Electronic System-Level Synthesis Methodologies

Andreas Gerstlauer, *Member, IEEE*, Christian Haubelt, *Member, IEEE*, Andy D. Pimentel, *Senior Member, IEEE*, Todor P. Stefanov, *Member, IEEE*, Daniel D. Gajski, *Fellow, IEEE*, and Jürgen Teich, *Senior Member, IEEE*

***Abstract*—With ever-increasing system complexities, all major semiconductor roadmaps have identified the need for moving to higher levels of abstraction in order to increase productivity in electronic system design. Most recently, many approaches and tools that claim to realize and support a design process at the so-called electronic system level (ESL) have emerged. However, faced with the vast complexity challenges, in most cases at best, only partial solutions are available. In this paper, we develop and propose a novel classification for ESL synthesis tools, and we will present six different academic approaches in this context. Based on these observations, we can identify such common principles and needs as they are leading toward and are ultimately required for a true ESL synthesis solution, covering the whole design process from specification to implementation for complete systems across hardware and software boundaries.**

***Index Terms*—Electronic system level (ESL), methodology, synthesis.**

I. Introduction

In ORDER to increase design productivity, raising the level of abstraction to the electronic system level (ESL) seems mandatory. Surely, this must be accompanied by new design automation tools [1]. Many approaches exist today that claim to provide ESL solutions. In [2], Densmore *et al.* define an ESL classification framework that focuses on individual design tasks by reviewing more than 90 different point tools. Many of these tools are devoted to modeling purposes (functional or platform) only. Other tools provide synthesis functionality by either software code generation or C-to-RTL high-level synthesis. However, true ESL synthesis tools show the ability to combine design tasks under a complete flow that can generate systems across hardware and software boundaries from an algorithmic specification. In this paper, we therefore aim to provide an extended classification focusing on such complete ESL flows on top of individual point solutions.

Manuscript received February 24, 2009; revised June 6, 2009. Current version published September 18, 2009. This paper was recommended by Associate Editor P. Eles.

A. Gerstlauer is with the Department of Electrical and Computer Engineering, University of Texas at Austin, Austin, TX 78712 USA (e-mail: gerstl@ece.utexas.edu).

C. Haubelt and J. Teich are with the Department of Computer Science, University of Erlangen–Nuremberg, 91054 Erlangen, Germany (e-mail: haubelt@cs.fau.de; teich@cs.fau.de).

A. D. Pimentel is with the Informatics Institute, University of Amsterdam, 1098 XG Amsterdam, The Netherlands (e-mail: a.d.pimentel@uva.nl).

T. P. Stefanov is with the Leiden Institute of Advanced Computer Science, Leiden University, 2300 RA Leiden, The Netherlands (e-mail: stefanov@liacs.nl).

D. D. Gajski is with the Center for Embedded Computer Systems, University of California, Irvine, CA 92697 USA (e-mail: gajski@cecs.uci.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCAD.2009.2026356

Typically, ESL synthesis tools are domain specific and rely on powerful computational models [3] for description of desired functional and nonfunctional requirements at the input of the synthesis flow. Such well-defined rich input models are a prerequisite for later analysis and optimization. Typical computational models in digital system design are process networks, dataflow models, or state machines. On the other hand, implementation platforms for such systems are often heterogeneous or homogeneous multiprocessor system-on-chip (MPSoC) solutions [4]. The complexity introduced by both input computational model and target implementation platform results in a complex synthesis step, including hardware/software partitioning, embedded software generation, and hardware accelerator synthesis. Aside from this, at ESL, the number of design decisions, particularly in communication synthesis, is compelling in contrast to lower abstraction levels. Even more so, due to the increasing number of processors in MPSoCs, the impact of the quality in computation and communication synthesis is ever increasing.

In this paper, we aim to provide an analysis and comparative overview of the state-of-the-art current directions and future needs in ESL synthesis methodologies and tools. After identifying common principles based on our observations, we develop and propose a general framework for classification and, eventually, comparison of different tools in Section II. In Section III, we then present, in detail, a representative selection of three ESL approaches developed in our groups. To provide a more complete overview, Section IV briefly discusses three related academic approaches. After introducing all six tools, we follow with a comparison and discussion of future research directions based on our classification criteria in Section V. Finally, this paper concludes with a summary in Section VI.

II. Electronic System Design

In this section, we will identify common principles in existing ESL synthesis methodologies and develop a novel classification for such approaches. Later, this will enable a comparison of different methodologies. Furthermore, based on such observations, synergies between different approaches can be explored, and corresponding interfaces between different tools can be defined and established in the future.

A. Design Flow

Before deriving a model for ESL synthesis, we start by defining the system design process in general. As nearly all ESL synthesis methodologies follow a top–down approach, a definition of the design process should support this view.

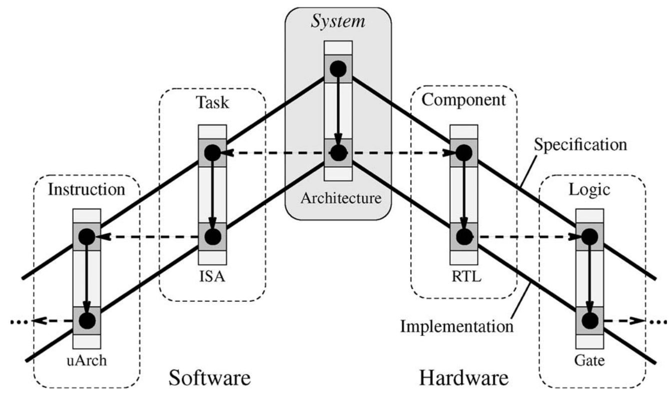


Fig. 1. Electronic system design flow.

Furthermore, it should show the concurrent design of hardware and software and required synthesis steps. A visualization of this is given by the *double roof model* [5] shown in Fig. 1.

The double roof model defines the ideal top–down design process for embedded hardware/software systems. One side of the roof corresponds to the *software design process*, whereas the other side corresponds to the *hardware design process*. Each side is organized in different *abstraction levels*, e.g., task and instruction levels or component and logic levels for the software or hardware design processes, respectively. There is one common level of abstraction, the ESL, at which we cannot distinguish between hardware and software. At each level, in a *synthesis step* (vertical arrow), a *specification* is transformed into an *implementation*. Horizontal arrows indicate the step of passing models of individual elements in the implementation directly to the next lower level of abstraction as specifications at its input.

The double roof model can be seen as extending the Y-chart [6] by an explicit separation of software and hardware design. Furthermore, for simplicity, we do not include a third *layout* roof representing a physical view of the design. Note, however, that layout information, while traditionally being of minor importance, is increasingly employed even at the system level, e.g., through early floorplanning, to account for spatial effects such as activity hot spots [7], wiring capacitances, or distance-dependent latencies [8].

The design process represented by the double roof model starts with an ESL specification given by a behavioral model that is often some kind of network of processes communicating via channels. In addition, a set of mapping constraints and implementation constraints (maximum area, minimal throughput, etc.) is given. The platform model at ESL is typically a structural model consisting of architectural components such as processors, busses, memories, and hardware accelerators. The task of *ESL synthesis* is then the process of selecting an appropriate platform architecture, determining a mapping of the behavioral model onto that architecture, and generating a corresponding implementation of the behavior running on the platform. The result is a refined model containing all design decisions and quality metrics, such as throughput, latency, or area. If selected, components of this refined model are then used as input to the design process at lower abstraction levels, where each hardware or software processor in the system architecture is further implemented separately.

Synthesis at lower levels is a similar process in which a behavioral or functional specification is refined down into a structural implementation. However, depending on the abstraction level, the granularity of objects handled during synthesis differs, and some tasks might be more important than others. For instance, at the *task level* on the software side, communicating processes/threads bound to the same processor must be translated into the instruction-set architecture (ISA) of the processor, targeted toward and running on top of an off-the-shelf real-time operating system (RTOS) or a custom-generated runtime environment. This software task synthesis step is typically performed using a (cross-)compiler and linker tool chain for the selected processor and RTOS. At the *instruction level*, the instruction set of programmable processors is then realized in hardware by implementing the underlying microarchitecture. This step results in a structural model of the processor's datapath organization, usually specified as a register-transfer level (RTL) description.

On the other hand, at the *component level* on the hardware side, processes selected to be implemented as hardware accelerators are synthesized down to an RTL description in the form of controller state machines that drive a datapath consisting of functional units, register files, memories, and interconnect. This refinement step is commonly referred to as behavioral or high-level synthesis. Today, there are several tools available to perform such a high-level synthesis automatically [9], [10]. Finally, at the *logic level*, the granularity of the objects considered during logic synthesis then corresponds to Boolean formulas implemented by logic gates and flip-flops.

An important observation that can be made from Fig. 1 is that, at the RT level, hardware and software worlds unite again, both feeding into (traditional) logic design processes down to the final manufacturing output. In addition, we note that a top–down ESL design process relies on the availability of design flows at the component or task (and eventually logic and instruction) levels to feed into on the hardware and software side, respectively. Lower level flows can be supplied either in the form of corresponding synthesis tools or by providing predesigned intellectual property (IP) components to be plugged into the system architecture.

B. Synthesis Process

Before identifying the main tasks in ESL synthesis, we first develop a general synthesis framework applicable at all levels. As discussed in the previous section, during synthesis, a specification is generally transformed into an implementation. This abstract view can be further refined into an *X-chart* as shown in Fig. 2. With this refinement, we can start to define terms that are essential in the context of synthesis.

A *specification* is composed of a *behavioral model* and *constraints*. The behavioral model represents the intended functionality of the system. Its expressibility and analyzability can be declared by its underlying *model of computation* (MoC) [3]. The behavioral model is often written in some programming language (e.g., C, C++, or JAVA), system-level description

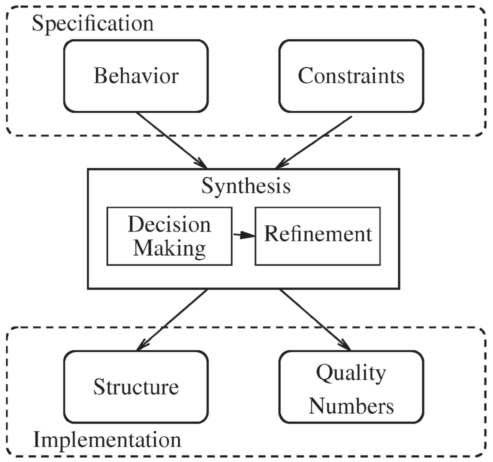


Fig. 2. Synthesis process.

language (SLDL) (e.g., SpecC or SystemC), or a hardware description language (HDL) (such as Verilog or VHDL).

The constraints often include an implicit or explicit *platform model* that describes an architecture template, e.g., available resources, their capabilities (or services), and their interconnections. Analogous to the classification of behavioral models into MoCs, specific ways of describing architecture templates can be generalized into *models of architecture* (MoAs) [11]. Similar to the concept of MoCs, an MoA describes the characteristics underlying a class of platform models in order to evaluate the richness of supported target architectures at the input of a synthesis tool. ESL architecture templates can be coarsely subdivided based on their processing, memory, and communication hierarchy. On the processing side, examples include single-processor systems, hardware/software processor/coprocessor systems, and homogeneous, symmetric or heterogeneous, asymmetric multiprocessor/multicore systems (MPSoCs) [4].[1] Memorywise, we can distinguish shared versus distributed memory architectures. Finally, communication architectures can be loosely grouped into shared bus-based or network-on-chip (NoC) approaches. Aside from the architecture template, constraints typically contain mapping restrictions and additional constraints on nonfunctional properties like maximum response time or minimal throughput.

The synthesis step then transforms a specification into an implementation. An *implementation* consists of a *structural model* and *quality numbers*. The structural model is a refined model from the behavioral model under the constraints given in the specification. In addition to the implementation-independent information contained in the behavioral model, the structural model holds information about the realization of design decisions from the previous synthesis step, i.e., mapping of the behavioral model onto an architecture template. As such, a structural model is a representation of the resulting architecture as a composition of components that are internally described in the form of behavioral models for input to the next synthesis step. On top of a well-defined combination of MoCs for component-internal behavior and functional semantics, we can hence introduce the term *model of structure* (MoS) for separate classification of such implementation representations and their architectural or structural semantics. Again, a MoS allows characterization of the underlying abstracted semantics of a class of structural models independent of their syntax. Hence, MoSs can be used to compare expressibility and analyzability of specific implementation representations as realized by different tools. For example, at many levels, a netlist concept is used with semantics limited to describing component connectivity. At the system level, pin-accurate models combine a netlist with bus-functional component models. Furthermore, transaction-level modeling (TLM) concepts and techniques are employed to abstract away from pins and wires.[2] Similar to behavioral models, structural models are often represented in a programming language, SLDL, or HDL.

Quality numbers are estimated values for different implementation properties, e.g., throughput, latency, response time, area, and power consumption. In order to get such estimates, synthesis tools often use so-called *performance models* instead of implementing each design option.[3] Performance models represent the contributions of individual elements to overall design quality in a given implementation. Basic numbers are composed based on specific semantics, e.g., in terms of annotation granularity or worst/average/best case assumptions, such that the overall quality estimates can be obtained, e.g., through simulation or static analysis. To distinguish and classify representations of quality numbers across different instances and implementations of performance models, we introduce the concept of an underlying *model of performance* (MoP). A MoP thereby refers to the overall accuracy and granularity in time and space. Generalizing from the detailed definitions of specific performance models, such as timing, power, or cost/area models, a MoP can be used to judge the accuracy of the quality numbers and the computational effort to get them. Examples of simulation-based MoPs for different classes of timing granularity are cycle-accurate performance models (CAPMs), instruction-set-accurate performance models (ISAPMs), or task-accurate performance models (TAPMs) [12]. Quality numbers are often used as objective values during design-space exploration (DSE) when identifying the set of optimal or near-optimal implementations.

Given a specification, the task of *synthesis* then generates an implementation from the specification by *decision making* and *refinement* (Fig. 2). At any level, synthesis is a process of determining the order or mapping of elements in the behavioral model in space and time, i.e., the where and when of their realization. Decision making is hence the task of computing an *allocation* of resources available in the platform model, a spatial *binding* of objects in the behavioral model onto these allocated resources, and a temporal *scheduling* to resolve resource contention of objects in the behavioral model bound to the same resource.

[1]While details of supported architecture features and restrictions, as defined, e.g., by tool database formats, can differ significantly, we limit discussions and comparisons to such high-level MoA classifications in this paper.

[2]Again, many definitions of specific TLM variants exist, but for simplicity, we limit discussions in this paper to a general classification.

[3]We use the term "performance" in the general sense to refer to any measured property.

Refinement is the task of incorporating the made decisions into the behavioral model resulting in a structural model, as discussed earlier. Moreover, with these decisions, a quality assessment of the resulting implementation can be done. The result of this assessment is the quality numbers.

Finally, in order to optimize an implementation, DSE should be performed. As DSE is a multiobjective optimization problem, in general, we will identify a set of optimal implementations instead of a single optimal implementation. For this purpose, the quality numbers provided by the MoP are used. In this paper, we define DSE being the multiobjective optimization problem of the synthesis task. In other words, decision making is the task of calculating a single feasible allocation, binding, and scheduling instance, whereas DSE is the process of finding optimal design points.

In summary, the X-chart shown in Fig. 2 combines two aspects: synthesis (left output) and quality assessment (right output). For both aspects, corresponding so-called Y-charts exist in the literature: The synthesis aspect was presented and later refined into a first system design methodology by Gajski *et al.* in [6] and [13], respectively, while the quality assessment aspect was proposed by Kienhuis *et al.* in [14].

With the earlier discussion, first classification criteria for synthesis tools can be derived.

1) Expressibility and analyzability of the specification.
   a) The MoC of the behavioral model. As, in general, expressibility can be traded against analyzability, the MoC has a huge influence on the automation capabilities of a synthesis tool.
   b) The MoA of the platform model given in the constraints. The MoA, as used for refinement, determines the classes of target implementations supported by a particular tool.
2) Representations of the implementation.
   a) The MoS of the structural model. As structural models are often used for validation and virtual prototyping, the MoS can have a large influence on issues such as simulation performance, observability, and accuracy.
   b) The MoP of the performance model given through the quality numbers. Performance models are employed for quality assessment, and thus, the MoP has large impact on the synthesis quality and estimation accuracy.

As DSE can be performed manually or automatically, an additional classification criterion to be considered is given in the following.

3) Is DSE automated, i.e., does a methodology integrate some multiobjective optimization strategy for decision making?

C. ESL Synthesis

In general, both decision making and refinement can be automated. However, ESL synthesis is a more complex task compared to synthesis at lower levels of abstractions. At any level, the tasks to be performed during decision making and supported during refinement are computing and realizing an *allocation, binding*, and *scheduling*. At ESL, however, these three steps have to be performed for a design space which is at its largest and are required for both computations and communications in the behavioral model. Furthermore, compared to lower levels where refinement is often reduced to producing a simple netlist, generating an implementation of system-level computation and communication decisions is a nontrivial task that requires significant coding effort.

In *computation synthesis, processing elements* (PEs), e.g., processors, hardware accelerators, memories, and IP cores, have to be allocated from the platform model. The resulting allocation has to guarantee that at least each process from the behavioral model can be bound to an allocated PE. A further task in computation synthesis is process binding where each process has to be bound to an allocated PE. A third task in computation synthesis is process scheduling, i.e., a partial/total order is imposed on the processes using a static or dynamic scheduling strategy.

In *communication synthesis, communication elements* (CEs), including busses, point-to-point connections, NoCs, bus bridges, and transducers, have to be allocated. Here, the resulting topology must guarantee that each application communication channel can be bound to an ordered set of architectural communication media and that channel accesses (transactions) can be routed on the CEs. A second task is application channel binding to route application-level communication channels over the allocated architectural network topology. Finally, transactions must be scheduled on the communication media using static time-division access or dynamic, centralized, or distributed arbitration. As is the case in process scheduling, transaction scheduling can result in static, dynamic, or quasi-static schedules.

It should be clearly stated that computation synthesis and communication synthesis are, by no means, independent tasks. Hence, an oversimplified synthesis method might result in infeasible or suboptimal solutions only. Many approaches are heavily biased toward either computation synthesis (e.g., [15] and [16]) or communication synthesis (e.g., [17]–[19]), assuming the counterpart to be done by a different tool. In order to ensure feasibility and optimality, however, an ESL synthesis methodology should support computation and communication synthesis with all their respective subtasks.

As ESL synthesis with its subtasks can be automated in decision making and/or refinement, we now can define additional classification criteria for ESL synthesis tools.

4) Is decision making automated, and if yes, which tasks are automated?
   a) Are computation design decisions computed automatically?
   b) Are communication design decisions computed automatically?
5) Is refinement automated, and if yes, which tasks are performed automatically?
   a) Is computation refinement automatic?
   b) Is communication refinement automatic?

With all the mentioned criteria in this paper, we can classify and compare ESL synthesis tools. In the following sections,

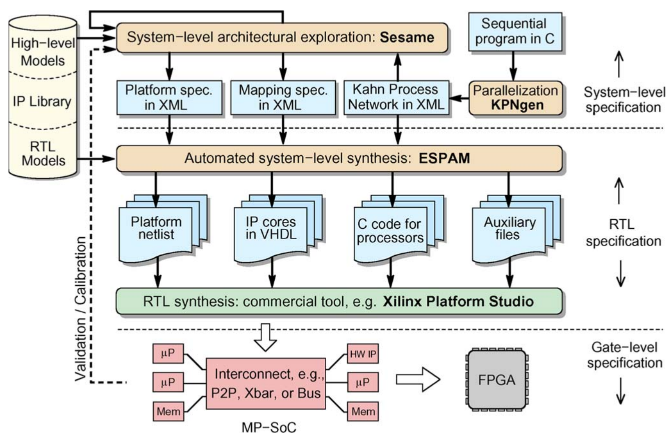
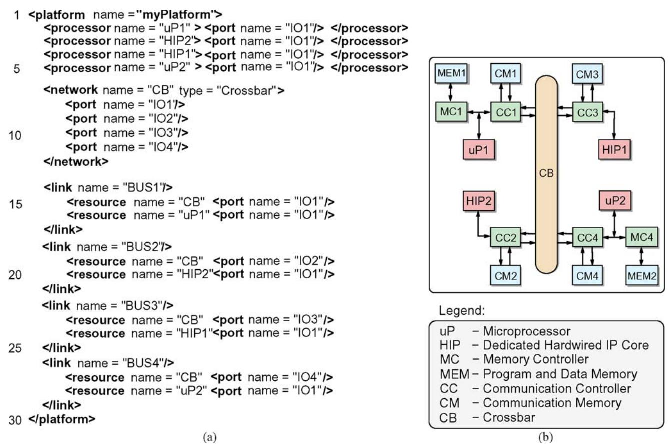


Fig. 3. Daedalus ESL design flow.



```
<platform name ="myPlatform">
    <processor name = "uP1" > <port name = "IO1"/> </processor>
    <processor name = "HIP2"> <port name = "IO1"/> </processor>
    <processor name = "HIP1"> <port name = "IO1"/> </processor>
    <processor name = "uP2" > <port name = "IO1"/> </processor>

    <network name = "CB" type = "Crossbar">
        <port name = "IO1"/>
        <port name = "IO2"/>
        <port name = "IO3"/>
        <port name = "IO4"/>
    </network>

    <link name = "BUS1"/>
        <resource name = "CB"  <port name = "IO1"/>
        <resource name = "uP1" <port name = "IO1"/>
    </link>
    <link name = "BUS2"/>
        <resource name = "CB"  <port name = "IO2"/>
        <resource name = "HIP2"<port name = "IO1"/>
    </link>
    <link name = "BUS3"/>
        <resource name = "CB"  <port name = "IO3"/>
        <resource name = "HIP1"<port name = "IO1"/>
    </link>
    <link name = "BUS4"/>
        <resource name = "CB"  <port name = "IO4"/>
        <resource name = "uP2" <port name = "IO1"/>
    </link>
</platform>
```

Fig. 4. Example of a Daedalus MPSoC platform. (a) Platform specification. (b) Elaborate platform.

we will discuss six ESL synthesis approaches. For all six approaches, we will evaluate their methodologies with respect to these classification criteria. In addition, three ESL synthesis approaches developed in our own groups will be elaborated on in some more detail.

III. Threesome of ESL Methodologies

In this section, we will present three synthesis approaches out of the authors' own research. In addition to classification of underlying methodologies based on previously introduced criteria, this includes details of design steps and experiences resulting from our development and experimental work.

A. Daedalus

Daedalus provides an integrated and highly automated framework for system-level architectural exploration, system-level synthesis, programming, and prototyping of heterogeneous MPSoC platforms [20], [21]. The Daedalus design flow, which is shown in Fig. 3, leads the designer in a number of steps from a sequential application (i.e., behavioral specification) to an MPSoC system implementation on a field-programmable gate array (FPGA) with a parallelized version of the application mapped onto it. This means that Daedalus includes or interfaces with component- and task-level back-end synthesis processes to produce an MPSoC implementation at the RTL and ISA levels for hardware components and software processes, respectively. Since the entire design trajectory can be traversed in only a matter of hours, it offers great potentials for quickly experimenting with different MPSoCs and exploring a variety of design options during the early stages of design.

1) Scope of Methodology: A key assumption for the Daedalus framework is that it considers only dataflow-dominated applications in the realm of multimedia, imaging, and signal processing that naturally contain tasks communicating via streams of data. Such applications are conveniently modeled by means of the Kahn Process Network (KPN) MoC [22]. The KPN MoC we use is a dataflow network of concurrent processes that communicate data in a point-to-point fashion over bounded first-in–first-out (FIFO) channels, using blocking read/write on an empty/full FIFO as synchronization mechanism. The KPNs that Daedalus operates upon can be manually derived or automatically generated. In the latter case, behavioral input specifications are sequential C programs. However, to allow for automatic translation into a KPN, these C applications need to be specified as so-called static affine nested loop programs (SANLPs) [23], which are an important class of programs in, e.g., the scientific and multimedia application domains.

In terms of target MoA, Daedalus considers MPSoC platforms in which both programmable processors and dedicated hardwired IP cores are used as processing components. They communicate data only through distributed memory units. Each memory unit can be organized as one or several FIFOs. The data communication and synchronization between processors are realized by blocking read and write primitives. Such platforms match and support the KPN operational semantics very well, thereby achieving high performance when KPNs are executed on the platforms. In addition, directly supporting the operational semantics of a KPN, i.e., the blocking mechanism, in the target platforms allows the processors to be self-scheduled. This means that there is no need for a global scheduler in the platforms.

Daedalus architectures are constructed from a library of predefined and preverified IP components. These components include a variety of programmable processors, dedicated hardwired IP cores, memories, and interconnects, thereby allowing the implementation of a wide range of heterogeneous MPSoC platforms. Thus, this means that Daedalus aims at *composable MPSoC design*, in which MPSoCs are strictly composed of IP library components. Fig. 4(b) shows a typical example of a Daedalus MPSoC platform. Daedalus produces platforms in the form of synthesizable VHDL (i.e., a netlist MoS) together with the C code for KPN processes that are mapped onto programmable processors. As a consequence, Daedalus designs can be readily mapped on an FPGA for prototyping.

Daedalus supports the mapping of multiple KPN processes onto a single processor. However, it tries to avoid using a multithreading operating system (MTOS) to execute multiple processes on a single processor in order to avoid execution overheads due to context switching. If possible, Daedalus performs compile-time scheduling of the processes that execute on a single processor and thus generates program code for a given processor that does not require an MTOS. However, if finding a compile-time schedule is not possible because of the dynamic (data-dependent) nature of an application, Daedalus uses a very lightweight MTOS to perform runtime scheduling of the processes that execute on a single processor.

The aforementioned design process is guided by automated DSE, which uses a MoP that combines a TAPM and an ISAPM to evaluate design instances. Moreover, Daedalus' computation synthesis trajectory is fully automated, while its communication synthesis is semiautomatic as it uses communication IP components which may need to be customized by hand.

2) Daedalus' Design Steps: As shown in Fig. 3, Daedalus' design flow consists of three key steps, which are implemented by the *KPNgen, Sesame*, and *ESPAM* tools, respectively. *KPNgen* [23] allows for automatically converting a sequential (SANLP) behavioral specification written in C into a concurrent KPN [22] specification. By means of automated source-level transformations, KPNgen is also capable of producing different input–output equivalent KPNs, in which, for example, the amount of concurrence can be varied. Such transformations enable behavioral-level DSE.

The generated or handcrafted KPNs are subsequently used by the Sesame modeling and simulation environment [24] to perform system-level architectural DSE. To this end, Sesame uses (high-level) architecture model components from Daedalus' IP component library (see the left part of Fig. 3). Sesame allows for quickly evaluating the performance of different design decisions in terms of target platform architectures (i.e., resource allocation), binding of KPN processes to architecture resources, and scheduling policies. Here, a balanced tradeoff has been made between simulation accuracy and performance, allowing for extremely fast TAPM-level simulations while still yielding trustworthy estimations. However, on the other hand, Sesame also supports a gradual refinement of its architecture performance models to increase accuracy. This can, for example, be realized by gradually incorporating (external) lower level simulation models, such as cycle-accurate instruction-set simulators, into Sesame's high-level architecture performance models.

Aside from exhaustive simulative DSE to study certain focused regions of a design space, Sesame also supports heuristic search methods, such as genetic algorithms, to steer DSE in larger design spaces. Moreover, it includes an additional design-space pruning step, which is based on analytical models and takes place before DSE to trim the design space that needs to be studied using simulation.

Sesame's DSE results in a set of promising candidate system designs, each of which is described using a high-level XML-based platform description [shown in Fig. 4(a)] and process binding description. These high-level descriptions, together with the (behavioral) KPN description, act as input to the ESPAM tool [25]. This tool subsequently uses RTL versions of the components from the IP library to automatically generate synthesizable VHDL that implements the candidate MPSoC platform architecture. In addition, it also generates the C code for those KPN processes that are mapped onto programmable cores. By using commercial synthesis tools and compilers, this implementation can be readily mapped onto an FPGA for prototyping. Such prototyping also allows for calibrating and validating Sesame's system-level models and thus improves the trustworthiness of these models.

3) Daedalus Experiences: Typically, Daedalus can be deployed in situations where rapid quantitative insight is needed into a variety of different design options during the very early stages of design. For example, Daedalus has recently been used in a case study, together with the Dutch SME Chess B.V. [21], for studying different MPSoC implementations for image compression of very high resolution (medical) images. Hence, Daedalus was used for DSE, both at the level of simulations and prototypes, in order to rapidly gain detailed insight on the system performance. The studied MPSoCs exploit concurrence at three levels: Multiple encoders are operating on different image tiles in parallel, each encoder exploits task parallelism in a pipelined fashion (i.e., streaming), and each encoder exploits data parallelism at the granularity of macroblocks. The complete design space that has been considered in this case study consists of around $2.5 \cdot 10^{13}$ design alternatives, of which only a few hundreds have actually been simulated during the DSE process. By using the DSE results, we selected 25 MPSoC design instances for implementation as FPGA prototypes. The number of PEs in these MPSoC implementations ranges from 1 to 24 processors, where a speedup of 19.7 was obtained for the 24-processor implementation. The encoder application in this case study consists of 2000 lines of C code, while the VHDL for the synthesized MPSoC prototypes ranges from 17 K to 161 K lines of code, dependent on the number of processing cores. Due to the highly automated design flow of Daedalus, all DSE and prototyping work was performed in only a short amount of time, five days in total. Around 70% of this time was taken by the low-level commercial synthesis and place-and-route FPGA tools. The prototype implementations also demonstrated that our DSE phase is not only fast (approximately one entire system-level MPSoC simulation per second) but also capable of accurately predicting the overall system performance: All measured errors were found to be below the 5%, with an average of about 3%.

Daedalus still has a number of restrictions, which will be addressed in the (near) future. For example, the SANLP input requirement for our KPNgen tool needs to be relaxed to allow for automatic parallelization of a wider range of behavioral specifications. Regarding Sesame-based DSE, high-level power models need to be included as well. Furthermore, the platforms studied by Sesame and generated by ESPAM do not include runtime reconfigurable components and do not allow runtime resource management and process binding. This limitation should be relaxed to allow for system-level synthesis of adaptive/reconfigurable MPSoCs that run multiple applications simultaneously with adaptable quality of service.

B. SCE

The system-on-chip environment (SCE) realizes an interactive and automated design flow with a consistent and seamless tool chain all the way from specification down to hardware/software implementation (Fig. 5) [26]. Starting from

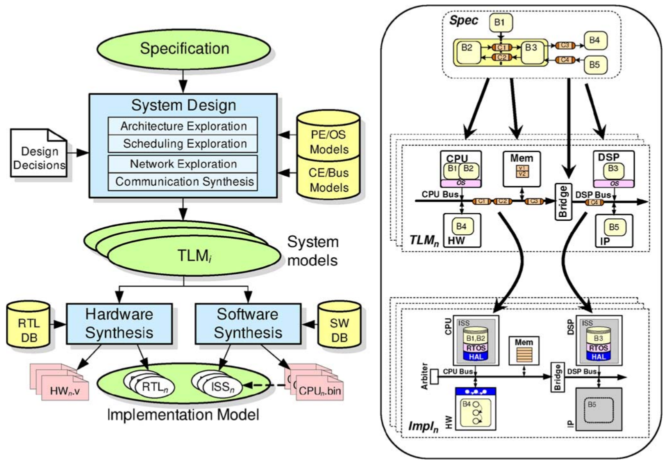


Fig. 5. SCE design flow.

an abstract behavioral specification of the desired system functionality, the SCE ESL synthesis front end allows for interactive user-driven exploration of the system-level design space. Given the design decisions and database components, SCE will automatically implement the specification on the given target platform and, in the process, generate structural TLMs of the system architecture at various levels of abstraction. In a component- and task-level back-end process, hardware and software processors in the TLMs are then individually synthesized further down to their final RTL and ISA implementations, respectively.

SCE is based on the SpecC SLDL and methodology [27]. SpecC technology is standardized and was chosen, for example, by the Japanese Aerospace Exploration Agency (JAXA) as the basis for development of a complete ESL design solution called ELEGANT.[4] ELEGANT is a joint project involving several partners to assemble a common design environment for all of JAXA's suppliers. It includes a derivative of the SCE front end as the core system-level design component [28].

1) Scope of Methodology: At the input of the SCE or ELEGANT design flow, the behavioral system-level specification provides the designer with an abstract high-level model for parallel programming of the platform across hardware and software processors. Computation is specified in a hierarchical and concurrent fashion following a program state machine MoC [13]. SpecC behaviors at the leaves of the hierarchy encapsulate basic algorithms in the form of ANSI C code. Behaviors can be composed hierarchically in arbitrary serial–parallel fashion. At each level, a sequential, parallel, pipelined, or state-machine composition is supported. Behaviors communicate through shared variables or abstract channels. A standard library of communication channels provides a rich set of high-level communication primitives, such as synchronous or asynchronous message passing, queues, events, or semaphores.

ESL refinement tools will then take an input specification and automatically implement it on a given target platform based on a given mapping. Through its PE, CE, and bus databases, SCE supports a system-level MoA that allows for heterogeneous bus-based MPSoCs consisting of PEs, such as custom hardware and programmable software processors, IP blocks, and memories, connected through complex networks of busses and CEs, such as bridges and transducers.

At the output of the ESL design front end, intermediate TLMs represent a system-level MoS that serves as a virtual prototype of the application computation and communication running on the platform processors, memories, and busses. System TLMs automatically generated by SCE integrate high-level task-accurate MoPs (TAPMs) with back-annotated task code running on top of abstract OS and processor models to provide fast, yet accurate, analysis and design validation without the need for slow instruction-set simulation.

At the output of the back end, behavioral hardware and software processor models in the TLM are synthesized down to their component- and task-level implementations ready for further synthesis and manufacturing. On the hardware side, both application algorithms and bus interfaces are refined into synthesizable VHDL or Verilog RTL models. On the software side, the code for application tasks, middleware, and bus drivers is automatically synthesized into final target binaries ready for download into the processors.

In addition to VHDL or Verilog descriptions and binary images for each hardware or software processor, respectively, an implementation model of the system is generated that allows for cosimulation of hardware RTL models with software instruction-set simulators running final target binaries. As a result, the pin- and cycle-accurate implementation model realizes a netlist MoS and a MoP that is based on a CAPM.

2) SCE Design Steps: SCE follows a *Specify-Explore-Refine* methodology [13]. The design process starts from a model specifying the desired functionality (*Specify*). In each following design step, the designer first makes necessary design decisions by exploring the design space (*Explore*). SCE then automatically generates a new model at the next lower level of abstraction by integrating decisions and database component models into the design (*Refine*). As such, through a gradual stepwise refinement process, SCE automatically generates models successively at lower levels of abstraction and with an increasing amount of implementation detail.

SCE integrates all design steps under a common graphical user interface (GUI). The GUI provides interactive and visual design model and database browsing, decision entry, and design analysis. In the exploration phase of each step, users can enter design decisions through the GUI or a command-line scripting interface. To aid the user in the exploration process, SCE includes retargetable profiling and estimation tools that provide a feedback about specification characteristics and effects of decisions on design quality metrics. In addition, SCE supports a plug-in mechanism for inclusion of optimizing algorithms that perform automated decision making.

As shown in Fig. 5, the SCE system design front end internally consists of four design steps: architecture and scheduling exploration for design of system computation, followed by network exploration and communication synthesis for design of system communication.

During architecture exploration, the processing platform (PEs and memories) is defined, and the computational aspects of the specification (behaviors and variables) are mapped

[4]Electronic Design Guidance Tool for Space Use.

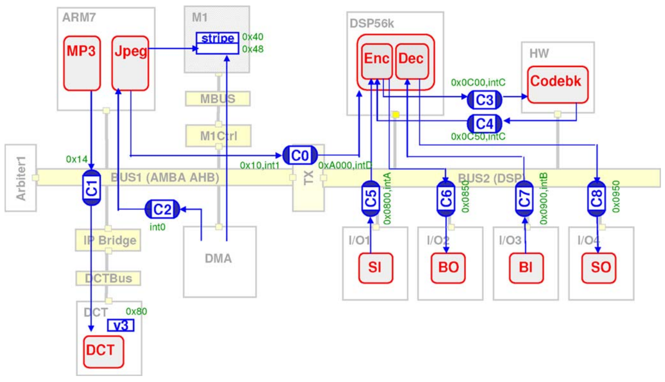


Fig. 6. SCE cellphone design example.

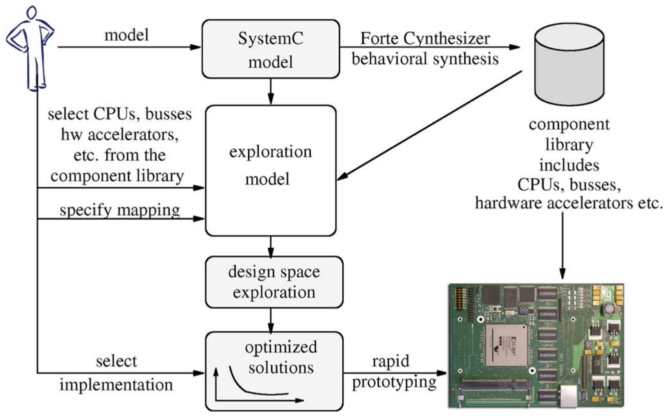

model
SystemC model
Forte Cynthesizer behavioral synthesis
component library includes CPUs, busses, hardware accelerators etc.
select CPUs, busses hw accelerators, etc. from the component library
exploration model
specify mapping
design space exploration
optimized solutions
select implementation
rapid prototyping

Fig. 7. ESL design flow using SystemCoDesigner.

onto that platform. During scheduling exploration, the order of execution on the inherently sequential PEs is determined. Behaviors can be statically scheduled and grouped into sequential tasks, and remaining concurrent tasks are dynamically scheduled on top of an RTOS.

During network exploration, the system communication topology (busses, CEs, and their connectivity) is defined, and the given end-to-end communication channels are mapped and routed over that network. During communication synthesis, point-to-point links in each network segment are implemented over the actual bus medium, and pin- and bit-accurate parameters, such as bus addresses and interrupts, are selected.

Finally, in the back end, the hardware and software synthesis of each synthesizable or programmable PE and CE is performed. Hardware synthesis follows an interactive and automated high-level synthesis process to take behavioral hardware models down to structural RTL descriptions. For software synthesis, SpecC code for application software, middleware, drivers, and interrupt handlers is generated, cross-compiled, and targeted toward and linked against RTOS to create final target binaries.

3) SCE Experiences: SCE has been applied to a large suite of industrial-size design examples. Fig. 6 shows an example design of a cellphone baseband MPSoC that combines an MP3 decoder and JPEG encoder running on an ARM subsystem with a GSM voice encoder/decoder running on a Motorola DSP. Subsystems include memories and I/O peripherals and are assisted by custom hardware PEs for DCT and codebook search acceleration. The complete cellphone specification consists of about 16 000 lines of SpecC code and is refined down to 30 000 lines in the final TLM.

For all investigated examples, several different design alternatives were explored. Given design decisions, final system TLMs are automatically refined by SCE within seconds, translating into productivity gains of several orders of magnitude compared to a tedious and error-prone manual model writing process. Furthermore, generated simulation models provide fast and accurate feedback. Complete MPSoC TLMs simulate at a speed of about 600 MIPS sustained and up to 2000 MIPS peak. Depending on back annotation of profiling or trace-based estimates, timing errors range from 12.5% down to an average of 3%. In all cases, however, models exhibit 100% fidelity. Together, automatic model generation paired with fast and accurate simulation enables rapid early DSE. For example, in a case study of a stand-alone MP3 decoder on a Xilinx platform (MicroBlaze CPU plus OPB bus), interactive exploration of more than ten alternatives led to an optimal architecture in less than an hour, including generation and simulation of all models at a rate of two to four models per minute.

As part of the ELEGANT project, JAXA initiated a variety of evaluations of the resulting tool environment in several of JAXA's suppliers and other independent investigators. For example, with SCE at its core, a single SpaceWire[5] specification could be automatically realized as both a pure hardware solution and a mixed hardware/software implementation. Both variants were successfully synthesized and validated to conform to protocol specifications. In another evaluation, an MPEG4 decoder was implemented on a MIPS-based platform with varying levels of hardware acceleration. Good quality of results could be observed for all automatically synthesized hardware, achieving a 30-frames/s decoding rate on an 80-MHz three-processor architecture.

With automatic refinement from specification down to implementation, the development of the initial specification model becomes the major bottleneck. Even though a C-based design allows the reuse of a large body of existing legacy code, the conversion of often unstructured C code into a parallelized specification remains a challenge. As such, further research into tool support for automation of specification capture or conversion from other high-level models, such as Matlab or UML, is needed in the future.

C. SystemCoDesigner

The goal of the SystemCoDesigner project is to automatically map applications written in SystemC to a heterogeneous MPSoC platform. By automating as many design steps as possible, an early evaluation of different design options is permitted [29]. The overall design flow is shown in Fig. 7. In a first step, the designer writes an actor-oriented application model using SystemC. In a second step, different hardware accelerators are automatically generated for actors and stored in a component library. This library also contains other

[5]A standard for high-speed and highly reliable networks in space and satellite applications.

synthesizable IP cores like processors, busses, or memories. The designer defines an MPSoC platform model from resources in the component library as well as mapping constraints for the actors, resulting in a system-level specification. An automatic DSE trades off several, often conflicting, design objectives. From the set of optimized solutions, the designer selects promising implementations for rapid prototyping. For this purpose, design decision leading to the optimized solution is represented as structural TLM. For rapid prototyping, hardware accelerators are synthesized to the RTL, and software is compiled to match the ISA of selected processors.

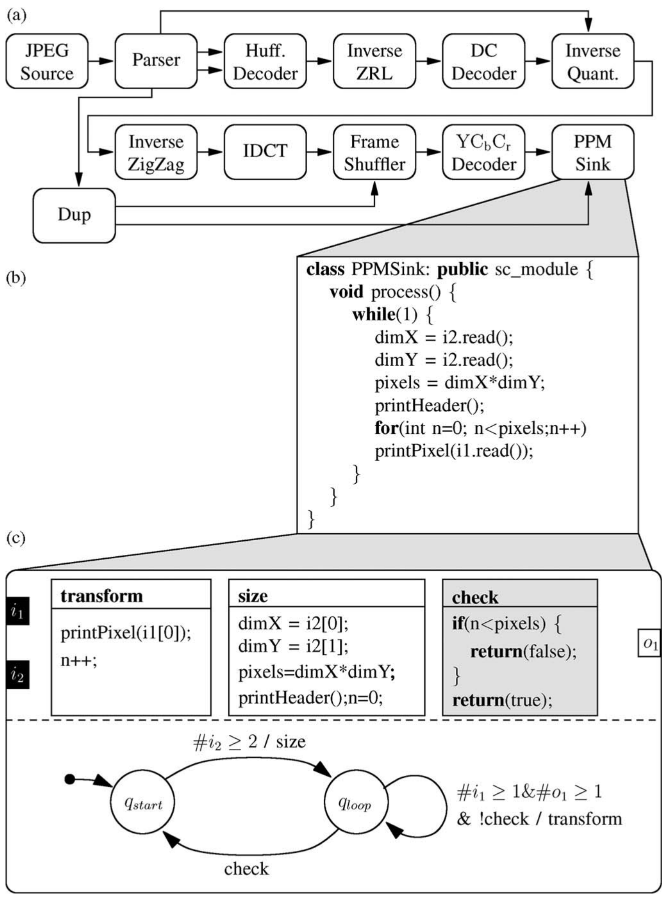
1) Scope of Methodology: Currently, SystemCoDesigner supports the design of streaming applications. These applications are typically modeled by the help of dataflow graphs where vertices represent actors and edges represent data dependences. Due to the complexity of many streaming applications, they often cannot be modeled as static dataflow graphs [30], [31], where consumption and production rates are known at compile time. Rather, they are described as a combination of static and dynamic dataflow (DDF) models, e.g., KPNs [22].

On the other hand, SystemC [32] is becoming a new defacto standard in industrial system-level design flows. Hence, SystemCoDesigner assumes that the application model is written in SystemC and represents a dataflow model, i.e., SystemC modules (actors) only communicate via SystemC FIFO channels and their functionality is implemented in a single SystemC thread. Such input descriptions can be transformed into a special subset of SystemC called SysteMoC [29]. An application modeled in SysteMoC resembles the *FunState* MoC (functions driven by state machines) [33] that allows one to express nondeterministic DDF models.

A SysteMoC model is composed of SysteMoC actors that communicate via queues with FIFO semantics. Each SysteMoC actor is defined by a finite state machine (FSM), specifying the communication behavior and methods controlled by the FSM. If activated by the FSM, these methods are executed atomically, and data consumption and production is only performed after computing a method.

As an example, Fig. 8(a) shows a Motion-JPEG decoder in SysteMoC. It consists of several actors interconnected by communication channels (edges) processing a stream of data. Fig. 8(b) exemplarily shows the SystemC definition of the PPM sink actor. The corresponding representation as SysteMoC actor is shown in Fig. 8(c). The FSM controlling the communication behavior of the SysteMoC actor checks for available input data (e.g., $\#i_1 \geq 1$) and available space on the output channels (e.g., $\#o_1 \geq 1$) to store results. Furthermore, constant methods called *guards* (e.g., `check`) can be used to test values of internal variables and data in the input channels. If predicates annotated to a state transition evaluate to true, this transition can be taken, and annotated *action* methods (e.g., `transform`) will be processed atomically.

SysteMoC actors can be transformed into both hardware accelerators and software modules [29]. The latter one is achieved by straightforward code transformations, whereas the hardware accelerators are built by the help of Forte Cynthesizer [9]. This allows for quick extraction of important performance parameters like the achieved throughput and the required area



```
class PPMSink: public sc_module {
  void process() {
    while(1) {
      dimX = i2.read();
      dimY = i2.read();
      pixels = dimX*dimY;
      printHeader();
      for(int n=0; n<pixels;n++)
      printPixel(i1.read());
    }
  }
}
```

```
transform
printPixel(i1[0]);
n++;
```

```
size
dimX = i2[0];
dimY = i2[1];
pixels=dimX*dimY;
printHeader();n=0;
```

```
check
if(n<pixels) {
  return(false);
}
return(true);
```

Fig. 8. (a) Block diagram of a Motion-JPEG decoder. (b) SystemC code of an actor that can be transformed into a SysteMoC actor given in (c).

which are used to calibrate the system-level specification. The generated hardware accelerators (synthesizable RTL code) are stored in the component library. This component library contains further synthesizable IP cores, including processors, busses, memories, etc. The MoA is a heterogeneous MPSoC platform which is specified by instantiating and connecting cores from the component library. Furthermore, the designer has to specify mapping constraints for each SysteMoC actor. Later, DSE is performed to find sets of optimized solutions.

From the set of optimized solutions, the designer selects any MPSoC implementation best suited for his needs. Once this selection has been made, the last step of the proposed ESL design flow is the rapid prototyping of the corresponding FPGA-based implementation in terms of model refinement. For this purpose, the resulting platform is assembled. Moreover, the program code for each processor is generated according to the binding of the actors. This results in a TLM, which is the MoS used as implementation representation by SystemCoDesigner. In order to generate high-quality software schedules, SystemCoDesigner supports the automatic classification of actors into synchronous or cyclo-static dataflow [34] and clustering static actors bound to the same processor into a single dynamic actor [35]. Finally, the implementation is compiled into an FPGA bit stream using the Xilinx Embedded Development Kit [36]. Thereby, connecting SystemCoDesigner to lower abstraction levels in the double roof model.

2) SystemCoDesigner Design Steps: All manual work in the SystemCoDesigner design flow has been performed after setting up the MPSoC platform model together with the mapping constraints. Starting with this input model, SystemCoDesigner automatically explores the design space. For this purpose, it optimizes the implementation of the streaming application while considering several objectives simultaneously, e.g., latency, throughput, area, and power consumption. While area consumption is assumed to be a linear cost function, timing and power estimation requires a simulation-based performance evaluation during exploration.

SystemCoDesigner generates task-accurate MoPs (TAPM) automatically from the SysteMoC model, and the performance values were annotated in the input model [29]. For this purpose, the MPSoC platform model is translated into a so-called *virtual architecture* using again SystemC. The performance evaluation is done by linking the SysteMoC model to the virtual architecture. Each invocation of an action of an actor is then relayed to the virtual component the actor is bound to. The virtual component then blocks the actor's execution until the estimated execution time of the action and possible other preemption times are expired.

Aside from evaluating a single design point, DSE is responsible for covering the search space. In order to perform decision making automatically, SystemCoDesigner translates the input model into a pseudo-Boolean (PB) formula. The variables of this formula encode the resource allocation, the actor binding, the queue mapping, and the routing of transactions on the communication structure. Each variable assignment satisfying this formula corresponds to a feasible implementation of the application. A PB solver is used to identify these solutions [29]. The optimization is performed using a multiobjective evolutionary algorithm.

3) SystemCoDesigner Experiences: For the experimental evaluation of the SystemCoDesigner design flow, a Motion-JPEG decoder, as shown in Fig. 8(a), has been implemented. The Motion-JPEG decoder case study consists of 8000 SysteMoC lines of code, supporting interleaved and noninterleaved baseline profiles without subsampling. The complete specification results in about $5 \cdot 10^{33}$ possible implementation alternatives. Owing to the integration of Forte Cynthesizer, the hardware accelerators for the different actors could be obtained directly from the SysteMoC specification. Furthermore, as SysteMoC offers a higher level of abstraction compared to RTL, the designer can progress more quickly. Taking the number of lines of code as a measure for complexity, the RTL design would have been eight to ten times more costly.

With the specification, the design space has been explored using SystemCoDesigner. The objectives taken into account during DSE have been the following: 1) throughput; 2) latency; 3) number of required flip-flops; 4) lookup tables (LUTs); and 5) block random access memories (BRAMs). During exploration, 7600 different solutions have been evaluated in two days, 17 h, and 46 min. The simulation time per solution is about 30 s for Motion-JPEG streams consisting of four QCIF frames. As a result, 366 nondominated solutions were found, each of them representing an arbitrary hardware/software implementation. Hardware-only implementations show real-time performance ($\geq$ 25 frames/s) for QCIF streams while occupying about 40 000 four-input LUTs and 14 500 flip-flops.

Finally, many of these solutions have been automatically prototyped onto a Xilinx Virtex II FPGA. However, a discrepancy of up to 30% can be identified when comparing the FPGA implementations with the performance estimations during DSE. The differences in the required hardware sizes ($\leq$ 15%) occurring between the predicted values and those measured in hardware can be explained by postsynthesis optimization like elimination of useless BRAMs. The discrepancy between the performance estimations for latency and throughput and those measured for hardware–software solutions is due to schedule overhead.

IV. Other ESL Synthesis Methodologies

In the following, we will present three more related academic approaches. Note that in contrast to our own work for which we have additional details available, discussion of other related work is limited to a classification of their underlying methodologies based on the criteria introduced in Section II.

A. Metropolis

Metropolis [37] is a modeling and simulation environment based on the platform-based design (PBD) paradigm [38]. PBD is an attempt at simplifying the system-level design problem by removing one degree of freedom: In PBD, the allocation of the target system platform consisting of computation and communication components is assumed to be given or at least significantly constrained. As such, the constraints at the input of the design process contain a fixed architecture template with no or little flexibility. Such a predefined and predetermined platform facilitates the reuse of common design patterns across different design instances. Therefore, PDB follows a meet-in-the-middle approach, and the system design problem is reduced to the mapping of a desired function onto the given target platform to create a specific design instance.

Metropolis provides a general proprietary metamodel language that is used to capture separate models for "functionality" (behavioral model), "architecture" (platform model), and their "mapping" (binding and scheduling). The metamodel employs a fundamental event-based execution model with concepts of concurrent processes communicating through channels (called media), including associated constraints and quantities. In a similar manner to other system-level languages, functionality is described in the form of event-driven process networks that are general in the sense that many classes of MoCs can be represented. In addition, functionality can be annotated with nonfunctional constraints. The architecture is defined following an MoA that uses processes and media to describe available resources (e.g., tasks) and services (e.g., CPUs, memories, or busses), respectively. Quantities can be associated with the architecture to define a MoP at the level of tasks (TAPM). Finally, given a specification in the form of functionality and architecture, synthesis or refinement is performed by defining a MoS as a mapping between the two through a set of additional constraints synchronizing their event execution.

Metropolis itself does not define any specific design tools but rather a general framework and language for modeling with support for simulation, validation, and analysis of models. Metropolis includes a front end for parsing of metamodels and a back end for translation of metamodels into C++/SystemC simulation code. In addition, several back-end point tools have emerged for scheduling, communication design, verification, and hardware synthesis [39].

B. Koski

The Koski design flow [40] provides a single infrastructure for modeling of applications; automatic architectural DSE; and automatic ESL synthesis, programming and prototyping of selected MPSoCs. Koski's design flow starts with the capturing of requirements for an application and architecture, including design constraints, such as the overall maximum cost. Subsequently, the functionality of the system is described with an application model in a UML design environment (using the Statecharts MoC to describe the actual functionality) and verified with functional simulations. The architecture model consists of components which are taken from a platform library, targeting the construction of heterogeneous bus-based MPSoCs (MoA). The relationship between application and architecture models is described with a mapping model.

The UML interface handles the transformation of application and architecture models to an abstracted model for fast architecture exploration. Particularly, the application model is transformed to an abstract process network model. In addition, the UML interface can back-annotate the UML design with performance information obtained from lower level simulations. Finding a good application-to-architecture mapping is carried out during a two-phase automatic architecture exploration step consisting of static and dynamic (i.e., simulative) exploration methods using a TAPM MoP. For controlling the architecture exploration, the designer constrains the design space by defining the platform parts that can be used as well as the allowed mapping combinations. In addition, the designer specifies the constraints for performance, area, and power.

In the last step, the parts of the UML description that were mapped to processors during the architecture exploration are passed to the automatic code generation. The generated low-level software code and the RTL descriptions (i.e., a netlist MoS) of the component instances from the platform (derived from Koski's platform library) are then combined for physical implementation. This stage also handles the RTOS integration, software executable generation, and hardware synthesis.

C. PeaCE/HOPES

PeaCE (Ptolemy extension as a Codesign Environment) [41] is an ESL synthesis framework for multimedia applications. Starting from a Ptolemy II application model, it provides a seamless codesign flow from functional simulation to system synthesis and prototyping. Although Ptolemy supports the hierarchical combination of many different MoCs, PeaCE restricts the input model to extension of synchronous dataflow and extended FSMs. In PeaCE, the application is modeled by a task graph where tasks are either signal processing tasks or control tasks. Signal processing tasks are modeled through synchronous piggybacked dataflow, a dataflow model with control token. Control tasks are modeled by flexible FSMs (hierarchical state machines without state transitions crossing hierarchy boundaries).

For functional simulation of the application model, PeaCE provides an automatic C code generation. For system synthesis, the architecture platform is specified by a list of processors and synthesizable IP cores, resulting in a heterogeneous MPSoC MoA. The DSE is two-phased: In a first step, the resource allocation and task binding are performed. During this step, communication overhead is assumed to be proportional to the amount of consumed and produced data. The objective of this step is to minimize system cost under timing constraints. In the second step, the communication architecture exploration, which is bus and memory allocation, is performed. For this purpose, communication and memory traces are generated for those solutions fulfilling the timing constraints in the first step. DSE in PeaCE can be performed automatically or manually and is guided by an ISAPM. After DSE, optimized MPSoC implementations can be prototyped either using a cosimulation environment or FPGAs. In both cases, the MoS is a Netlist representing the design decisions.

Recently, a new framework called HOPES has been proposed as an enhancement to PeaCE [42]. The main focus is on generating MPSoC software and overcoming the limitations of OpenMP and MPI. Its input model is called common intermediate code (CIC). A CIC model consists of two parts: The task code defines each task by the three methods init(), go(), and wrapup(). Intertask communication or communication to the environment is established by the help of several APIs. The second part is the architecture information, including the platform definition and additional constraints. The task code of a CIC model can be either written manually or automatically generated from PeaCE models.

A CIC translator transforms a CIC model into an optimized software for the processors in the MPSoC platform. For this purpose, the API calls must be replaced by a platform-specific code, interface code for hardware accelerators has to be generated, and scheduling of tasks bound to the same processor has to be performed. Optionally, an OpenMP compiler can be used for optimization.

V. Discussion

A summary of all six presented tools based on the classification criteria introduced in Section II is given in Table I. In this table, a full circle implies that a certain synthesis aspect (DSE, decision making, or refinement) is taken care of in a fully automated fashion by an ESL synthesis approach, while an open circle means partial support/automation.

As can be seen, tools share many common characteristics. For example, all discussed tools target heterogeneous bus-based MPSoCs and almost uniformly support task-based performance models. On the other hand, each tool has its particular strengths and weaknesses, specifically in the level of automation for different design tasks. All together, this provides a tremendous

TABLE I
CLASSIFICATION OF DIFFERENT ESL SYNTHESIS APPROACHES

	Specification		Implementation			Decision Making		Refinement	
Approach	MoC[(1a)]	MoA[(1b)]	MoS[(2a)]	MoP[(2b)]	DSE[(3)]	Comp[(4a)]	Comm[(4b)]	Comp[(5a)]	Comm[(5b)]
Daedalus	KPN	HeMPSoC	Netlist	T/ISAPM	●	●	○	●	○
Koski	Statecharts	HeMPSoC	Netlist	TAPM	●	●	○	●	○
Metropolis	PN	HeMPSoC	TLM	TAPM		○		○	
PeaCE/HoPES	DDF/FSM	HeMPSoC	Netlist	ISAPM	○	○		●	○
SCE	PSM	HeMPSoC	TLM/Netlist	T/CAPM				●	●
SystemCoDesigner	DDF	HeMPSoC	TLM	TAPM	●	●	●	●	

DDF	Dynamic Dataflow	HeMPSoC	Heterogeneous, Bus-Based Multi-Processor System-On-Chip	TAPM	Task Accurate Performance Model
(K)PN	(Kahn) Process Network			ISAPM	Instruction Set Accurate Performance Model
PSM	Program State Machine	TLM	Transaction-Level Model	CAPM	Cycle-Accurate Performance Model

opportunity to exploit tool synergies. By merging automation capabilities of different tools, a complete ESL synthesis solution should be achievable. We are currently in the process of exploring such integration of our own tools, e.g., by combining DSE and decision-making algorithms of SystemCoDesigner with SCE's refinement engine.

One of the biggest hurdles for tool interoperability will always remain the definition of proper standardized interfaces. As part of our integration work, we expect to obtain insights into requirements for such interfaces, e.g., for a canonical design decision description format between decision making and refinement. Another open question is the choice of MoC at the specification level. While restricted MoCs show the potential to perform domain-specific optimizations, other more general MoCs should be used for expressing implementation details and even conducting platform-dependent optimization steps. As both aspects are important ingredients for ESL synthesis tools, a well-defined MoC hierarchy and MoC interoperability might help to improve future design methodologies at the system level.

On the modeling side, language and MoS standardization efforts such as SpecC or SystemC consortia, TLM standards, and the IP-XACT netlist format are only a first step into this direction. As exemplified by the various tools presented in this paper, standardized languages can provide a common basis for exchange of design models between different point tools and design steps, even across different vendors as demonstrated by the SCE/ELEGANT project. However, experiences from these projects also showed that synthesis nevertheless requires tight integration for exchange of semantic metainformation on top of basic inherently ambiguous simulation languages.

In general, interoperability issues will require an industry-wide approach. In this sense, it may be worthwhile to consider the definition and development of a common design flow infrastructure (CDFI) which facilitates the development of system-level design flows and fosters the reuse of design tools. Such a CDFI would be a kind of metatool for developing system-level design flows, having design flow steps as plug-ins, i.e., similar to the goals of the Metropolis project. This requires the definition (and broad adoption) of standardized tool, model, and data descriptions and file formats to allow the interchange of information between the CDFI framework and external tools (i.e., plug-ins). Moreover, the framework could also allow for explicitly defining design flows, which would make it possible to build prepackaged standardized or customized design flows.

Finally, the synergy between the various ESL synthesis efforts also necessitates the development of standard case studies and benchmarks for ESL design. This would invigorate ESL synthesis research as it enables the direct comparison of research results. Currently, such a comparison between ESL synthesis research efforts in terms of their qualitative characteristics remains difficult. We also believe that the flow of ideas from academia to industry will benefit from good standardized benchmarks and case studies, as research results can always be demonstrated on industrially relevant examples.

VI. SUMMARY AND CONCLUSION

Being an active research topic at its relative infancy, the ESL space is, as of yet, characterized by fragmentation and partial or wrongly positioned solutions. In this paper, we have developed and proposed a classification framework for evaluation of different ESL synthesis approaches. Within the context of this framework, we presented a comparison and analysis of six different state-of-the art ESL tools. These observations show that recent approaches are converging toward largely similar design principles and flows. Nevertheless, no single approach currently provides a complete solution, and further research in many areas is required. On the other hand, based on the common concepts and principles identified in this classification, it should be possible to define interfaces such that different point tools can be combined into an overall ESL design environment. In the future, we plan to investigate such interoperability issues using combinations of different tools presented in this paper.

ACKNOWLEDGMENT

Aside from the authors, a large number of people are responsible for, or have contributed to, the work described in Section III of this paper. The main cocontributors of the Daedalus framework are H. Nikolov, M. Thompson, C. Erbas, S. Polstra, and E. Deprettere. For SCE, the authors would like to thank the main developers, namely, R. Dömer, J. Peng, D. Shin, and Q.-V. Dang. The main cocontributors to the SystemCoDesigner framework are J. Falk, J. Gladigau, M. Glaß, J. Keinert, M. Lukasiewycz, T. Schlichter, M. Streubühr, and C. Zebelein. Last, but not least, the authors would like to thank the reviewers for their helpful comments and suggestions in making this paper a much stronger contribution.

References

[1] G. Martin, "Overview of the MPSoC design challenge," in *Proc. DAC*, San Francisco, CA, Jul. 2006, pp. 274–279.

[2] D. Densmore, R. Passerone, and A. Sangiovanni-Vincentelli, "A platform-based taxonomy for ESL design," *IEEE Des. Test Comput.*, vol. 23, no. 5, pp. 359–374, Sep./Oct. 2006.

[3] E. A. Lee and A. Sangiovanni-Vincentelli, "A framework for comparing models of computation," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 17, no. 12, pp. 1217–1229, Dec. 1998.

[4] W. Wolf, A. A. Jerraya, and G. Martin, "Multiprocessor system-on-chip (MPSoC) technology," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 27, no. 10, pp. 1701–1713, Oct. 2008.

[5] J. Teich, "Embedded system synthesis and optimization," in *Proc. Workshop SDA*, Rathen, Germany, Mar. 2000, pp. 9–22.

[6] D. D. Gajski and R. H. Kuhn, "New VLSI tools," *Computer*, vol. 16, no. 12, pp. 11–14, Dec. 1983.

[7] C. Zhu, Z. P. Gu, R. P. Dick, and L. Shang, "Reliable multiprocessor system-on-chip synthesis," in *Proc. CODES+ISSS*, 2007, pp. 239–244.

[8] S. Pasricha, N. Dutt, E. Bozorgzadeh, and M. Ben-Romdhane, "Fabsyn: Floorplan-aware bus architecture synthesis," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 14, no. 3, pp. 241–253, Mar. 2006.

[9] [Online]. Available: http://www.forteds.com

[10] NEC System Technologies, Ltd., *CyberWorkBench*. [Online]. Available: http://www.necst.co.jp/product/cwb

[11] B. Kienhuis, E. Deprettere, P. van der Wolf, and K. Vissers, "A methodology to design programmable embedded systems," in *Embedded Processor Design Challenges: Systems, Architectures, Modeling, and Simulation (SAMOS)*, vol. 2268. New York: Springer-Verlag, 2002, pp. 18–37.

[12] M. Gries, "Methods for evaluating and covering the design space during early design development," *Integr. VLSI J.*, vol. 38, no. 2, pp. 131–183, Dec. 2004.

[13] D. D. Gajski, F. Vahid, S. Narayan, and J. Gong, *Specification and Design of Embedded Systems*. Englewood Cliffs, NJ: Prentice–Hall, 1994.

[14] B. Kienhuis, E. Deprettere, K. Vissers, and P. van der Wolf, "An approach for quantitative analysis of application-specific dataflow architectures," in *Proc. IEEE Int. Conf. Appl.-Specific Syst., Architectures Processors*, Zurich, Switzerland, Jul. 1997, pp. 338–349.

[15] K. Huang, S. Han, K. Popovici, L. Brisolara, X. Guerin, L. Li, X. Yan, S. Chae, L. Carro, and A. A. Jerraya, "Simulink-based MPSoC design flow: Case study of Motion-JPEG and H.264," in *Proc. DAC*, 2007, pp. 39–42.

[16] G. Stitt and F. Vahid, "Binary synthesis," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 12, no. 3, pp. 1–30, Aug. 2007.

[17] K. Lahiri, A. Raghunathan, and S. Dey, "Design space exploration for optimizing on-chip communication architectures," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 23, no. 6, pp. 952–961, Jun. 2004.

[18] F. Dumitrascu, I. Bacivarov, L. Pieralisi, M. Bonaciu, and A. A. Jerraya, "Flexible MPSoC platform with fast interconnect exploration for optimal system performance for a specific application," in *Proc. DATE Designers' Forum*, 2006, pp. 166–171.

[19] S. Pasricha and N. Dutt, "A framework for co-synthesis of memory and communication architectures for MPSoC," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 26, no. 3, pp. 408–420, Mar. 2007.

[20] M. Thompson, T. Stefanov, H. Nikolov, A. D. Pimentel, C. Erbas, S. Polstra, and E. F. Deprettere, "A framework for rapid system-level exploration, synthesis, and programming of multimedia MP-SoCs," in *Proc. CODES+ISSS*, 2007, pp. 9–14.

[21] H. Nikolov, M. Thompson, T. Stefanov, A. D. Pimentel, S. Polstra, R. Bose, C. Zissulescu, and E. F. Deprettere, "Daedalus: Toward composable multimedia MP-SoC design," in *Proc. DAC*, Jun. 2008, pp. 574–579.

[22] G. Kahn, "The semantics of a simple language for parallel programming," in *Proc. IFIP Congr.*, 1974, pp. 471–475.

[23] S. Verdoolaege, H. Nikolov, and T. Stefanov, "PN: A tool for improved derivation of process networks," *EURASIP J. Embed. Syst.*, vol. 2007, no. 1, p. 19, Jan. 2007. Article ID 75947.

[24] A. D. Pimentel, C. Erbas, and S. Polstra, "A systematic approach to exploring embedded system architectures at multiple abstraction levels," *IEEE Trans. Comput.*, vol. 55, no. 2, pp. 99–112, Feb. 2006.

[25] H. Nikolov, T. Stefanov, and E. F. Deprettere, "Systematic and automated multi-processor system design, programming, and implementation," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 27, no. 3, pp. 542–555, Mar. 2008.

[26] R. Dömer, A. Gerstlauer, J. Peng, D. Shin, L. Cai, H. Yu, S. Abdi, and D. D. Gajski, "System-on-chip environment: A SpecC-based framework for heterogeneous MPSoC design," *EURASIP J. Embed. Syst.*, vol. 2008, no. 3, pp. 1–13, Jan. 2008.

[27] D. D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, and S. Zhao, *SpecC: Specification Language and Design Methodology*. Norwell, MA: Kluwer, 2000.

[28] A. Gerstlauer, J. Peng, D. Shin, D. Gajski, A. Nakamura, D. Araki, and Y. Nishihara, "Specify-explore-refine (SER): From specification to implementation," in *Proc. DAC*, Anaheim, CA, Jun. 2008, pp. 586–591.

[29] J. Keinert, M. Streubühr, T. Schlichter, J. Falk, J. Gladigau, C. Haubelt, J. Teich, and M. Meredith, "SystemCoDesigner—An automatic ESL synthesis approach by design space exploration and behavioral synthesis for streaming applications," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 14, no. 1, pp. 1–23, Jan. 2009.

[30] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," *Proc. IEEE*, vol. 75, no. 9, pp. 1235–1245, Sep. 1987.

[31] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete, "Cyclo-static dataflow," *IEEE Trans. Signal Process.*, vol. 44, no. 2, pp. 397–408, Feb. 1996.

[32] T. Grötker, S. Liao, G. Martin, and S. Swan, *System Design With SystemC*. Norwell, MA: Kluwer, 2002.

[33] K. Strehl, L. Thiele, M. Gries, D. Ziegenbein, R. Ernst, and J. Teich, "FunState—An internal design representation for codesign," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 9, no. 4, pp. 524–544, Aug. 2001.

[34] C. Zebelein, J. Falk, C. Haubelt, and J. Teich, "Classification of general data flow actors into known models of computation," in *Proc. MEMOCODE*, Anaheim, CA, Jun. 2008, pp. 119–128.

[35] J. Falk, J. Keinert, C. Haubelt, J. Teich, and S. Bhattacharyya, "A generalized static data flow clustering algorithm for MPSoC scheduling of multimedia applications," in *Proc. EMSOFT*, Atlanta, GA, Oct. 2008, pp. 189–198.

[36] *Embedded SystemTools Reference Manual—Embedded Development Kit EDK 8.1ia*, XILINX, San Jose, CA, Oct. 2005. [Online]. Available: http://www.xilinx.com/ise/embedded/est_rm.pdf

[37] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli, "Metropolis: An integrated electronic system design environment," *Computer*, vol. 36, no. 4, pp. 45–52, Apr. 2003.

[38] A. Sangiovanni-Vincentelli, "Quo vadis SLD: Reasoning about the trends and challenges of system level design," *Proc. IEEE*, vol. 95, no. 3, pp. 467–506, Mar. 2007. [Online]. Available: http://chess.eecs.berkeley.edu/pubs/263.html

[39] Gigascale Systems Research Center (GSRC), *Core Design Technology for Complex Heterogeneous Systems*. [Online]. Available: http://www.gigascale.org/theme/core/

[40] T. Kangas, P. Kukkala, H. Orsila, E. Salminen, M. Hännikäinen, T. D. Hämäläinen, J. Riihimäki, and K. Kuusilinna, "UML-based multiprocessor SoC design framework," *ACM Trans. Embed. Comput. Syst.*, vol. 5, no. 2, pp. 281–320, May 2006.

[41] S. Ha, S. Kim, C. Lee, Y. Yi, S. Kwon, and Y.-P. Joo, "PeaCE: A hardware-software codesign environment of multimedia embedded systems," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 12, no. 3, pp. 1–25, Aug. 2007.

[42] S. Kwon, Y. Kim, W.-C. Jeun, S. Ha, and Y. Paek, "A retargetable parallel programming framework for MPSoC," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 13, no. 3, pp. 1–18, Jul. 2008.

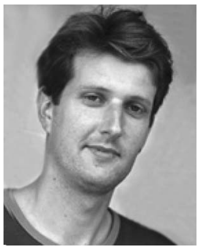
Andreas Gerstlauer (S'97–M'04) received the Dipl.-Ing. degree in electrical engineering from the University of Stuttgart, Stuttgart, Germany, in 1997, and the M.S. and Ph.D. degrees in information and computer science from the University of California, Irvine (UCI), in 1998 and 2004, respectively.

Since 2008, he has been with the University of Texas, Austin, where he is currently an Assistant Professor in electrical and computer engineering. Prior to joining the University of Texas, he was an Assistant Researcher with the Center for Embedded Computer Systems, UCI, leading a research group to develop electronic system-level design tools. His research interests include system-level design automation, system modeling, design languages and methodologies, and embedded hardware and software synthesis.

Dr. Gerstlauer serves on the program committee of major conferences such as DATE and CODES+ISSS.

Christian Haubelt (M'02) received the Diploma degree in electrical engineering from the University of Paderborn, Paderborn, Germany, in 2001 and the Ph.D. degree in computer science from the Friedrich-Alexander-University of Erlangen–Nuremberg, Erlangen, Germany, in 2005.

He currently leads the System-Level Design Automation Group, Department of Hardware-Software-Co-Design, University of Erlangen–Nuremberg. He serves as a Reviewer for several well-known international conferences and journals. His special research interests focus on electronic system-level design, design-space exploration, and multiobjective evolutionary algorithms.



Andy D. Pimentel (M'05–SM'06) received the M.Sc. and Ph.D. degrees in computer science from the University of Amsterdam, Amsterdam, The Netherlands.

He is currently an Associate Professor with the Computer Systems Architecture Group, Informatics Institute, University of Amsterdam. His research interests include computer architecture, computer architecture modeling and simulation, system-level design, design-space exploration, performance and power analysis, embedded systems, and parallel computing.

Dr. Pimentel is a member of the European Network of Excellence on High-Performance Embedded Architecture and Compilation and a Cofounder of the International Symposium on embedded computer Systems: Architectures, Modeling, and Simulation (SAMOS). He serves on the editorial boards of Elsevier's *Simulation Modelling Practice and Theory* as well as Springer's *Journal of Signal Processing Systems*. Moreover, he has also served on the organizational committees for a range of leading conferences and workshops, such as DATE, IEEE ICCD, FPL, SAMOS, and IEEE ESTIMedia.

Todor P. Stefanov (S'01–M'05) received the Dipl.Ing. and M.S. degrees in computer engineering from the Technical University of Sofia, Sofia, Bulgaria, in 1998 and the Ph.D. degree in computer science from Leiden University, Leiden, The Netherlands, in 2004.

From 1998 to May 2000, he was a Research and Development Engineer with Innovative Micro Systems, Ltd., Sofia. From June 2000 to August 2007, he was with the Leiden Institute of Advanced Computer Science, Leiden University, where he was a Research Assistant (Ph.D. student) and a Postdoc Researcher with the Leiden Embedded Research Center. From September 2007 to August 2008, he was a Senior Researcher with the Computer Engineering Laboratory, Delft University of Technology, Delft, The Netherlands. Since September 1, 2008, he has been an Assistant Professor with the Leiden Institute of Advanced Computer Science, where he performs research at the Leiden Embedded Research Center. His research interests include several aspects of embedded systems design, with particular emphasis on system-level design automation, multiprocessor systems-on-chip design, and hardware/software codesign.

Daniel D. Gajski (M'77–SM'83–F'94) received the Dipl.Ing. and M.S. degrees in electrical engineering from the University of Zagreb, Zagreb, Croatia, and the Ph.D. degree in computer and information sciences from the University of Pennsylvania, Philadelphia.

After ten years as a Professor with the University of Illinois, Urbana–Champaign, he joined the University of California, Irvine (UCI), where he currently holds the Henry Samueli Endowed Chair in Computer System Design. He also currently directs the Center for Embedded Computer Systems, UCI, with a research mission to incorporate embedded systems into automotive, communications, and medical applications. He has authored over 300 papers and numerous textbooks, including *Principles of Digital Design* (Prentice Hall, 1997) that has been translated into several languages.

Jürgen Teich (M'93–SM'07) received the Dipl.-Ing. degree (with honors) from the University of Kaiserslautern, Kaiserslautern, Germany, in 1989 and the Ph.D. degree (*summa cum laude*) from the University of Saarland, Saarbrücken, Germany, in 1993.

In 1994, he joined the DSP Design Group University of California, Berkeley, where worked in the Ptolemy project (Postdoc). From 1995 to 1998, he held a position with the Institute of Computer Engineering and Communications Networks Laboratory (TIK), ETH Zurich, Switzerland, finishing his habilitation in 1996. From 1998 to 2002, he was a full Professor with the Department of Electrical Engineering and Information Technology, University of Paderborn, Paderborn, Germany, holding a chair in computer engineering. Since 2003, he has been a Full Professor with the Computer Science Institute, Friedrich-Alexander-University of Erlangen–Nuremberg, Erlangen, Germany, holding a chair in hardware–software codesign.

Dr. Teich has been a member of multiple program committees of well-known conferences and a Program Chair for CODES+ISSS 2007 and FPL 2008. In 2004, he was elected a Reviewer for the German Science Foundation (DFG) for the area of computer architecture and embedded systems.