Automatic System-Level Synthesis: From Formal Application Models to Generic Bus-Based MPSoCs

Jens Gladigau[1], Andreas Gerstlauer[2], Christian Haubelt[1], Martin Streubühr[1], and Jürgen Teich[1]

[1] Department of Computer Science, University of Erlangen-Nuremberg, Erlangen, Germany

[2] Department of Electrical and Computer Engineering, University of Texas at Austin, Austin, USA

Abstract. System-level synthesis is the task of automatically implementing application models as hardware/software systems. It encompasses four basic subtasks, namely *decision making* and *refinement* for both computation and communication. In the past, several system-level synthesis approaches have been proposed. However, it was shown that each of these approaches has drawbacks in at least one of the four subtasks. In this paper, we present our efforts towards a comprehensive system-level synthesis by combining two academic system-level solutions into a seamless approach that automatically explores and generates pin-accurate implementation-level models starting from a formal application model and a generic MPSoC platform. We analyze the system-level synthesis flow and define intermediate representations in terms of transaction level models that serve as link between existing tools; automated transformations between these models are presented. Furthermore, we drive design decisions for both flows through a single design space exploration engine. We demonstrate the resultant flow and show the benefits of fully automatic exploration and synthesis for rapid and early system-level design.

1 Introduction

System-level design has long been touted as the holy grail for increasing designer productivity, raising the level of abstraction while providing associated design automation techniques. Several approaches provide at least partial solutions for synthesis at the system-level. However, the landscape remains fragmented. There are various attempts that focus on certain aspects of the problem, but a complete system synthesis solution is lacking [10, 5, 15].

In this paper, we identify and define different abstraction levels in typical system-level design flows. Additionally, we show how existing approaches can

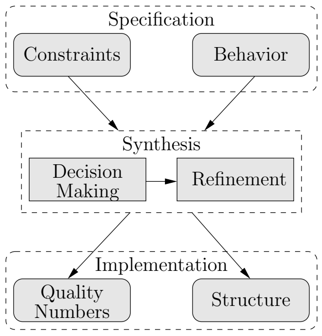

Fig. 1. X-chart showing the synthesis process [10]

be combined to a seamless system-level synthesis. First, actual needs and elements of *synthesis* have to be clarified. At any abstraction level, synthesis can be defined as the process of transforming a specification into an implementation (Fig. 1). We concentrate on the system-level, where synthesis is performed across hardware/software boundaries. This is represented by the X-chart as follows: A system-level specification is composed of an application behavior and constraints. Constraints at this level include non-functional constraints (such as area restrictions or performance requirements), and a platform that describes available resources [23]. Resources include communication (busses, gateways, memories, etc.) and computation resources (such as processors and hardware accelerators). Supplemented by allowed connectivity and associated parameters, a platform describes all possible platform instances. Note that only a subset of elements in the platform may be chosen for a platform instance. System-level synthesis generates an (optimal) implementation of the application model under the given constraints. This is achieved through decision making and refinement. Decision making is understood as the task of: (1) computing an allocation of resources for computation and communication available in the platform, (2) a spatial binding of the application (tasks and communication) onto allocated resources, and (3) a temporal scheduling to resolve resource contention of objects bound to the same resource. Decision making has additionally to respect the non-functional constraints from the specification. Taking all these decisions, system-level refinement automatically generates an implementation. The implementation consists of a structural model and quality numbers. The structural model represents the resulting platform instance, mapping, and scheduling decisions. For structural representation, Transaction Level Modeling (TLM) is almost exclusively used today [11]. Quality numbers are estimated values for different implementation properties, e.g., timing, area, or power consumption. In the following, we will refer to the structural representation as implementation.

In general, in EDA the goal at any abstraction level is to automate both tasks, decision making and refinement. However, compared to lower levels, synthesis at the system-level has to deal with vast design spaces and increased complexities. These can only be managed by orthogonalization of concerns [17]. In addition to a

Table 1. Classification of different ESL synthesis approaches [10] with updates

		Decision Making		Refinement	
Approach	DSE	Comp.	Comm.	Comp.	Comm.
Daedalus [28]	●	●	○	●	○
Koski [27]	●	●	○	●	○
Metropolis [1]	–	○	–	○	–
PeaCE [14, 18]	●	○	○	●	○
SCE [6]	–	–	–	●	●
SystemCoDesigner [16]	●	●	●	●	–

– no support ○ partial support ● full support

separation of decision making and refinement, both steps are typically performed separately for computation and communication.

A previous analysis of the system-level synthesis landscape [10] has shown that various approaches exist that perform decision making and/or refinement for either computation or communication (cf. Table 1). However, none of the investigated approaches can handle comprehensive system-level synthesis that includes automated decision making and refinement for both computation and communication. By combining existing tools, a comprehensive system-level synthesis method is achieved. This is the motivation for the work at hand.

System-level synthesis approaches can be divided into two classes: (1) approaches starting with formal, domain specific application models (e.g., based on data flow models) like Daedalus, Koski, PeaCE, SystemCoDesigner, and (2) approaches starting with implementation oriented application models like Metropolis and SCE; the latter are typically based on programing language extensions. Approaches from (1) usually consider and support very limited, restricted target architectures that often directly match the semantic of the domain specific model; approaches from (2) are closer to traditional implementation flows and support more general target architectures.

For the application behavior, a well defined Model of Computation (MoC), as deployed by approaches from (1), that allows for analysis and utilization of formal methods is preferred. MoCs used for application modeling include process, data flow, or state-machine models [19]. Executable application models additionally help to avoid ambiguous behavior. Therefore, in our approach, we advocate an executable application model based on a well defined MoC.

The key contribution of this paper is the definition of an automatic system-level synthesis flow that allows us to bridge the synthesis gap between the two classes—formal model based and implementation centric synthesis approaches. As a proof of concept, we couple representatives from both classes: the System-CoDesigner (SCD) [16] and the System-On-Chip Environment (SCE) [6] solutions. From the analysis of Table 1, combining these flows is most promising. Both tools are in use with industrial partners, e.g., SCE has been commissioned by JAXA for use by its suppliers, and SCD is used in collaborations with Daimler, Siemens, Audi, and IBM. The combined flow is novel, i.e., as of now academic and has not yet been evaluated in an industrial context. What seems like an engineering task includes two major conceptual challenges: (1) As typically all design decisions at a given abstraction level interfere with each other, we need to use a single design space exploration (DSE) engine in order to perform multi-objective optimization. SCD's DSE needs to be substantially extended, as now decisions for a more general target platform have to be made. This includes new non-functional constraints and new resource capabilities, which have not been considered in the previous, domain specific platform. (2) The result of SCD's synthesis is not the final implementation anymore, but an intermediate model that must be further processed by SCE. Thereby, the link to the formal model must not be lost, as SCE also needs to interpret and implement design decisions, made by the DSE engine from SCD. For this purpose, we introduce well defined intermediate transaction level models. The combined design flow allows for fully automatic design space exploration including automatic decision making and refinement for both, computation and communication. Thereby, the resultant design flow starts from a formal application model going down to pin-accurate models of arbitrary, bus-based MPSoCs.

Organization The remainder of the paper is organized as follows: Section 2 describes related work, followed by an overview of our methodology in Section 3. The refinement procedure from a formal application model down to pin-accurate models is explained, and intermediate transaction level models are defined in Section 4. The design space and its exploration is detailed in Section 5. The coupling of design flows is presented in Section 6. Results of applying the proposed methodology to a typical streaming application case study, a JPEG decoder design, are presented in Section 7. Finally, the paper concludes with a summary in Section 8.

2 Related Work

Many approaches exist today that tackle a subset of what we expect from comprehensive system-level synthesis. We target a synthesis approach that covers the complete flow shown in Fig. 2: a formal application Model that enables analysis, and automation in refinement and design space exploration. In [5], Densmore et al. define a classification framework for design tasks by reviewing more than 90 different tools. Many of these tools are devoted to modeling purposes (functional or platform) only. Other tools provide back-end synthesis functionality through either software code generation or C-to-RTL high-level synthesis. A more recent survey [15] focuses on methods and tools for embedded reconfigurable systems. Their analysis also shows that for all tasks in system-level synthesis tools and solutions exist, but are not seamless connected.

Not rephrasing the comprehensive discussion in the cited surveys, we relate other approaches to our proposed design flow. Targeting streaming applications mapped to network on chip architectures, in [21] stochastic automata networks are used for performance analysis. According to analysis results, the application is then mapped to a target architecture. Similarly, the DeepCompass framework [3] is able to perform analysis and design space exploration for software systems deployed on multiprocessor platforms, based on manual developed resource models of system components. For both, the selected model then provides design decisions for engineers to develop the final implementation. An approach close to our proposal is presented in [18]. There, the application model is limited to synchronous communication, and shows lesser flexibility in communication topology exploration. Due to this specialization, compared to our proposal, better implementations can be expected in context of synchronous applications. The OSSS approach [13] developed in the ICODES project is a SystemC-based solution for modeling and refinement. While refinement is automated, decision making is not. However, an ideal system-level synthesis flow has the ability to *generate* systems across hardware and software boundaries from an application model *automatically.* Several academic approaches to system-level synthesis exist today, yet none of them fully automates decision making and refinement (see Table 1). Metropolis [1] is a modeling and simulation environment based on the platform-based design paradigm. It supports many application domains and target architectures. However, it does not provide a high degree of automation, neither in decision making nor in model refinement. PeaCE [14, 18] supports automatic computation refinement and decision making. The flow is particular suited for DSP applications, and supports limited target architectures. Daedalus [28], Koski [27], and SystemCoDesigner (SCD) [16] are system-level synthesis tools that automatically map applications to MPSoC targets. These tools support decision making and refinement for application computation, but decision making and refinement for communication is only supported for limited types of communication architectures. Since communication is mandatory for MPSoCs implementations, and significantly influences performance, system-level synthesis should support both computation and communication. Concentrating on communication, in [4] a formal application model (KPN) is used. While they implement refinement, major difference to our approach are manual decision making, and a different target architecture. An approach that automates design space exploration at system-level for MPSoCs based on shared memory communication architectures has been proposed in [20]. It uses a special decoding based on SAT solving during design space exploration (DSE). Thereby, it determines and optimizes resource allocation, process binding, channel mapping, and transaction routing. However, refinement was not considered in this paper. A comparable approach using ant colony optimization was presented in [9]. Automatic refinement for both computation and communication is implemented in the System-On-Chip Environment (SCE) [6]. There, however, no support for automatic decision making is integrated, and the refinement process starts from a C-based system model.

In summary, today system-level synthesis tools, which support automatic design space exploration, are model-based and make use of restricted target architectures; synthesis-tools, which support automatic refinement to generic MP-

SoCs, are implementation centric and, hence, cannot exploit high-level model information. As a consequence, in the present paper, we will close the gap between higher-level formal models feeding into such implementation-driven synthesis flows by combining an automatic decision making with refinement steps from SCD and SCE.

3 Methodology Overview

An overview of the proposed design flow is shown in Fig. 2. As described using the X-chart, system-level synthesis starts from an application model and a platform. System behavior is given as a formal application model based on the data flow oriented FunState MoC [25], where actors communicate via channels. Because using only queues with FIFO semantics, this MoC is best suited for streaming applications, as found in the multimedia or networking domains. In FunState, each actor is modeled by a finite state machine, which controls the consumption and production of data on the channels. Moreover, the FSM triggers so called actions to transform data values. While we can later map several actors to a single resource, we, for now, do not further parallelize a single actor. Hence, the initial decomposition of the system in the formal model also defines the maximum parallelism of processes in the final implementation. Note that support for other MoCs, e.g., synchronous data flow (SDF), Kahn process networks (KPN), or communicating sequential processes (CSP), could be integrated, but is not further discussed.[3] Constraints, on the other hand, are the platform as well as additional non-functional constraints (restrictions on area, performance requirements, etc.). A platform defines a set of available resources at the granularity of processors, memories, busses, bridges, as well as their possible interconnection and feasible parameter settings.

Given a complete system specification, system synthesis could be performed. During system-level synthesis, three abstraction levels are defined through the following intermediate models: (1) Application Models, (2) Scheduled Models, and (3) Architecture Models. In these models, communication is abstracted to transactions. Hence, we define them in terms of TLM concepts. Detailed definitions follow in Section 4.

Beside a single system synthesis step, a design space exploration (DSE) engine performs multi-objective optimization to obtain high-quality design points. Design points represent different implementations according to varying design decisions such as computation and communication resource allocation, resource parameter settings including protocol and operating system selection, actor binding, channel mapping, memory and synchronization transaction routing, and scheduling/arbitration. Decisions of a chosen design point are then fed into a

[3] In this paper, we use restricted FunState. In this MoC, application behavior is described as a set of actors that communicate via queues with FIFO semantics. Each actor's behavior is given as finite state machine. Basically, this MoC extends KPN by non-deterministic behavior and allows for non-blocking reads. For simplicity, we further refer to this MoC as FunState.

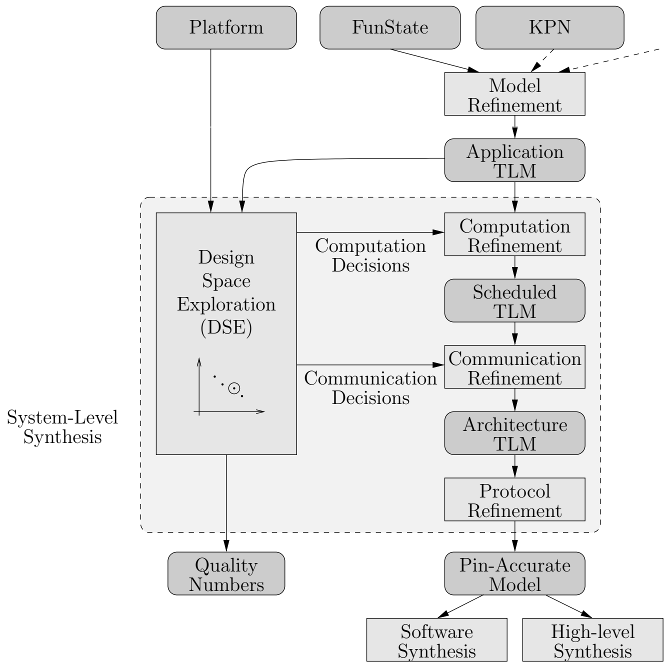

Fig. 2. System-level synthesis overview

refinement flow that refines the application model down to a Scheduled TLM, an Architecture TLM, and finally a PAM, by implementing the actor binding, channel mapping, computation and communication resource instantiation, and scheduling/arbitration. It is important to point out that a single design space exploration engine determines all system-level design decisions for the intermediate levels in refinement, performed by different tools. Refinement is performed in four phases:

1) **Model refinement**: The formal application model is refined into a transaction level model. Queues using FIFO semantics are implemented in shared memory. Hence, in the Application TLM, queues are refined into memory access and synchronization transactions. Each actor is implemented as a process (cf. Section 4.1).
2) **Computation refinement**: Computation refinement uses computation decisions from DSE to generate an intermediate Scheduled TLM. In the Scheduled TLM, processes are bound and scheduled on allocated and configured computational resources (processors or hardware accelerators). Interprocessor communication is equal to the Application TLM (cf. Section 4.2).
3) **Communication refinement**: Design decisions about the topology of the communication architecture (allocation and configuration of communication resources), channel mapping, routing of memory and synchronization transactions, and bus address and interrupt mapping are realized. During refine-

ment, memory and synchronization transactions are implemented down to the level of arbitrary bus protocol transactions. The Architecture TLM is the result. In this refinement, bus drivers and bus interfaces in the processors are generated to implement all memory and event communication as bus read, write, and interrupt transactions (cf. Section 4.3).

4) **Protocol refinement**: Complementing communication refinement, communication is refined all the way down to cycle-accurate signals and events over a set of wires. In the resulting PAM, bus-functional models of processors, memories, hardware accelerators, busses, and gateways communicate at a level down to the sampling and driving of individual wires (cf. Section 4.4).

System-level synthesis results in a pin-accurate model (PAM) and corresponding quality numbers. PAMs describe the system as a netlist of task-accurate, bus-functional component models that are ready for further software and high-level synthesis.

Focusing on system-level synthesis aspects, the required model refinement steps from an application model to a PAM will be discussed in detail in Section 4. Design space exploration is presented in more detail in Section 5.

4 System-Level Refinements

We now explain the refinement steps of our proposed system-level synthesis approach shown in Fig. 2 in more detail. We thereby define intermediate canonical abstraction levels (Application TLM, Scheduled TLM, and Architecture TLM) necessary for a seamless synthesis flow. The basic idea is to synthesize MPSoC implementations from application models by defining intermediate models in terms of transaction level concepts. These models are the basis for performing computation, communication, and protocol refinement across tool borders. In the following, we present the necessary model transformations in more detail, starting with the refinement process from FunState application models. In the figures and the text, we use the widespread ISO/OSI layers for illustration to denote the abstraction level. These layers are not part of the final implementation and optimized.

4.1 Model Refinement

To support further synthesis down to general bus-based MPSoC target architectures, an Application TLM is represented in a C-based system-level design language, such as SystemC/TLM [22]. Therefore, abstract communication in the formal model must be implemented as transactions in the Application TLM. In the following, we will show the translation of FunState models using a fixed decision for implementing communication as shared memory.[4] With such a refinement, the same formal model used in an approach supporting limited target

[4] Of course, alternative implementations for this intermediate model in the design flow like distributed communication buffers and polling schemes are possible. However, for simplicity, we will only consider models as described.

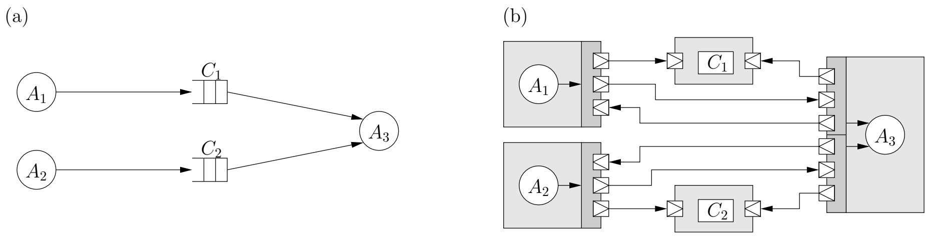

Fig. 3. (a) FunState model and (b) Application TLM

architectures can be automatically transformed into a model suitable for implementation oriented approaches—the key for synthesizing general bus-based MPSoC target architectures.

In FunState models, actors communicate via point-to-point queues with FIFO semantics. These queues support blocking and non-blocking read operations as well as write operations. In the transaction level model, these operations are implemented using shared memory to store data and control information (e.g., read/write pointers) combined with transactions for data access and synchronization. The corresponding model transformation is shown in Fig. 3. Fig. 3(a) represents a FunState model consisting of three actors A_1, A_2, and A_3, and two queues C_1 and C_2. In order to generate an equivalent transaction level model using shared memory communication, each queue is replaced by a shared memory module and sockets for synchronization, as shown in Fig. 3(b). The behavior of an actor is encapsulated as process in a SystemC module. TLM adapters [12] are inserted for interconnection (indicated as dark gray layer in the figure). Adapters consist of two initiator sockets and one target socket. One initiator socket is used for memory access, whereas the second initiator socket is used for synchronization (to notify the peer actor module about queue updates). These adapters provide an abstract interface to the process for communication. Adapters therewith implement the FIFO semantics of queues from the formal model.

4.2 Computation Refinement

The main purpose of system-level computation refinement is to partition processes in the application model towards their implementation as hardware accelerator or on a software programmable processor. As such, groups of processes are partitioned according to the process binding according to the design decisions. End-to-end communication over sockets is implemented the same way as in the Application TLM. The resulting model is called Scheduled TLM, as scheduling decisions are implemented during computation refinement.

For partition blocks that result in a hardware implementation, system-level computation refinement adds another hierarchy level. Thus, hardware blocks are encapsulated and the inherent parallelism in the model is retained. The result is a single SystemC module for each hardware accelerator. After refining the communication (see Section 4.3), high-level synthesis tools, such as Forte's Cyn-

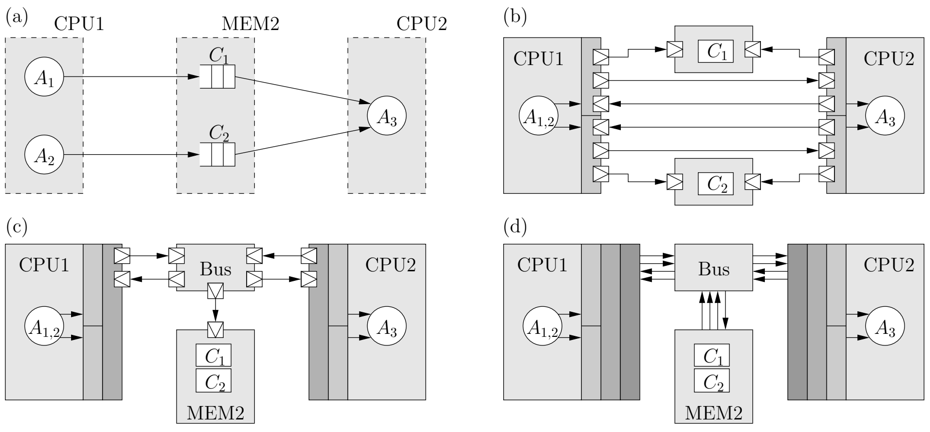

Fig. 4. (a) FunState model with resource mapping; (b) Scheduled TLM; (c) Architecture TLM; (d) pin-accurate model

thesizer, Cadence's C-to-Silicon, Mentor's Catapult, or NEC's CyberWorkBench, can generate RTL implementations for each module. To allow for high-level synthesis, actions in the FunState model have to obey the restrictions from the used synthesis tool.

For software partitions, computation refinement mainly deals with scheduling, taking processor allocation and process binding into account. See Fig. 4(a) for such a mapping. Scheduling serializes execution of processes mapped to a single processor. Several scheduling approaches can be considered: (1) static scheduling, (2) quasi-static scheduling, or (3) dynamic scheduling (e.g., round-robin or priority based). The first two scheduling techniques are optimizations that may be applied to subsystems with static communication rates [2, 8]. Applying static scheduling results in a single process representing the software partition block. Dynamic scheduling is the most general solution and always applicable. It may result in a single process implementing a custom schedule, or in multiple threads later executed on an operating system. In our experiements, we only used the latter option. Further software synthesis in the back-end then generates C/C++ code for each process. The resulting code is compiled and possibly linked with an operating system to run on the corresponding target processor.

4.3 Communication Refinement

Communication refinement from Scheduled TLM (Fig. 4(b)) to Architecture TLM (Fig. 4(c)) is performed in two steps. First, adapters are aggregated for each hardware or software partition [12]. Aggregation includes insertion of code performing the tasks of transport and network layers, i.e., code that encapsulates all queue adapters of a partition block and performs end-to-end packeting, addressing, and routing, according to design decisions. The results are SystemC modules for processors and hardware accelerators. Each module is equipped with a pair of sockets for communication.

In a second step, sockets are further aggregated and refined down to read or write transactions over shared busses or other communication media. Link and media access layers are inserted to implement addressing, data transfers (using various protocol modes, such as burst or DMA), and synchronization (such as polling or interrupts). The result is an Architecture TLM realizing communication at the media access level, which is indicated by the dark gray bars in Fig. 4(c).

4.4 Protocol Refinement

Protocol refinement further implements fixed decisions for communication, namely address mapping and interrupt assignment. It synthesizes the Architecture TLM down to a PAM (illustrated in Fig. 4(d)) as follows: Modules are refined into bus-functional models by inserting protocol and physical layers that implement bus protocol state machines for each bus transaction, driving and sampling ports and wires in accordance with the selected protocol timing. Consequently, communication in the PAM is pin- and cycle-accurate. The PAM is described as a signal-level netlist of processors, memories, bus wires, and interconnect components, such as multiplexers, arbiters, bridges, and gateways. The PAM represents the final system netlist, which is the basis for further software and high-level, logic and physical hardware synthesis, e.g., for ASIC manufacturing or FPGA-based prototyping.

5 Design Space Exploration

During design space exploration different implementations are generated and iteratively improved, in order to find high quality solutions. For this purpose, system-level design decisions are varied, and their impact on the quality of the implementation is estimated. In the following, an automatic design space exploration is proposed. It is based on a single optimization engine to generate decision for intermediate levels occurring in system-level synthesis, namely Scheduled TLMs, Architecture TLMs, and pin-accurate models.

The Design Space All possible decisions spawn the design space. In the proposed design flow, there are four refinement steps, explained in Section 4. In our experiments, model refinement and protocol refinement implement fixed design decisions: channel-based communication is implemented as shared memory, and address mapping and interrupt assignment is performed. Computation refinement and communication refinement implement decisions from design space exploration. These computation and communication refinement lead to intermediate levels, namely the Scheduled TLM and Architecture TLM. In order to define the design space for an automatic exploration, Table 2 summarizes important decisions and examples of such decisions at each intermediate level. This list is by no means complete. Instead, it shows the overall complexity of the system-level synthesis task. This becomes particularly obvious, when seeing that

Table 2. Design decisions at intermediate levels

Intermediate Level	Decisions	Examples
Computation	Allocation of computational resources	CPUs, DSPs, ASIPs, HW accelerators
	Allocation parameters	ARM1 runs at 300MHz
	Binding of processes	DCT process runs on a HW accelerator
	Scheduling	ARM1 schedules preemptive
	Scheduling parameters	Parser process at ARM1 has priority 10
Communication	Allocation of communication resources	Memories, busses, bridges, P2P connections; interconnection with computational and communication resources
	Allocation parameters	Shared memory SM1 has size 16kB
	Channel mapping	The communication buffer between Parser and Huffman decoder is placed in memory SM1
	Routing	memory transactions between Parser and Huffman decoder are routed via BRIDGE1
	Arbitration	Bus BUS1 is running TDMA arbitration
	Arbitration parameters	Time slices on BUS1 are 800ms
	Communication controller parameters	Buffer sizes, reliable/unreliable connections, maximal transfer unit (MTU)

most of these decisions are interdependent. Design space exploration determines all these parameters at one.

Most important decisions implemented in computation refinement include allocation of computational resources, the binding of processes, and the selection of the scheduling policy. Additionally, implementation parameters are selected. This includes general parameters like frequencies and scheduling priorities as well as more specific parameters like the initializing processor for the entire system.

Major decisions respected in communication refinement are allocation of communication resources, channel mapping, and memory and synchronization transactions. They are complemented by implementation related parameter selections. The latter include selection of arbitration policies for shared media as well as associated parameter settings. It should be noted that the allocation of communication resources also includes the interconnection of all resources. Hence, communication refinement generates the overall topology of the implementation. The channel mapping often includes a local address selection, i.e., placing channel buffers into memories, as well as a global address mapping, which results in transaction routing information needed for inter-resource communication via shared memory.

Constraints Implementations must obey a number of constraints that narrow the number of feasible and valid implementations. Constraints can be either

Table 3. Design parameters and constraints

	Parameter	Constraints
Computation	Process binding	Each process is bound exactly once
	Clock frequency	per processor: exacly one out of {100, 150, ...} is selected
	Synchronization method	per channel: exactly one out of {polling, interrupt} is selected
	Scheduling policy	per SW partition: exactly one out of {priority based, round-robin} is selected
	Task priority	unique per task inside SW partition: {2, ..., 61}
Communication	Resources	# master per AHB ≤ 4 # ARM per AHB ≤ 1
	Channel binding	Each channel is mapped exactly once; one ARM initializes all channels mapped to shared memories
	Bus type	per bus: exactly one out of {AHB, ... } is selected
	Arbitration policy	per AHB: exactly one out of {priority based, round-robin} is selected
	Transaction routing	# of bus transducer ≤ 1 per route

functional or non-functional. Functional constraints, for example, require that the behavior is entirely mapped onto the platform, and communication between processes due to data dependencies is established via connected resources. Non-functional constraints are often area, power, or throughput constraints. Some important functional and non-functional constraints for each intermediate level are shown in Table 3.

All possible decisions including computation and communication resource allocation, process binding, channel mapping, transaction routing, and parameter assignments define the design space. Each point in the design space corresponds to an implementation. If the implementation does not satisfy the functional constraints, the implementation is said to be *infeasible*. In general, infeasible implementation do not permit a meaningful quality estimation used for the optimization. The set of feasible implementations defines the *feasible region* (see decision space in Fig. 5) of the design space. Feasible implementations that do not fulfill the non-functional constraints are called *invalid* implementations. Hence, the set of valid implementations defines the *valid region* of the design space. As non-functional constraints are often related to quality numbers, these numbers typically are determined or estimated to allow checking validity of an implementation.

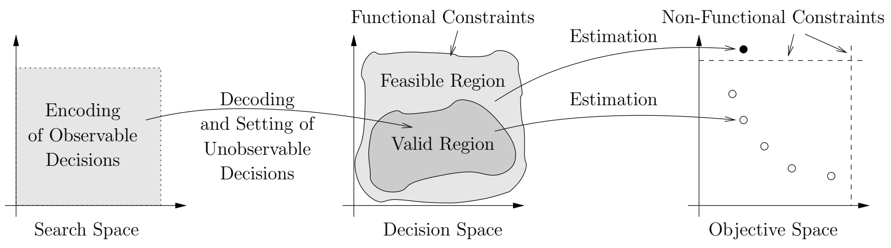

Fig. 5. Search space, decision space, and objective space

Objectives From among the valid implementations, we are especially interested in the best (high quality) solutions. Embedded systems are typically optimized with respect to several, often conflicting objectives. So, in general, not one optimal solution exists. Objectives to be optimized are often area, power consumption, throughput, or reliability. The considered objectives are a subset of the quality numbers associated with each implementation. Hence, estimation methods during the design space exploration are required in order to determine the quality numbers associated with each design point.

In general, the performance assessment can be done using the PAM, which is also part of the output of the systems-level synthesis. However, this requires that all refinement steps have been performed taking all system-level decisions into account. Such an approach might be prohibitively slow, especially in early design phases. Another option is to perform the estimation at any intermediate level defined in our proposed design flow. In our experiments, we used an Architecture TLM to perform a simulation-based performance estimation (latency and throughput) in order to achieve high exploration performance. The abstraction results in a transition accurate modeling of actor execution delays, and a modeling of simple transfer delays for transactions. Scheduling and arbitration for CPUs and busses are abstracted to simple policies like priority-based or round robin. Besides latency and throughput, we estimated the number of required look-up tables (LUTs), the number of block RAM (BRMAs), and the number of flip-flops (FFs) for an FPGA implementation. These numbers can be used to quantify the area consumption of an implementation. The estimation is done analytically by summing up the number of LUTs (the same for BRAMs and FFs) from each allocated computation and communication resource.

This abstraction for estimating the quality numbers results in a bipartition of all design decisions: *observable* and *unobservable* design decisions (see search space Fig. 5). Only observable design decisions have an observable impact on quality estimations and can be iteratively improved during design space exploration. As a consequence, unobservable design decisions (e.g., address mapping) are typically constructed without being subject to optimization.

Automatic Exploration An automatic exploration of the design space is performed by defining an appropriate encoding of the decision space. Based on the chosen encoding, a general optimization heuristic like particle swarm optimization or evolutionary algorithms can be applied. However, this also requires the implementation of a corresponding decoding strategy that transforms a solution represented in the search space into a design point in the decision space (Fig. 5). If the decoding strategy guarantees that the solutions in the decision space are feasible, it is called *feasibility preserving*. Only for feasible solutions in the decision space quality estimation can be performed. After estimation, it can be checked if a feasible solution obeys to all non-functional constraints. If yes, it is a valid solution. Among the valid solution DSE searches for the best solutions, i.e., the non-dominated solutions, and tries to improve them.

In our experiments, we used a hybrid encoding by representing computation and communication resource allocation, process binding, and channel mapping decisions by binary variables. We construct a Boolean formula, such that a satisfying variable assignment corresponds to a feasible solution, i.e., a solution satisfying all functional constraints. Parameter selections are encoded as permutation lists. The single optimization engine is a Multi-Objective Evolutionary Algorithms (MOEA) that varies the variable assignments and the permutation lists through mutation and crossover. A feasibility preserving decoding for the allocation, binding, and mapping decisions is based on a Boolean Satisfiability solver (SAT solver) [20]. However, as parameter settings might have interdependencies, this part of a solution could only be checked after the decoding. Here, design points that do not obey all functional constraints have to be marked as infeasible. Hence, the whole decoding is not feasibility preserving in our experiments.

6 Design Flow Implementation

After analyzing refinements, intermediate models, and design space exploration in system-level synthesis, we now present our experimental setup for transformation of formal models to generic bus-based MPSoC implementations in more detail. We implemented the proposed system-level synthesis by combining the SystemCoDesigner (SCD) and the System-on-Chip Environment (SCE) design flows. To benefit the most from the strengths of both tools, we identified the Scheduled TLM as link. As such, automatic decision making and computation refinement is performed using SCD, while the more elaborated communication refinement from SCE is used. Here, we concentrate on the hand-over, as the other refinement steps are explained in the cited literature.

Neglecting intermediate steps, Fig. 6 sketches the resulting system-level synthesis tool flow, related to the X-chart. Behavior is given as executable representation using the SysteMoC library [26]. Constraints are fed to the DSE engine inside SCD using an XML file description of the platform, and non-functional constraints. One feature of SysteMoC is the capability to automatically extract model information. These include finite state machines, describing the behavior of actors, and the overall structure of the model. Following the methodology described in Section 4, this information together with computation decisions from DSE is used to automatically generate the Scheduled TLM in SpecC format [24]

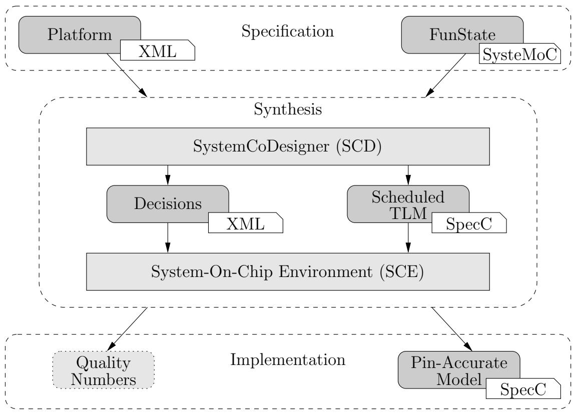

Fig. 6. Tool flow

for a chosen design point. Decisions made by the exploration framework from SCD are propagated using XML files, which in turn feed into scripts that drive the SCE refinement engine. Taking the Scheduled TLM and communication decisions, SCE refines abstract memory access and synchronization transactions in the model down to bus transactions and interrupts. The resulting implementation as pin-accurate model is in SpecC format. The quality numbers of the implementation are either taken from DSE estimation, or gained by running timed SpecC simulation of Scheduled TLM or Architecture TLM models. Also, estimation at PAM level using further synthesized software running in instruction set simulation is possible.

After briefly describing modeling concepts for SysteMoC in the following subsection, we explain how the SpecC model is generated from the SysteMoC model, implementing design decisions. This SpecC model acts as the link between the two tool flows.

6.1 SysteMoC

SysteMoC is a C++-library for implementing formal, executable FunState based representations of embedded systems [7]. For the running example introduced in Fig. 3, an illustrating SysteMoC model is shown in Fig. 7(a) in more detail. Every actor contains a finite state machine (FSM) with one or more states, depicted above a dashed line. Below the dashed line, actor methods (actions) and internal variables are shown. Actors access channels via named ports (black rectangles in the figure). Every transition in the FSM is annotated with an activation pattern and an action, separated by a slash. If all conditions in an activation pattern are fulfilled, the transition can be taken and the corresponding action is atomically executed. For example, the transition $i_1(1)\&i_2(1)/d()$ in actor A_3 is activated if there is at least one token available in channel C_1 and at least one in channel C_2. If the transition is taken, action `d()` is executed. Afterwards, tokens are consumed (taken from the channel) according to the activation pattern.

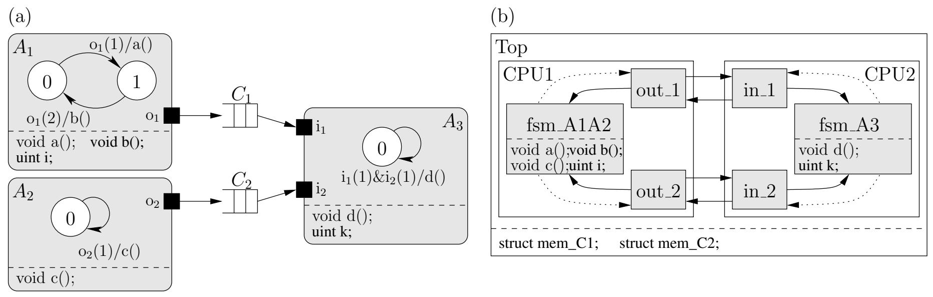

Fig. 7. (a) SysteMoC model; (b) SpecC model

6.2 SpecC Generation

In this section, we explain how to generate SpecC programs from SysteMoC descriptions and computation decisions, using the running example. That is, the transformation from the partitioned model shown in Fig. 4(a) to the Scheduled TLM in Fig. 4(b). We first give a basic overview, before emphasizing details:

- For each partition block, a module (called *behavior* in SpecC) representing one or more finite state machines is generated.
- Each SysteMoC port is implemented as an additional interface process and behavior.
- For each SysteMoC channel, memory structures for data, read pointer, and write pointer are instantiated.
- Actions are transformed so that port accesses in the SysteMoC model are mapped to interface functions of the corresponding SpecC port behavior.
- Internal variables of actors are instantiated in the behavior corresponding to the partition block.
- Signals are connected according to the structure of the model.

An important observation is that queues in the formal model imply synchronization: Actors block, if no activation pattern is fulfilled in the current state; changes on a connected channel cause reevaluation. As described in Section 4.1, asynchronous communication using abstract queues is implemented in shared memory. To avoid polling, we implement asynchronous communication of a single queue using a pair of behaviors (e.g., behavior `out_1` and `in_1` for channel C_1). These behaviors are called adapters and serve three purposes: (1) they provide interfaces, for actions to access data and for FSM code to access meta information, such as fill size; (2) using handshake signals, they awake blocked FSM behaviors on data change by the peer; (3) using handshake signals, they inform the peer adapter about data change by the FSM behavior. To fulfill these three tasks, signals are interconnected as depicted by arrows in Fig. 7(b). Note that this pattern is one out of many possible solutions and meant for generic hand over between tools. Later communication refinement may alter the implementation drastically.

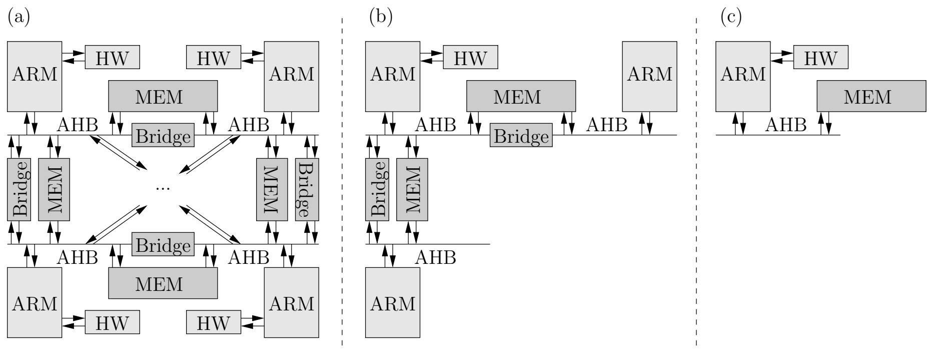

Fig. 8. (a) platform; selected (b) multi-processor, and (c) single platform instances

For hardware partition blocks, concurrent execution is retained by implementing each FSM as behavior. For software partition blocks, result depends on scheduling decisions. In case of dynamic scheduling of software partition blocks, refinement retains concurrent execution, and a general-purpose operating system is inserted as part of the SCE flow. Alternatively, a custom schedule can be used and a single behavior is the result.

For the SysteMoC example in Fig. 7(a), actors A1 and A2 are partitioned and mapped onto a CPU1 whereas actor A3 is mapped onto CPU2. As a result, a Scheduled SpecC TLM as shown in Fig. 7(b) is generated.

7 Results

To evaluate our proposed methodology implemented as shown in Fig. 6, we automatically synthesized a SysteMoC model of a JPEG decoder application into various different bus-based MPSoC implementations. We chose the JPEG decoder as this is a widely used and well known example. The application model consists of 14 actors and 22 channels.

The platform used in design space exploration contained four processor subsystems, see Fig. 8(a). Each subsystem consists of hardware accelerators, an ARM processor executing μOS, an AHB bus, and a local memory. The four subsystems are interconnected using six dual-ported shared memories and six bus bridges connected with the AHB busses. In the figure, the cross connecting bridges and memories are omitted. All actors of the JPEG decoder can be mapped to any of the ARM processors. Additionally, actors Inverse Quantization, IDCT, and DC Decoding, can be mapped to the hardware accelerators. The mapping possibilities for the queues are all six shared memories plus the four local memories. Transaction routing is constrained to the AHB busses and bus bridges. The total number of mappings in this case is approximately 10^{22}.

A Linux workstation with an Intel Core 2 Quad processor was used for design space exploration. We considered a 5-dimensional objective space: latency, throughput, #LUTs, #FFs, #BRAMs. The estimation of latency and through-

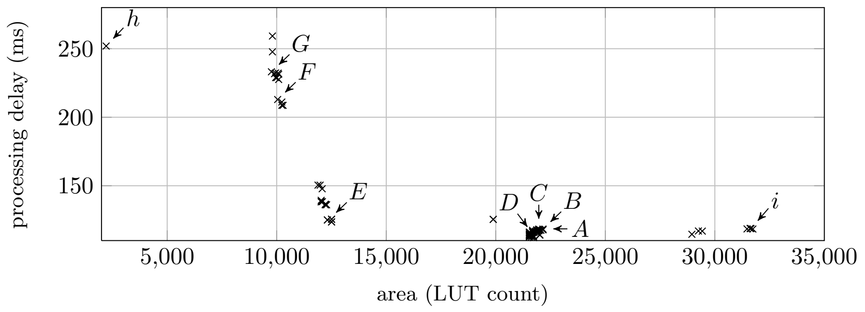

Fig. 9. Area and processing delay trade-off for the 100 best found implementations

put has been performed by simulation using an annotated SysteMoC model corresponding to the Architecture TLM. The resource utilizations is estimated using an analytical method. The exploration engine evaluated 51,200 feasible design points. It ran for approximately 38 hours. Most of the exploration time was used for simulation-based performance evaluation. Exploration result was an archive containing 100 valid design points.

The two-dimensional projection of area and processing delay is depicted in Fig. 9. There, clusters of implementations with similar quality numbers can be identified. Source of similarity is related to allocation of resources and mapping: small differences are from variations in hardware accelerators; large differences source at number of ARM cores and shared memories. A cluster containing single ARM solutions includes points *F* and *G*; the cluster at point *E* represents implementations with two ARM cores; the cluster with points *A*, *B*, *C*, and *D* includes solutions using three ARM cores. The point *h* represents a single ARM core solution without hardware accelerator and only local memory. The area increase from this point to the single ARM cluster with hardware accelerators results from constraints: to connect an accelerator, bus and shared memory are mandatory. Area differences between clusters result from more allocated resources. The cluster at point *i* includes implementations with four ARM cores and three shared memories. Due to communication overhead, the processing delay does not decrease any further, but the throughput is increased due to pixel pipelining (not noticeable in the two-dimensional projection).

System-level synthesis was performed for seven different design points (*A*-*G* in Table 4, also marked in Fig. 9). These design points represent fundamental design decisions for cheap or fast implementations. The resulting platform instances for a multi-processor solution represented by design point *A* is shown in Fig. 8(b). Fig. 8(c) sketches the single processor implementation determined by design point *G*. All designs were automatically synthesized down to Architecture TLMs and PAMs. The total time needed for refinement and generation of all models was in the order of minutes. Functional correctness of all synthesized models was verified by simulation using a testbench that decodes several color JPEG pictures in QCIF format.

Table 4. Synthesis and simulation results

	Simulation time		Code lines	
Design	Arch. TLM	PAM	Arch. TLM	PAM
A	42:55	4:55:58	36,834	35,251
B	44:34	5:00:37	35,749	34,167
C	46:11	5:27:23	35,729	34,157
D	41:34	5:51:00	32,203	30,642
E	36:01	4:57:00	28,899	27,780
F	25:41	3:13:13	27,478	26,847
G	26:17	3:10:54	25,041	24,421

Table 4 shows synthesis and simulation results for the resulting Architecture TLMs (Arch. TLM) and PAMs. We can observe a typical growth of simulation time with increasing level of detail and accuracy. Runtimes are thereby proportional to the amount of simulated inter-module communication, and number of allocated ARM cores. Compared to the initial model, significant amounts of code and implementation detail, such as middleware, bus drivers, and interrupt handlers, are automatically synthesized during refinement. The Application TLM consists of 10,945 lines of source code. The code size of the synthesized PAMs has more than doubled compared to the Application TLM. Note that Architecture TLMs are larger than PAMs since they include code for approximately-timed simulation models of AMBA AHB busses. All in all, results demonstrate the feasibility and benefits of fast and expressive formal application models for high-level algorithmic design coupled with automatic synthesis for rapid exploration and correct-by-construction generation of detailed and optimized MPSoC implementations.

We omitted comparison of synthesis results with hand crafted models gained in a more traditional design flow, but these models may perform better. Manual fine-tuning of models comes at the expense of time-consuming expert work—including the risk of adding errors in every manual refinement step. The proposed combined design flow provides insight, confidence, and good understanding of a design almost for free developing the application only.

8 Conclusions

In this paper, we presented an approach towards a comprehensive system-level synthesis solution—to automatically and seamlessly integrate the four basic subtasks of decision making and refinement for both computation and communication. We identified and classified different intermediate models in system-level synthesis, and we showed how to transform a high-level, formal model into an intermediate models for generic synthesis. Using these standardized model for interfacing and design exchange, we combined two advanced design flows into a seamless system-level synthesis solution from formal streaming application models all the way down to heterogeneous MPSoC implementations. An enhanced design space exploration engine provides all necessary decisions for refinement in the combined design flow. We are able to synthesize pin-accurate models for complex applications in a matter of minutes. In contrast, applying similar manual refinement steps would likely have required several man-months of effort.

Acknowledgments We thank Manan Kathuria for his precious work at the University of Texas at Austin. His efforts enabled the automatic link of our tools.

References

1. Balarin, F., Watanabe, Y., Hsieh, H., Lavagno, L., Passerone, C., Sangiovanni-Vincentelli, A.: Metropolis: An integrated electronic system design environment. IEEE Computer 36(4), 45–52 (April 2003)
2. Bhattacharyya, S.S., Murthy, P.K., Lee, E.A.: APGAN and RPMC: Complementary heuristics for translating DSP block diagrams into efficient software implementations. Design Automation for Embedded Systems 2(1), 33–60 (1997)
3. Bondarev, E., Chaudron, M.R.V., de Kock, E.A.: Exploring performance trade-offs of a JPEG decoder using the DeepCompass framework. In: Proc. of the 6th International Workshop on Software and Performance. pp. 153–163. ACM, New York, NY, USA (2007)
4. Chagoya-Garzon, A., Guerin, X., Rousseau, F., Petrot, F., Rossetti, D., Lonardo, A., Vicini, P., Paolucci, P.S.: Synthesis of communication mechanisms for multi-tile systems based on heterogeneous multi-processor system-on-chips. In: Proc. of the 2009 IEEE/IFIP International Symposium on Rapid System Prototyping. pp. 48–54. RSP '09, IEEE Computer Society, Washington, DC, USA (2009)
5. Densmore, D., Passerone, R., Sangiovanni-Vincentelli, A.: A platform-based taxonomy for ESL design. IEEE Des. Test. Comput. 23(5), 359–374 (2006)
6. Dömer, R., Gerstlauer, A., Peng, J., Shin, D., Cai, L., Yu, H., Abdi, S., Gajski, D.: System-on-Chip Environment: A SpecC-based Framework for Heterogeneous MPSoC Design. EURASIP JES 2008(647953), 13 (2008)
7. Falk, J., Haubelt, C., Teich, J.: Efficient representation and simulation of model-based designs in SystemC. In: Proc. of Forum on Specification and Design Languages. pp. 129–134. Darmstadt, Germany (Sep 2006)
8. Falk, J., Keinert, J., Haubelt, C., Teich, J., Bhattacharyya, S.S.: A generalized static data flow clustering algorithm for MPSoC scheduling of multimedia applications. In: Proceedings of the 8th ACM international conference on Embedded software. pp. 189–198. ACM, New York, NY, USA (2008)
9. Ferrandi, F., Lanzi, P., Pilato, C., Sciuto, D., Tumeo, A.: Ant colony heuristic for mapping and scheduling tasks and communications on heterogeneous embedded systems. Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on 29(6), 911–924 (2010)
10. Gerstlauer, A., Haubelt, C., Pimentel, A., Stefanov, T., Gajski, D., Teich, J.: Electronic system-level synthesis methodologies. IEEE Trans. Computer-Aided Design Integr. Circuits Syst. 28(10), 1517–1530 (Oct 2009)
11. Ghenassia, F.: Transaction-Level Modeling with Systemc: TLM Concepts and Applications for Embedded Systems. Springer-Verlag New York, Inc., Secaucus, NJ, USA (2006)

12. Gladigau, J., Haubelt, C., Niemann, B., Teich, J.: Mapping actor-oriented models to TLM architectures. In: Proc. of Forum on Specification and Design Languages. pp. 128–133 (Sep 2007)
13. Grüttner, K., Oppenheimer, F., Nebel, W., Colas-Bigey, F., Fouilliart, A.M.: Systemc-based modelling, seamless refinement, and synthesis of a jpeg 2000 decoder. In: Proc. of the Conference on Design, Automation and Test in Europe. pp. 128–133. ACM, New York, NY, USA (2008)
14. Ha, S., Kim, S., Lee, C., Yi, Y., Kwon, S., Joo, Y.P.: PeaCE: A hardware-software codesign environment of multimedia embedded systems. ACM TODAES 12(3), 1–25 (2007)
15. Jówiak, L., Nedjah, N., Figueroa, M.: Modern development methods and tools for embedded reconfigurable systems: A survey. Integr. VLSI J. 43, 1–33 (January 2010)
16. Keinert, J., Streubühr, M., Schlichter, T., Falk, J., Gladigau, J., Haubelt, C., Teich, J., Meredith, M.: SystemCoDesigner - An Automatic ESL Synthesis Approach by Design Space Exploration and Behavioral Synthesis for Streaming Applications. ACM TODAES 14(1), 1–23 (2009)
17. Keutzer, K., Newton, A.R., Rabaey, J.M., Sangiovanni-Vincentelli, A.: System-level design: Orthogonalization of concerns and platform-based design. IEEE Trans. Computer-Aided Design Integr. Circuits Syst. 19(12), 1523–1543 (2000)
18. Lee, C., Kim, S., Ha, S.: A systematic design space exploration of mpsoc based on synchronous data flow specification. Journal of Signal Processing Systems 58, 193–213 (2010)
19. Lee, E.A., Sangiovanni-Vincentelli, A.: A framework for comparing models of computation. IEEE Trans. Computer-Aided Design Integr. Circuits Syst. 17(12), 1217–1229 (Dec 1998)
20. Lukasiewycz, M., Streubühr, M., Glaß, M., Haubelt, C., Teich, J.: Combined system synthesis and communication architecture exploration for MPSoCs. In: Proc. of the Conference on Design, Automation and Test in Europe. pp. 472–477. Nice, France (Apr 2009)
21. Marculescu, R., Ogras, U.Y., Zamora, N.H.: Computation and communication refinement for multiprocessor SoC design: A system-level perspective. ACM Trans. Des. Autom. Electron. Syst. 11(3), 564–592 (2006)
22. Open SystemC Initiative (OSCI): Transaction Level Modeling (TLM) Library, Release 2.0, http://www.systemc.org
23. Sangiovanni-Vincentelli, A.: Quo Vadis, SLD? Reasoning About the Trends and Challenges of System Level Design. Proc. IEEE 95(3), 467–506 (2007)
24. http://www.cecs.uci.edu/˜specc/
25. Strehl, K., Thiele, L., Gries, M., Ziegenbein, D., Ernst, R., Teich, J.: Funstate—an internal design representation for codesign. IEEE Trans. Very Large Scale Integr. (VLSI) Syst. 9(4), 524–544 (2001)
26. http://www12.cs.fau.de/research/scd/systemoc.php
27. T. Kangas et al.: UML-based multi-processor SoC design framework. ACM TECS 5(2), 281–320 (May 2006)
28. Thompson, M., Stefanov, T., Nikolov, H., Pimentel, A.D., Erbas, C., Polstra, S., Deprettere, E.F.: A framework for rapid system-level exploration, synthesis, and programming of multimedia MP-SoCs. In: Proc. of the International Conference on Hardware-Software Codesign and System Synthesis. pp. 9–14 (2007)