (also discussed in ▶ Chap. 19, "Host-Compiled Simulation"). Other hardware PEs are generated as simplified variants of complete processor models that do not include all layers.

Figure 31.5 depicts the final multi-core processor model generated by SCE for the example of a dual-core ARM PE. The innermost application layer generated during architecture refinement encapsulates the specification behaviors mapped onto the processor. During scheduling refinement, these behaviors are converted into user tasks running on top of scheduling services provided by an OS layer and OS model (realized as a SpecC OS channel). As described in ▶ Chap. 19, "Host-Compiled Simulation", the OS channel provides typical services for OS and task management, synchronization, inter-process communication and timing via a canonical OS API that is later translated into real OS calls during back-end software synthesis (see Sect. 31.6). In the process of converting specification behaviors into OS tasks, the code is also back-annotated with estimated delays as described in more detail in ▶ Chap. 19, "Host-Compiled Simulation".

In addition to basic OS and task services, the OS layer also provides high-level communication functionality for sending and receiving inter-processor application-level messages via an underlying Hardware Abstraction Layer (HAL). During architecture and scheduling refinement, only basic models for channel adapters, drivers, interrupt tasks, and interrupt handlers are inserted as templates into the OS layer and HAL. These templates are later filled with actual code during link refinement, where HALs with pre-written, canonical models for Media Access Control (MAC) and bus/TLM interfaces are taken out of SCE's component database. Together, the OS and HAL ultimately provide and realize timing-accurate models of communication protocol stacks that transform application-level message channels all the way down to corresponding transaction-level bus accesses plus interrupt-driven or polling-based synchronizations, if required. See Sect. 31.5.2 for more details.

Finally, the HAL is encapsulated into a Hardware (HW) layer that models external bus communication via a bus channel. The HW layer of the processor model also emulates monitoring of processor interrupt signals and associated processor exceptions to model a general, timing-accurate multi-core interrupt handling logic and chain. From the hardware side, core-specific interrupt requests are generated by a generic multi-core interrupt controller (GIC) model, which manages the distribution and dispatch of interrupt signals to processor cores. The HW layer in turn contains core-specific interrupt detection logic (shown as interrupt interfaces in Fig. 31.5) that triggers interrupt execution in the HAL. To emulate processor suspension, hardware-triggered interrupts are modeled as special, high-priority interrupt handlers associated with each interrupt source. Thus, when an interrupt is detected by a core's interrupt interface, the HAL will notify the OS model (via a

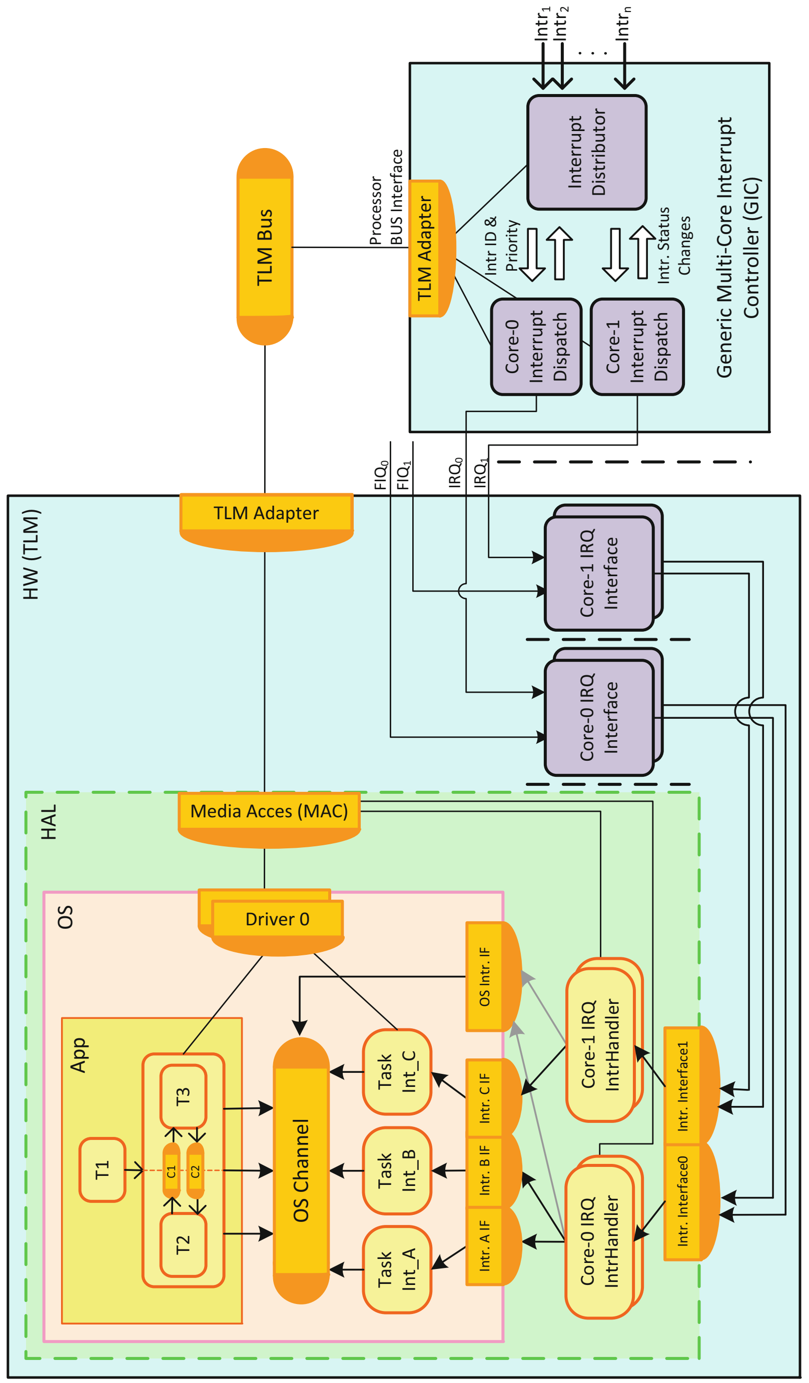


Fig. 31.5 General layer-based multi-core processor model

special OS interrupt interface) to preempt and switch execution to the corresponding interrupt handler in the HAL. The interrupt handler, as generated during link refinement, can then in turn communicate with the GIC and, through associated OS layer interfaces, trigger corresponding secondary interrupt-specific OS tasks (as shown for the example of interrupts A, B, and C in Fig. 31.5).

Overall, the application, OS, HAL, and HW layers all combined constitute the host-compiled SpecC processor model. This processor model is then integrated into a standard TLM backplane for simulation in the context of an overall multiprocessor system environment.

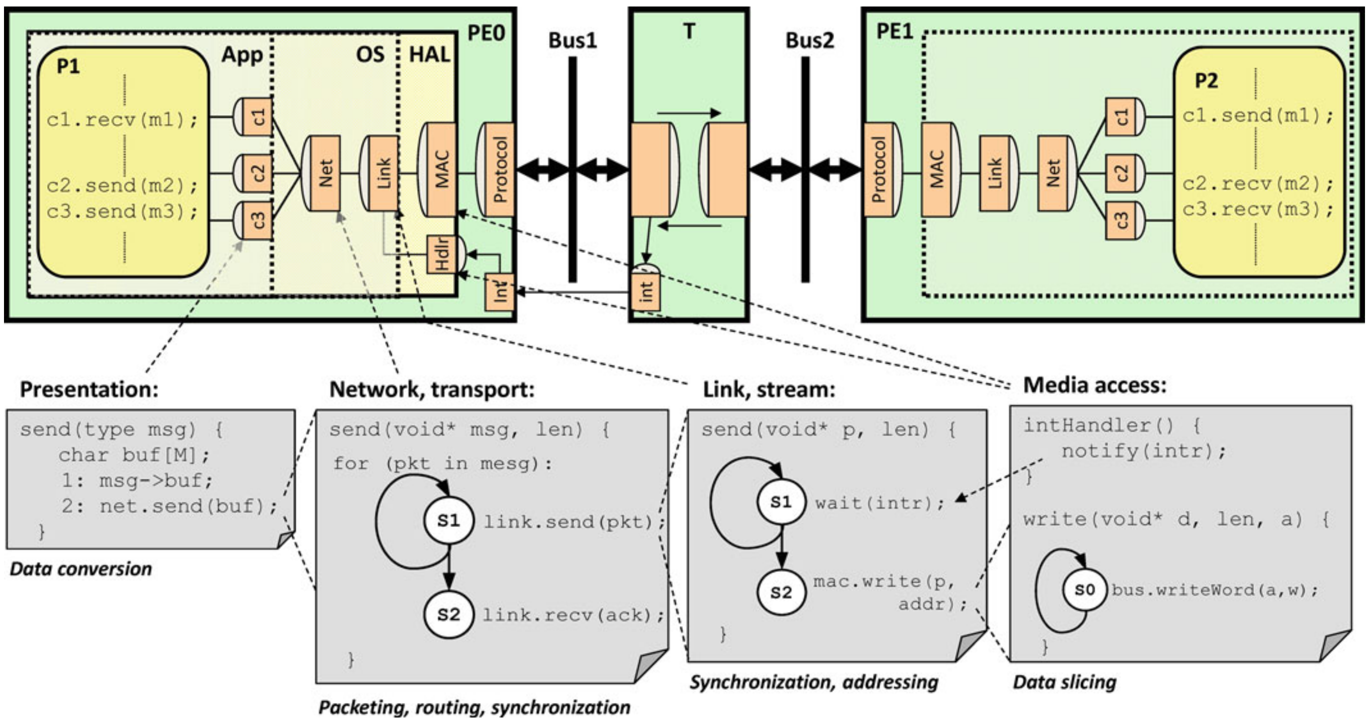
31.5.2 Communication Modeling and Refinement

In the model generated after architecture and scheduling refinement, PEs still communicate via high-level primitives at the message-passing level, where a PE's HAL and HW layers are still left out. Network and link refinement then transform such abstract application-level communication channels all the way down to transactions over busses or other (shared) communication media. In the process, HAL and HW layers are added to PEs, and optimized code is generated for drivers, interrupt handlers, and bus transactors inserted into software and hardware PEs, respectively. All driver, interrupt handling, and transactor code is back-annotated with estimated delay information to provide an overall timing-accurate simulation of communication overheads.

Similar to the computation side, communication modeling and refinement follow a layer-based organization adapted and derived from the ISO/OSI 7-layer model [21]. At the output, low-level TLMs that include protocol stacks and inter-PE communication at varying levels of detail depending on the number of included layers are automatically generated. These TLMs are synthesized into actual software or hardware during back-end synthesis.

Figure 31.6 shows the general organization of a final TLM including all layers as generated by SCE for the example of a system architecture with two PEs, *PE0* and *PE1*, representing a software processor and a hardware accelerator, respectively. The two PEs are connected via two busses and an intermediate transducer *T*. Application behaviors *P1* and *P2* within each PE communicate with each other using abstract *send()* or *receive()* primitives. Additional protocol layers are inserted into PEs to realize all such channel communication over external busses. On the software side, this protocol functionality is inserted into corresponding processor model layers as described in Sect. 31.5.1 above.

Network refinement transforms end-to-end messages exchanged over abstract communication channels into point-to-point packet transfers over individual bus segments. In the process, additional CEs such as protocol-level bridges or network-level transducers are inserted as necessary to interconnect, translate, and forward packets between different bus segments or communication media. Communication layers inserted during network refinement include a presentation layer for channel-specific data conversion (*c1*, *c2*, and *c3*) and a network and transport layer (*Net*) for packeting, routing, and end-to-end synchronization.



```
c1.recv(m1);
c2.send(m2);
c3.send(m3);
```

```
c1.send(m1);
c2.recv(m2);
c3.recv(m3);
```

```
send(type msg) {
   char buf[M];
   1: msg->buf;
   2: net.send(buf);
 }
```

```
send(void* msg, len) {
for (pkt in mesg):
      link.send(pkt);
      link.recv(ack);
  }
```

```
send(void* p, len) {
      wait(intr);
      mac.write(p,
            addr);
  }
```

```
intHandler() {
   notify(intr);
}
write(void* d, len, a) {
      bus.writeWord(a,w);
}
```

Fig. 31.6 General layer-based transaction-level communication model

Subsequent link refinement then transforms all point-to-point transfers with each bus segment into actual bus transactions. It inserts a link layer (*Link*) for addressing and interrupt- or polling-based synchronization, a MAC layer for data slicing, and a final *Protocol* layer implementing actual bus state machines. In case of interrupt-driven synchronization, this further includes hardware-level interrupt detection and generation logic (*Int*) as well as software-level interrupt handlers (*Hdlr*) as described in previous sections.

TLMs of the system can be generated to model inter-PE communication at any of these levels. For example, bus channels connecting PEs in a protocol TLM describe communication at the level of individual bus transactions. By contrast, faster but less accurate MAC or link TLMs describes communication at the level of larger unsynchronized or synchronized whole-packet transfers [35]. Finally, a pin- and bus cycle-accurate BFM can be generated by also including a low-level protocol layer describing individual events of bus transactions on each wire.

Basic network and link refinement generates unoptimized code for protocol stacks following a strict layer-based organization as shown in Fig. 31.6. Recent extensions to SCE allow protocol code to be further optimized specifically for efficient back-end synthesis [29]. Protocol stack optimizations flatten and merge basic communication layers above the MAC and apply back-end-specific cross-layer optimizations for message merging, protocol fusion, and interrupt hoisting. Depending on the target PE, fused upper layers are later synthesized into optimized software drivers or hardware transactors. By contrast, MAC and protocol layers are usually provided as pre-designed software or hardware IP in the back-end synthesis databases. In case of hardware PEs, higher-level protocol functionality can be further coupled with and synthesized into low-level bus state machines using protocol IP generators that support corresponding customizations.

31.6 Software Synthesis

Once the designer has settled on a set of architecture decisions and is satisfied with the performance of the generated TLM, the back-end synthesis can be invoked as illustrated on the bottom portion of the flow overview in Fig. 31.1. This includes both software synthesis and hardware synthesis.

Software synthesis [37] uses the generated TLM (i.e., the output of link refinement; see Sect. 31.5) as input and produces embedded code. It is invoked for each programmable PE generating a SW stack matching the overall system. For this, the synthesis approach implements the application modeled in the TLM, which is captured in the SpecC SLDL, on a target processor. The TLM includes SLDL-specific keywords and concepts, such as behaviors, events, channels, and port mappings. To realize these SLDL concepts in target software, one approach would be to replicate the SLDL simulation environment directly on the target platform. This, however, potentially results in overhead for performance and code size. Instead, our software synthesis directly generates embedded ANSI-C code out of the SLDL to achieve compact and efficient code.

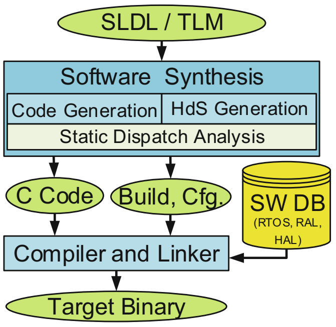


Fig. 31.7 Software generation flow in SCE

We divide *software synthesis* into *code generation* and *Hardware-Dependent Software (HDS) generation*, shown in Fig. 31.7. Code generation deals with the code inside each task and generates flat ANSI-C code out of the hierarchical model captured in the SLDL. Meanwhile, HDS generation creates code for processor-internal and processor-external communication, adjusts for multitasking, and eventually generates configuration/build files (e.g., Makefiles) for the cross compilation process. Software synthesis is supported by a SW database, which contains static target-specific artifacts, such as an operating system, that will be linked in when creating the final binary.

31.6.1 Software Code Generation

Code generation [40] produces sequential ANSI-C code for each task within a programmable PE. It translates the hierarchical composition of behaviors in SpecC into flat C-code consisting of functions and data structures. For SLDL features not natively present in the targeted ANSI-C language (e.g., port/interface concept, hierarchical composition), code generation realizes these SLDL features out of available ANSI-C constructs.

While ANSI-C was chosen as it is widely used in the embedded context and has a rich compiler support, some challenges emerge as ANSI-C does not have language constructs to realize object-oriented programming. As one example, SpecC behaviors offer local variables visible within an instance of the behavior. This could be realized with a class. ANSI-C, however, does not provide an encapsulation/scope for class-local storage as it lacks the class concept. To overcome this limitation, all behaviors' local variables are added to a behavior-representing structure, and all accesses to behavior-local variables are replaced with accesses to the corresponding member of the behavior-representing structure.

Similar challenges appear for methods that are exposed as an interface on the behavior's port. Here, each port also becomes a member of the behavior-representing structure. In addition, the structure includes a virtual function table (VTABLE) pointing to the implementing methods. All calls to these methods are replaced with function calls through the VTABLE entries. In principle, embedded code generation realizes some basic object-oriented concepts on top of ANSI-C. As such, it solves similar issues as C++ to C compilers that translate a class hierarchy into flat ANSI-C code.

31.6.2 Hardware-Dependent Software Generation

The second portion, HDS generation [36, 37], generates code for processor-internal and processor-external communication, including drivers and synchronization (polling or interrupt). It also generates code to execute multiple tasks on the same processor, realizing the task mapping defined in the computation refinement (Sect. 31.3.3). To create the complete binary SW image, it finally generates configuration and build files (e.g., Makefile) which select and configure database components.

To realize the HDS generation, we distinguish code that is only platform specific and code that is both platform and application specific. The former, i.e., platform-specific code, is instantiated from the SW database. This DB includes a selection of RTOSs to provide multitasking and a basic HAL for canonical access to common platform resources. During HDS generation, code for instantiating the selected RTOS is generated, and the RTOS is configured toward the application requirements. In order to unify the access to a wide variety of RTOSs, each RTOS in the DB is accompanied by an RTOS Abstraction Layer (RAL). The RAL abstracts from the particular RTOS's function names and parameters. To ensure a generic API, we investigated different RTOS APIs (uCOS-II, vx-Works, eCos, ITRON, POSIX) and chose common primitives for task scheduling, communication, and synchronization. Along similar lines, the HAL provides canonical access to common platform resources, which we assume to be present in all target platforms (such as timers and an interrupt controller). The canonical APIs (as realized by RAL and HAL) limit the interdependency between HDS generation and the selected target architecture, thus making HDS generation more scalable.

Code that is both application and platform specific is produced by the HDS generation. This includes code for multitasking, internal communication, and external communication. For multitasking, application-specific code is generated to instantiate and control the tasks as defined the TLM (e.g., *T1, T2, and T3* in Fig. 31.5). Internal communication, which occurs within the same PE, is realized along the same lines. Channels *C1* and *C2* in Fig. 31.5 are examples of PE-internal communication. These channels are replaced with an implementation on top of the RAL of the selected RTOS (e.g., using semaphores and memcpy). External communication and synchronization occur between PEs. As part of communication refinement (Sect. 31.5.2), external communication has been refined into a set of stacked channels as visualized in Fig. 31.6. To realize the specified communication, HDS generation traverses the TLM from the innermost layer (e.g., *PE0*'s presentation layer with the stacked channels *c1*, *c2*, and *c3* in Fig. 31.6) and generates a matching driver stack. The generated driver stack utilizes the RAL (e.g., for interrupt registration, handling, and synchronization) as well as the HAL (e.g., for accessing the processor bus).

Combining the outputs of code generation and HDS generation yields all code for a PE. Using a cross-compiler, the final target binary is created. The SW synthesis process repeats for each software PE in the TLM. The produced binaries can directly execute on the target processor(s) of the final hardware. For early binary validation, before availability of the real hardware, *virtual platforms* can be used. To facilitate this step, our software synthesis also includes a model refinement step that converts the input TLM into an ISS-based system model. For this, SW synthesis removes the processor model for which it has generated the SW earlier (i.e., everything inside the *HAL* layer of *PE0* in Fig. 31.6) and replaces it with an ISS from the database. The integrated ISS instance can then run the generated binary including all platform-specific interactions (e.g., for communication with custom hardware) for early binary validation and further codevelopment.

31.6.3 Software Optimization

Software synthesis offers several optimization opportunities across code generation and HDS generation. One example is static dispatch analysis, which, if permissible, eliminates the overhead of virtual function calls. Virtual function calls are common in object-oriented code to allow multiple implementations for the same interface. This is very frequently used in layered implementation of complex systems (such as in our layer-based realization of communication between PEs). The overhead of virtual functions appears in many languages, such as C++, SpecC, SystemC, as well as in our generated code. Generally, the overhead for a virtual function call itself is low (2 cycles [14]). More importantly, however, this indirection hinders inlining optimizations. Especially when the callee function has only minimal computation, the virtual function call overhead becomes quite significant on embedded platforms.

To improve the performance and quality of SLDL synthesized embedded SW, we have enhanced our embedded ANSI-C code synthesis with a dispatch-type analysis to reduce/eliminate this overhead [43]. Our approach utilizes the fact that the complete model (with all connectivity) is known during SW synthesis and no SW is linked later on top of the generated code. Hence, static dispatch-type analysis can determine if a direct call is possible (i.e., the call target can be determined statically) and replace a virtual call with a direct function call.

31.7 Hardware Synthesis

Hardware synthesis refines models of custom hardware processors in the TLM down to complete RTL implementations following an extended High-Level Synthesis (HLS) design step [29]. The generated RTL can then be further synthesized down to a gate-level netlist and final physical realization using standard logic and back-end synthesis flows. SCE integrates a hardware synthesis flow that combines existing commercial, off-the-shelf HLS tools for synthesis of computation with the capability

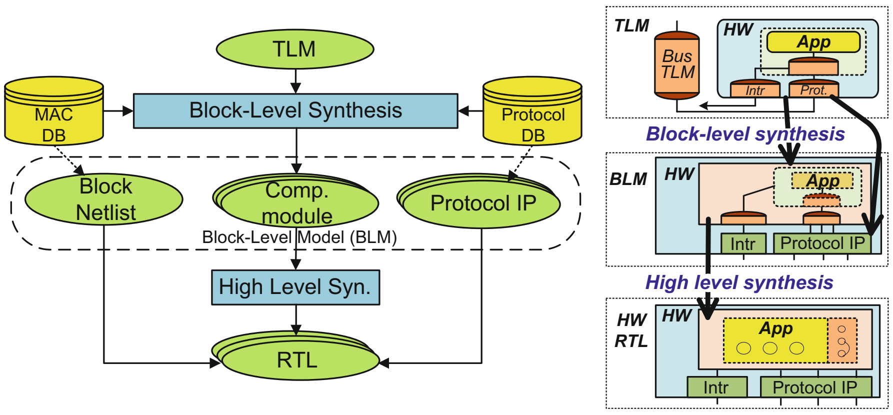


Fig. 31.8 Hardware synthesis flow in SCE

to synthesize communication interfaces and hardware bus transactors supporting a wide range of optimized target implementations [29].

Figure 31.8 shows the overall flow of hardware synthesis in SCE. We follow a three-step methodology to transform the TLM into a SystemC/C++-based Block-Level Model (BLM), further synthesize computation blocks in the BLM down to cycle-accurate RTL using a traditional HLS tool, and then perform logic synthesis to generate a final gate-level netlist.

31.7.1 Block-Level Synthesis

Block-level synthesis refines the TLM generated during front-end computation and communication refinement (see Sect. 31.5) down to block-level modules, protocol IPs, and a block-level netlist integrating all of these together. Low-level blocks for protocol layer IPs are thereby taken directly out of a protocol database in pre-written RTL form. By contrast, other higher-layer blocks are converted from SpecC into synthesizable SystemC or C++ code to be further synthesized and as supported by existing HLS tools.

Most HLS tools can synthesize multiple single-threaded C++/SystemC modules with the capability to stitch blocks together. However, they cannot automatically partition preexisting code. In order to provide a general approach that can be easily adapted to different HLS back ends, we partition the code into separately synthesized modules that are integrated through our own netlisting engine. In the process, each computation or communication behavior in the TLM is converted into a separate, synthesizable hardware block. After block partitioning, TLM communication stacks are inlined into each accessing computation block. This enables computation/communication cooptimizations in the following HLS step, with the scheduling freedom to overlap computation with communication and to perform general, joint optimizations, such as resource sharing or parallelization.

Communication layers in the TLM connect to each other and to computation behaviors via an interface mechanism. Such high-level functional interface and variable ports are converted into low-level wire, register or First-In First-Out (FIFO) interfaces between blocks in the BLM as supported by the underlying HLS engine. At the lowest MAC and protocol levels, thin Media Access Control (MAC) layer implementations inserted from a hardware database provide the glue logic between application-specific higher layers and pre-written bus protocol IPs. Synthesizable MAC database implementations thereby replace canonical MAC layer models and interfaces in the TLM with code that provides equivalent, canonical MAC-level bus interfaces to higher layers while internally translating each transaction into corresponding pin- and wire-level interactions necessary to interface with a target-specific bus protocol IP component.

31.7.2 Protocol IP Generation

Each bus protocol layer in the TLM is replaced with an actual protocol IP from the protocol database. Based on parameters, such as the type and number of ports, a protocol generator thereby creates a custom IP block from pre-written RTL templates in the protocol database to implement external bus protocols and, depending on the synchronization mechanism selected and defined in the TLM, either interrupt generation or polling logic. The internal interface of bus protocol IPs is designed to match associated protocol wrappers in the MAC database, which, when synthesized together with other blocks, will realize appropriate pin- and wire-level interactions with the generated IP.

31.7.3 RTL Netlisting and Synthesis

The connectivity of all blocks, IPs, and external ports is converted into an block-based RTL netlist to complete the block-level synthesis step. As indicated in Sect. 31.7.1 above, in the process high-level functional variable and channel interfaces between modules are converted into wires, registers, and glue logic connecting the pin-level FIFO ports of synthesized blocks and protocol IPs. After block-level synthesis, computation blocks in the generated BLM are further synthesized down to cycle-accurate RTL descriptions using an external HLS tool. Generated RTL blocks, RTL bus protocol IPs, and the block netlist are then synthesized into a gate-level netlist using a traditional logic synthesis tool.

Utilizing tools and databases along with a HLS engine, our flow can be easily adapted to different HLS back ends and synthesis targets. Our current SCE setup includes support for Mentor (formerly Calypto) Catapult, and Xilinx Vivado HLS tools targeting ASICs or FPGAs with Mentor Precision or Xilinx ISE as logic synthesis back ends.

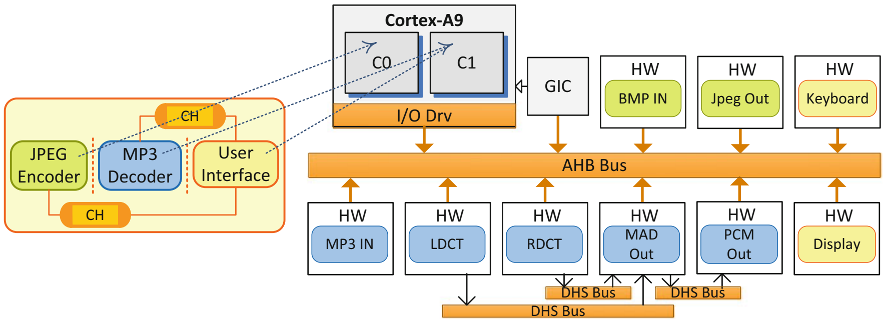


Fig. 31.9 Cellphone baseband example

31.8 Experimental Results

We demonstrate the benefits of SCE for fast and accurate system-level Design Space Exploration (DSE) and synthesis as applied to a cellphone baseband example running concurrent Motion-JPEG (M-JPEG), MP3, and user interface tasks on a dual-core 650 MHz Cortex-A9 platform. The overall architecture of the system is shown in Fig. 31.9 [31]. The MP3 decoder and JPEG encoder use optional hardware accelerators to perform audio decoding or Discrete Cosine Transform (DCT) and quantization acceleration, respectively. Tasks communicate with external hardware and the rest of the system via an AHB bus and 12 interrupts. In this experiment, MP3 decodes 13 frames at a bitrate of 384 kbit/s, and JPEG encodes 10 frames of a movie with standard 352×288 resolution at a rate of 30 frames/s.

We explored a wide range of architectures by applying different OS and processor configurations, including mapping of M-JPEG tasks and interrupts to two different cores (C0 and C1) in a dual-core architecture. We explored both First-Come First-Serve (FCFS)/FIFO and priority (Prty) scheduling. In dual-core architectures with task-attached interrupts, application tasks are distributed among two cores and a task, and its associated interrupts are mapped to the same core. By contrast, dual-core architectures with a core-attached interrupt always handle all interrupts on C1. Figure 31.10 summarizes the average frame delays and average absolute errors in frame delays as well as maximum error bars of MP3 and JPEG tasks. Frame delays and frame delay errors are reported both for high- and low-level TLM simulations of the platform at a link and MAC level of communication abstraction, respectively, as compared to an ISS simulation. Task delays were back-annotated at the function level from measurements taken on the ISS. Moreover, average Linux context-switch overhead was measured and back-annotated into the OS model.

As can be seen, the best MP3 performance is achieved when a higher priority is assigned to MP3, or MP3 and JPEG are running on separate cores. In other configurations, average MP3 frame delay is close to its deadline boundary (i.e., 26.1 ms).

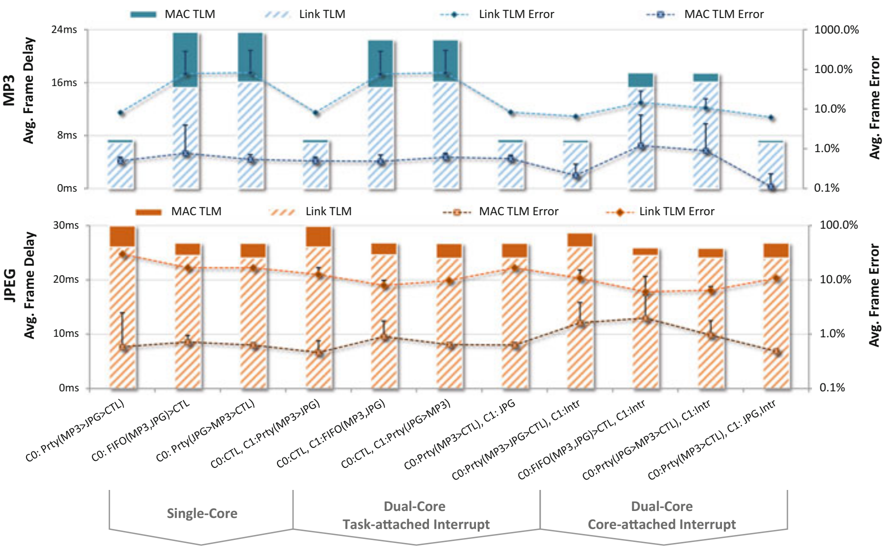


Fig. 31.10 Cellphone design space exploration

Minimized JPEG delay is obtained from configurations with FIFO scheduling, when JPEG has higher priority or when it runs on a separate core. All combined, explorations confirm that shortest-job-first or rate-monotonic scheduling guarantee that MP3 and JPEG meet their performance requirements. Overall, optimized MP3 and JPEG performance is achieved when tasks run on separate cores. Finally, by mapping all interrupts to a separate core (C1), we only see slight performance benefits in MP3 and JPEG delays.

Overall, the SCE system design front end provides an efficient platform for rapid, early, and accurate DSE. On average, low-level MAC TLMs generated by SCE simulate with 1,400 Million Instructions Per Second (MIPS) at less than 1% error. A model at higher level of abstraction that does not account for synchronization overhead can achieve even higher simulation speeds, but errors can reach as high as 100% in some configurations where inaccuracies lead to MP3 and JPEG tasks executing in the wrong order.

With the definition of architecture and mapping in place, the SCE system design back end is activated to generate both software and hardware for each processing element.

31.8.1 Software Synthesis

SW synthesis has been used to create the implementation for the cellphone example described above. To provide more insight into the optimization potential offered by SW synthesis, we will look into the static dispatch analysis, which can convert virtual function calls to direct calls and by that avoid the VTABLE indirection overhead. For this, we look more closely into the JPEG compression task of the cellphone platform.

We anticipate the most optimization potential in the generated drivers as they use a layered approach and contain very little own computation. To gain more insight into the influence of communication, we vary the communication granularity. The input (*BMP In* to *JPEG*) can either operate on individual pixels or coarser on a whole pixel row. The output (*JPEG* to *JPEG Out*) is configurable to operate on individual bytes or more efficiently by using a queue for buffered communication (queue size 256 bytes). If selected, the queue is mapped to the processor. This increases efficiency as the processor writes with small operations (mostly bytes) and the JPEG output retrieves larger blocks. The coarser granularity reduces the number of transactions by using fewer, yet larger transactions. This results in fewer calls into the layered drivers. Thus, we anticipate less optimization potential through static dispatch analysis with fewer, larger transactions. Please note that varying the communication granularity is only done for analysis purposes. A real product would use the coarsest granularity to reduce overhead.

Table 31.2 quantifies the static dispatch analysis optimization for the JPEG encoder example in its four configurations. In addition to the execution time, it shows the speedup as well as the number of virtual function calls converted into direct calls.

Table 31.2 confirms the expectations. With the coarsest communication granularity (row, queue), the JPEG encoder executes the fastest (39.6 ms), while the finest granularity (pixel, byte) executes the slowest (57 ms). The number of MAC driver calls gives an indication for the slowdown. With 41 K calls, the finest granularity needs almost 4 times as many calls for the same amount of data as the coarsest granularity. All configurations significantly benefit from static dispatch analysis. The speedup through converting to direct calls ranges from 12.4% for the coarsest to 16.1% for the finest grained communication. Although fewer port method calls are converted to direct calls in the fine-grained case (81 instead of 89), the drivers are called much more frequently (41 K calls instead of 11 K calls) leading to the larger speed up through our optimization. Irrespective of the communication granularity, our approach offers tremendous benefits as it eliminates the virtual function calls for the generated embedded SW while also improving code readability.

Table 31.2 Static dispatch analysis optimization

Input	Output	Exec. time [ms]	Exec. time (opt) [ms]	Improvement	# Direct calls	# MAC drv. calls
Row	Queue	39.64	34.74	12.4%	89	11469
	Byte	42.69	36.89	13.6%	85	16560
Pixel	Queue	54.01	46.31	14.3%	85	36351
	Byte	57.06	47.86	16.1%	81	41442

31.8.2 Hardware Synthesis

To demonstrate the hardware back end of SCE, we have applied the flow to synthesis of hardware accelerators for the JPEG encoder subsystem of the cellphone example [29]. Specifically, the DCT and Quantize blocks (DCT-Q) in the JPEG encoder are mapped into hardware while the rest of the application functionality executes on the application processor. We used Mentor Catapult as HLS back end, targeting an ARM+FPGA platform consisting of a Freescale i.MX21 applications processor and a Xilinx Spartan-3 Field-Programmable Gate Array (FPGA) communicating over Freescale's EIM bus.

Using our fully automated hardware synthesis flow, we were able to synthesize the TLM into RTL ready for further FPGA download within minutes, yielding substantial productivity gains compared to a manual design process. On the processor side, software was automatically synthesized, combined with driver code, and cross-compiled into a Linux executable (see Sect. 31.8.1 above).

We synthesized computation blocks and communication channels using polling- or interrupt-based synchronization targeting a 50MHz clock frequency. We applied protocol stack and protocol coupling optimizations as described in Sect. 31.5.2 during synthesis. We compare FPGA resource usage and hardware latency of the optimized design (POPT) against an unoptimized design (NOPT) and a purely manual implementation of communication interfaces (MAN). For a fair comparison, the manual design utilizes the same computation block and the same firmware code synthesized by a HLS tool and SCE, respectively. We evaluated end-to-end hardware latency by instrumenting the JPEG encoder software to record the average turnaround times over 180 DCT-Q invocations, including all communication and synchronization overhead. Area and latency results of synthesized accelerator PEs are summarized in Table 31.3.

We can observe that the latency of final, optimized hardware synthesized with our flow is always significantly less than in a manual design. In contrast to a manual design in which communication and computation blocks are designed separately to manage complexities, our approach is able to perform cooptimizations across computation and communication boundaries [29]. In the DCT-Q case, data is processed strictly in the order it is received and sent, which allows computation and communication to be pipelined and scheduled in parallel. This can overlap and hide communication latencies behind computation delays. Due to its complexity

Table 31.3 JPEG hardware synthesis results

Synch.	Opt.	LUTs	FFs	Logic score	Mem [bytes]	Latency
Intr.	NOPT	5334	3197	8531	1024	59.9 ms
	POPT	4091	2046	6137 (−28%)	1024 (0%)	59.7 ms (0%)
	MAN	3284	2182	5466 (−36%)	1024 (0%)	89.0 ms (49%)
Poll.	NOPT	5407	3302	8709	1024	9.4 ms
	POPT	4133	2341	6474 (−26%)	1024 (0%)	9.8 ms (4%)
	MAN	3281	2179	5460 (−37%)	1024 (0%)	28.4 ms (203%)

and non-modularity, such optimizations are typically not applied in manual designs. However, a naive, unoptimized realization of such computation/communication codesign can lead to a large increase in total area. The proposed protocol stack optimizations prove to be efficient in reducing this area overhead through resource sharing, which results in up to average 26% logic reduction compared to an unoptimized design. Overall quality of results is comparable to or better than a manual design, where on average 2.2 times improvement in latency can be achieved.

31.9 Conclusions

Designing modern embedded systems is increasingly challenging due to hardware complexity (e.g., increasing heterogeneity), functional complexity (increasingly complex, integrated features), more stringent nonfunctional requirements, all while reducing the time to market. Addressing the complexity challenges requires raising the level of abstraction to jointly consider digital hardware and software while paving a path to automated implementation.

In this chapter, we have presented our comprehensive system-level design framework, the System-on-Chip Environment (SCE). SCE realizes a top-down codesign flow, from an abstract, functional specification captured in the SpecC language down to synthesized hardware and software. It supports a wide range of heterogeneous target platforms consisting of custom hardware components, embedded software processors, and complex communication bus architectures.

This chapter has provided an overview and highlighted key aspects of the SCE framework: the underlying language and simulation principles as well as the layer-based modeling and refinement. It also provided an overview of the software and hardware synthesis processes. We have illustrated the capabilities of our SCE codesign framework through a cellphone example targeted to a heterogeneous target architecture.

The focus of SCE is on the modeling, refinement, and implementation synthesis process. In its current form, all design decisions have to be entered manually through SCE's Graphical User Interface (GUI) or via an external scripting interface. This complements other system-level design tools that have a focus on automated Design Space Exploration (DSE) as presented in the book Part 3, "Design Space Exploration". In order to leverage different strengths of existing tools [20], we have combined SCE with other academic DSE engines to provide a comprehensive, seamless, and fully automated system-level synthesis solution all the way from high-level, data-flow-based system specifications down to concrete HW and SW implementations for generic MPSoC platforms [22]. This provides a proof-of-concept realization of a complete system-level synthesis solution. The authors thank Dongwook Lee and Parisa Razaghi for their help in preparing this manuscript and the anonymous reviewers for their valuable suggestions on its improvement. The authors express gratitude to all members of the SpecC team who have contributed to SCE over the years, namely Samar Abdi, David Berner, Lukai Cai, Pramod Chandraiah, Che-Wei Chang, Vincent Chang, Weiwei Chen, Alexan- der Gluhak, Yitao Guo, Xu Han, Ran Hao, Eric Johnson, Jon Kleinsmith, Deepak Mishra, Guantao Liu, Junyu Peng, Gautam Sachdeva, Yasaman Samei, Dongwan Shin, Sanyuan Tang, Ines Viskic, Qiang Xie, Haobo Yu, Pei Zhang, Jiaxing Zhang, Shuqing Zhao, Jianwen Zhu. Last but not least, the authors thank Daniel D. Gajski for his visionary leadership.

References

1. Balarin F, Chiodo M, Giusto P, Hsieh H, Jurecska A, Lavagno L, Passerone C, Sangiovanni-Vincentelli A, Sentovich E, Suzuki K, Tabbara B (1997) Hardware-software co-design of embedded systems: the POLIS approach. Kluwer Academic Publishers, Boston
2. Balarin F, Watanabe Y, Hsieh H, Lavagno L, Passerone C, Sangiovanni-Vincentelli A (2003) Metropolis: an integrated electronic system design environment. Trans Comput 36(4):45–52
3. Cai L, Gerstlauer A, Gajski D (2005) Multi-metric and multi-entity characterization of application for early system design exploration. In: IEEE/ACM Asia and South Pacific design automation conference (ASP-DAC)
4. Cai L, Gerstlauer A, Gajski DD (2004) Retargetable profiling for rapid, early system-level design space exploration. In: Proceedings of the design automation conference (DAC), San Diego
5. Chen W, Han X, Chang CW, Dömer R (2013) Advances in parallel discrete event simulation for electronic system-level design. IEEE Des Test Comput 30(1):45–54
6. Chen W, Han X, Chang CW, Liu G, Dömer R (2014) Out-of-order parallel discrete event simulation for transaction level models. IEEE Trans Comput Aided Des Integr Circuits Syst (TCAD) 33(12):1859–1872. doi:10.1109/TCAD.2014.2356469
7. Chen W, Han X, Dömer R (2011) Multi-core simulation of transaction level models using the system-on-chip environment. IEEE Des Test Comput 28(3):20–31
8. Chen W, Han X, Dömer R (2012) Out-of-order parallel simulation for ESL design. In: Proceedings of the design, automation and test in Europe conference and exhibition (DATE)
9. Cortadella J, Kondratyev A, Lavagno L, Massot M, Moral S, Passerone, C, Watanabe Y, Sangiovanni-Vincentelli A (2000) Task generation and compile time scheduling for mixed data-control embedded software. In: Proceedings of the design automation conference (DAC), Los Angeles
10. Dömer R (1999) The SpecC internal representation. Technical report, information and computer science. University of California, Irvine. SpecC V 2.0.3
11. Dömer R (2000) System-level modeling and design with the SpecC language. Ph.D. thesis, University of Dortmund
12. Dömer R, Gerstlauer A, Gajski D (2002) SpecC language reference manual, version 2.0. SpecC Technology Open Consortium. http://www.specc.org
13. Dömer R, Gerstlauer A, Peng J, Shin D, Cai L, Yu H, Abdi S, Gajski D (2008) System-on-chip environment: a SpecC-based framework for heterogeneous MPSoC design. EURASIP J Embed Syst 2008:647953
14. Driesen K, Hölzle U (1996) The direct cost of virtual function calls in C++. SIGPLAN Not 31(10):306–323. doi:10.1145/236338.236369
15. Ecker W, Müller W, Dömer R (2009) Hardware dependent software: introduction and overview. In: Ecker W, Müller W, Dömer R (eds) Hardware dependent software: principles and practice. Springer, Berlin
16. Eclipse Foundation. Eclipse. http://www.eclipse.org/
17. Fujimoto R (1990) Parallel discrete event simulation. Commun ACM 33(10):30–53
18. Gajski DD, Zhu J, Dömer R, Gerstlauer A, Zhao S (2000) SpecC: specification language and design methodology. Kluwer Academic Publishers, Boston

19. Gerstlauer A, Dömer R, Peng J, Gajski DD (2001) System design: a practical guide with SpecC. Kluwer Academic Publishers, Boston
20. Gerstlauer A, Haubelt C, Pimentel A, Stefanov T, Gajski D, Teich J (2009) Electronic system-level synthesis methodologies. IEEE Trans Comput Aided Des Integr Circuits Syst 28(10):1517–1530
21. Gerstlauer A, Shin D, Peng J, Dömer R, Gajski DD (2007) Automatic layer-based generation of system-on-chip bus communication models. IEEE Trans Comput Aided Des Integr Circuits Syst (TCAD) 26(9):1676–1687
22. Gladigau J, Gerstlauer A, Haubelt C, Streubühr M, Teich J (2011) Automatic system-level synthesis: from formal application models to generic bus-based MPSoCs. Trans High-Perform Embed Archit Compil (Transactions on HiPEAC) 5(4):1–22
23. Grötker T, Liao S, Martin G, Swan S (2002) System design with SystemC. Kluwer Academic Publishers, Dordrecht
24. Grüttner K (2015) Application mapping and communication synthesis for object-oriented platform-based design. Ph.D. thesis, Carl von Ossietzky University Oldenburg
25. Ha S, Kim S, Lee C, Yi Y, Kwon S, Joo YP (2008) Peace: a hardware-software codesign environment for multimedia embedded systems. ACM Trans Des Autom Electron Syst 12(3):24:1–24:25. doi:10.1145/1255456.1255461
26. Hong S, Yoo S, Lee S, Lee S, Nam HJ, Yoo BS, Hwang J, Song D, Kim J, Kim J, Jin H, Choi KM, Kong JT, Eo S (2006) Creation and utilization of a virtual platform for embedded software optimization: an industrial case study, Seoul
27. Kangas T, Kukkala P, Orsila H, Salminen E, Hännikäinen M, Hämäläinen TD, Riihimäki J, Kuusilinna K (2006) UML-based multiprocessor SoC design framework. ACM Trans Embed Comput Syst 5(2):281–320. doi:10.1145/1151074.1151077
28. Keinert J, Streubühr M, Schlichter T, Falk J, Gladigau J, Haubelt C, Teich J, Meredith M (2009) SystemCoDesigner – an automatic ESL synthesis approach by design space exploration and behavioral synthesis for streaming applications. Trans Des Autom Electron Syst 14(1):1:1–1:23
29. Lee D, Park H, Gerstlauer A (2012) Synthesis of optimized hardware transactors from abstract communication specifications. In: Proceedings of the international conference on hardware/software codesign and system synthesis (CODES+ISSS)
30. Nikolov H, Thompson M, Stefanov T, Pimentel A, Polstra S, Bose R, Zissulescu C, Deprettere E (2008) Daedalus: toward composable multimedia MP-SoC design. In: Proceedings of the design automation conference (DAC 2008), pp 574–579
31. Razaghi P, Gerstlauer A (2014) Host-compiled multi-core system simulation for early real-time performance evaluation. ACM Trans Embed Comput Syst (TECS) 13(5s):166:1–166:26
32. Ritz S, Pankert M, Zivojnvic V, Meyr H (1993) High-level software synthesis for the design of communication systems. IEEE J Sel Areas Commun 11(3)348–358. doi:10.1109/49.219550
33. Samei Y, Dömer R (2014) Automated estimation of power consumption for rapid system level design. In: Proceedings of the IEEE international performance computing and communications conference
34. Samei Y, Dömer R (2014) Powermonitor: a versatile API for automated power-aware ESL design. In: Proceedings of the forum on specification and design languages (FDL)
35. Schirner G, Dömer R (2009) Quantitative analysis of the speed/accuracy trade-off in transaction level modeling. ACM Trans Embed Comput Syst (TECS) 8(1):4:1–4:29
36. Schirner G, Dömer R, Gerstlauer A (2009) High-level development, modeling and automatic generation of hardware-dependent software. In: Ecker W, Müller W, Dömer R (eds) Hardware dependent software: principles and practice. Springer, Berlin
37. Schirner G, Gerstlauer A, Dömer R (2008) Automatic generation of hardware dependent software for MPSoCs from abstract system specifications. In: Proceedings of the design automation conference. Asia and South Pacific (ASPDAC), Seoul
38. Schirner G, Gerstlauer A, Dömer R (2010) Fast and accurate processor models for efficient MPSoC design. ACM Trans Des Autom Electron Syst (TODAES) 15(2):10:1–10:26

39. Viskic I, Dömer R (2006) A flexible, syntax independent representation (SIR) for system level design models. In: Proceedings of the EUROMICRO conference on digital system design (DSD), pp 288–294
40. Yu H, Dömer R, Gajski D (2004) Embedded software generation from system level design languages. In: Proceedings of the design automation conference. Asia and South Pacific (ASPDAC), Yokohama
41. Zhang J, Schirner G (2014) Automatic specification granularity tuning for design space exploration. In: Proceedings of the ACM/IEEE conference on design, automation & test in Europe (DATE), Dresden. doi:10.7873/DATE.2014.227
42. Zhang J, Schirner G (2015) Towards closing the specification gap by integrating algorithm-level and system-level design. Des Autom Embed Syst (DAEM) 19:389–419. Springer. doi:10.1007/s10617-015-9161-1
43. Zhang J, Tang S, Schirner G (2015) Reducing dynamic dispatch overhead (DDO) of SLDL-synthesized embedded software. In: Asia and South Pacific design automation conference (ASPDAC), Chiba