

A System-Level Synthesis Approach from Formal Application Models to Generic Bus-Based MPSoCs

Jens Gladigau¹, Andreas Gerstlauer², Christian Haubelt¹, Martin Streubühr¹, and Jürgen Teich¹

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

²Department 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 sub tasks, 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 sub tasks. 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 generates pin-accurate implementation-level models starting from a formal application model and generic MPSoC architecture templates. We analyze the system-level synthesis flow and define intermediate representations in terms of transaction level models that serve as link between existing tools. Furthermore, we present the automated transformation between models for combining two design flows. We demonstrate the combined flow on an industrial-strength example and show the benefits of fully automatic exploration and synthesis for rapid and early system-level design.

I. 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 [1].

In this paper, we identify and define different abstraction levels in typical system-level design flows, and we show how existing approaches can 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). Concentrating on the system-level, where synthesis is performed across hardware/software boundaries, this is represented by the X-chart as follows: A specification is composed of an application behavior and constraints. Constraints often include a platform that describes available resources [2]. A set of these resources supplemented by possible interconnections form the platform template. Given a specification, synthesis generates an (optimal) implementation of the application model under the given constraints through decision making and refinement. Decision making is understood as the task of computing an allocation

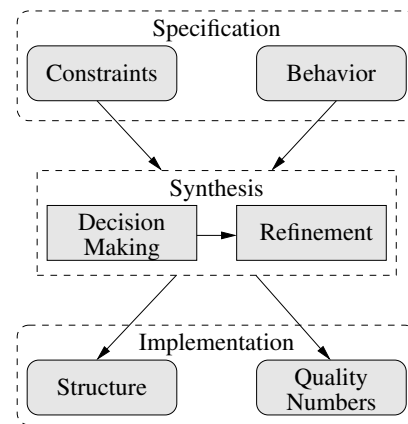


Fig. 1. X-chart showing the synthesis process [1]

of resources available in the platform template, a spatial binding of the application onto allocated resources, and a temporal scheduling to resolve resource contention of objects bound to the same resource. Refinement then incorporates these decisions into the application model to automatically generate an implementation in the form of a structural model and quality numbers. The structural model represents the resulting architecture and mapping decisions. Quality numbers are estimated values for different implementation properties, e.g., timing, area, or power consumption.

There are several options to represent specifications. For the application behavior, a well defined Model of Computation (MoC) that allows for analysis and utilization of formal methods is preferred. At the system-level, MoCs used for application modeling include process, data flow, or state-machine models [3]. Beyond a formal nature, executable application models help to avoid ambiguous behavior. Therefore, in our proposed approach, we advocate an executable application model based on a well defined MoC. To specify architectural constraints, platforms are in most cases targeted towards Multi-Processor System-on-Chip (MPSoC) architectures [4]. For structural representation of such implementations, Transaction Level Modeling (TLM) is almost exclusively used today [5].

In general, 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 huge design spaces and increased complexities that can only be managed by orthogonalization of concerns [6]. In

TABLE I
CLASSIFICATION OF DIFFERENT ESL SYNTHESIS APPROACHES [1].

Approach	DSE	Decision Making		Refinement	
		Comp.	Comm.	Comp.	Comm.
Daedalus	●	●	○	●	○
Koski	●	●	○	●	○
Metropolis	—	○	—	○	—
PeaCE/HoPES	○	○	—	●	○
SCE	—	—	—	●	●
SystemCoDesigner	●	●	●	●	—

— no support ○ partial support ● full support

addition to a 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 [1] has shown that various approaches exist that perform decision making and/or refinement for either computation or communication (see Table I). However, none of the investigated approaches can handle comprehensive system-level synthesis with automated decision making and refinement for both computation and communication.

More specifically, synthesis approaches can be divided into two classes: (1) approaches starting with formal, domain specific models (e.g., based on data flow), and (2) approaches starting with implementation oriented models (typically based on programming 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, but lack advantages gained from well defined MoCs, such as automated exploration, mapping, and formal analysis.

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 formal model based and implementation centric synthesis approaches. For this purpose, we introduce well defined intermediate transaction level models as the canonical interface between various formal models and generic implementations. As a proof of concept, we couple representatives from both classes, namely the SystemCoDesigner [7] and the System-On-Chip Environment (SCE) [8] solutions. We concentrate on bus-based MPSoCs, as these architectures are most common, and memory-mapped bus modeling is the focus of the TLM 2.0 standard [9]. The combined design flow allows for fully automatic design space exploration for both computation and communication refinement in a seamless flow from a formal application model down to pin-accurate models of arbitrary, bus-based MPSoCs.

A. Organization

The remainder of the paper is organized as follows: Section II describes related work, followed by an overview of our methodology in Section III. The refinement procedure from a formal application model down to pin-accurate models is explained, and intermediate transaction level models are

introduced in Section IV. The coupling of design flows is presented in Section V. Results of applying the proposed methodology to a typical streaming application case study, a JPEG decoder design, are presented in Section VI. Finally, the paper concludes with summary and outlook in Section VII.

II. RELATED WORK

Many approaches exist today that tackle a subset of what we expect from comprehensive system-level synthesis. In [10], Densmore et al. define a classification framework for such 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 back-end synthesis functionality through either software code generation or C-to-RTL high-level synthesis.

Targeting streaming applications mapped to network on chip architectures, in [11] 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 [12] 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. However, an ideal system-level synthesis tool 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 I). Metropolis [13] 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], on the other hand, supports automatic computation refinement while decision making mostly has to be performed manually. Daedalus [15], Koski [16], and SystemCoDesigner (SCD) [7] 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 plays an important role at the system-level, however, true system synthesis must support both computation and communication.

An approach that automates design space exploration at system-level for MPSoCs based on shared memory communication architectures has been proposed in [17]. 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.

Automatic refinement for both computation and communication is implemented in the System-On-Chip Environment (SCE) [8]. There, however, no support for automatic decision making is integrated and the refinement process starts from

a generic, relatively low-level C-based system model. In this 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.

III. 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 of the system and a platform template. Synthesis results in an implementation as Pin-Accurate Model (PAM), which describes the system as a netlist of task-accurate, bus-functional component models that are ready for further software and high-level synthesis.

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 will follow in the next section.

System behavior is given as a formal application model based on the data flow oriented FunState MoC [18]. Because using only data channels for communication, this MoC is best suited for streaming applications, as found in the multimedia or networking domains. Note that support for other formal MoCs, e.g., synchronous data flow (SDF), Kahn process networks (KPN), or communicating sequential processes (CSP), could be integrated, but is not further discussed.¹ Constraints, on the other hand, are given in the form of a platform, which defines a set of available resources (such as processors, busses, and gateways). Supplemented by allowed connectivity and associated parameters, a platform template describes all possible architectures. Note that only a subset of elements in the template may be chosen for the final architecture during design space exploration. Given a complete system specification, a design space exploration (DSE) engine performs multi-objective optimization to obtain optimized design points, each representing design decisions such as resource allocation and binding information. Decisions of a chosen design point are then fed into a refinement flow that refines the application model down to an Architecture TLM and finally a PAM. Refinement is performed in four phases:

- 1) In a first phase, the application model is refined into a transaction level model, where queues using the 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.
- 2) In a second phase, the Scheduled TLM is refined from the Application TLM. Thereby, computation refinement uses computation decisions from DSE, namely processor

¹In 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.

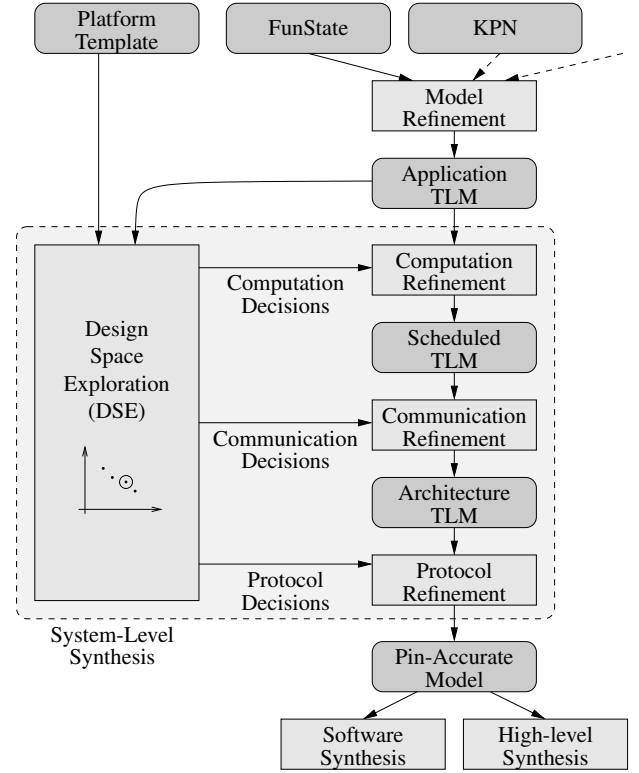


Fig. 2. System-level synthesis overview

allocation, process binding, and scheduling, to generate an intermediate Scheduled TLM. In the Scheduled TLM, processes are bound and scheduled on selected processors, while inter-processor communication is equal to the Application TLM.

- 3) In phase three, communication refinement realizes design decisions about the topology of the communication architecture, routing of memory and synchronization transactions, and bus address and interrupt mapping. During refinement, memory and synchronization transactions are implemented down to the level of arbitrary bus protocol transactions, resulting in the Architecture TLM. In the process, refinement generates bus drivers and bus interfaces in the processors to implement all memory and event communication as bus read, write, and interrupt transactions.
- 4) Finally, through protocol refinement, communication can be 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.

The final PAM serves as input to further back-end high-level and software synthesis tools. Focusing on system-level synthesis aspects, the required model refinement steps from an application model to a PAM will be discussed in more detail in the following sections.

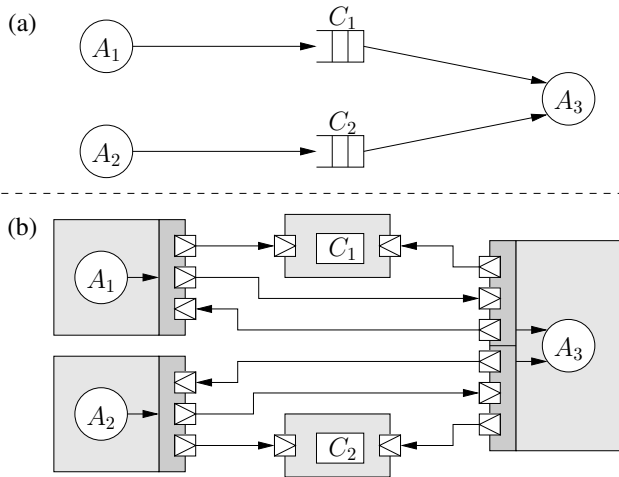


Fig. 3. (a) FunState model and (b) Application TLM

IV. SYSTEM-LEVEL REFINEMENTS

After the overview of the methodology, we now explain the refinement steps in more detail and thereby define intermediate canonical abstraction levels 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 and communication refinement across tool borders. In the following, we will present the necessary model transformations starting with the refinement process from FunState application models.

A. Model Refinement

In support of further synthesis down to general bus-based MPSoC target architectures, a canonical Application TLM is represented in a C-based system-level design language, such as SystemC/TLM [9]. Hence, translation of arbitrary MoCs into synthesizable Application TLMs needs to support corresponding implementation-oriented communication assumptions. In the following, we will exemplarily show the translation of FunState models using a fixed decision for implementing communication as shared memory.² Due to this refinement, the same formal model used in an approach supporting limited target architectures is now appropriable 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.

²Of course, alternative implementations for this intermediate model in the design flow are possible. However, for simplicity, we will only consider models as described. Note that this refinement perfectly fits as input to the subsequent DSE.

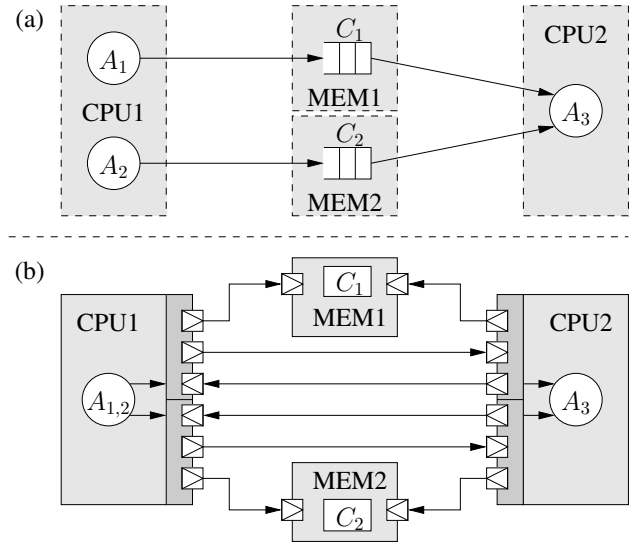


Fig. 4. (a) FunState model with resource mapping and (b) Scheduled TLM

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 into a SystemC module and process with special TLM adapters [19] (indicated as dark gray layer in the figure). These adapters provide an abstract interface to the process for communication and therewith implement the FIFO semantics of the queue in the formal model. An adapter consisting of two initiator sockets and one target socket is generated for each queue connection in the FunState model. One initiator socket is used for memory access whereas the second initiator socket is used to notify the peer actor module about queue updates.

An additional note on parallelization: While we later can map several actors to a single resource, we, for now, do not further parallelize a single actor. So, the initial decomposition of the system in the formal model also defines the maximum parallelism in the final implementation.

B. Computation Refinement

The purpose of system-level computation refinement, from Application TLM to Scheduled TLM, is to partition processes in the application model towards their implementation as hardware accelerator or on software programmable processors. As such, groups of processes are partitioned according to the architecture mapping. End-to-end communication over sockets is implemented the same way as in the Application TLM.

For partition blocks that result in a hardware implementation, system-level computation refinement simply adds another hierarchy level, encapsulating hardware blocks while retaining the inherent parallelism in the model. The result is a single SystemC module for each hardware accelerator. After refining the communication (see Section IV-C), high-level synthesis

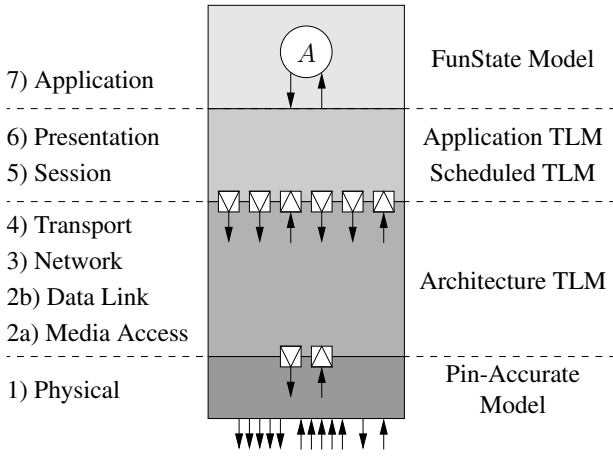


Fig. 5. Communication refinements related to the ISO/OSI model

tools, such as Forte's Cynthesizer, Cadence's C-to-Silicon, Mentor's Catapult, or NEC's CyberWorkBench, can be used to generate RTL implementations for each module.

For software implementations, 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. 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 the example in Fig. 4(b) a dynamic scheduler implemented in a single process was chosen for CPU1, and process $A_{1,2}$ is the result. Further software synthesis in the back-end then generates C/C++ code for each process, which is compiled and possibly linked with an operating system to run on the corresponding target processor.

C. Communication Refinement

Abstract communication in the FunState model is refined all the way down to communication over wires in the pin-accurate model in three phases. For all models shown in Section III, the abstraction level of communication is visualized and related to ISO/OSI layers in Fig. 5. The refinement phases are indicated by dashed lines. Communication refinement from FunState model to Application TLM was explained earlier: queue-based communication is refined to shared memory based communication. In terms of the ISO/OSI model, communication in the Scheduled TLM is described at the transport level.

Communication refinement from Scheduled TLM to Architecture TLM is performed in two steps. First, adapters are aggregated for each hardware or software partition block. Aggregation includes insertion of code performing the tasks

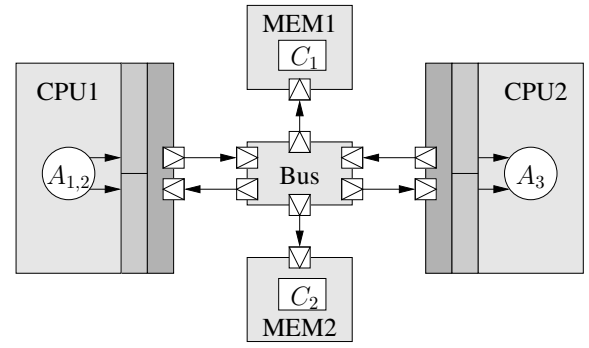


Fig. 6. Architecture TLM

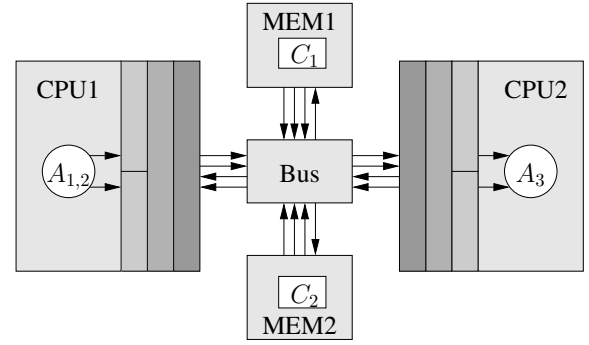


Fig. 7. Pin-Accurate Model (PAM)

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 with a pair of sockets for communication each.

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. 6.

In a final communication refinement phase, called protocol refinement, the Architecture TLM is synthesized down to a PAM (illustrated in Fig. 7) 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 high-level, logic and physical synthesis, e.g., for ASIC manufacturing or FPGA-based prototyping.

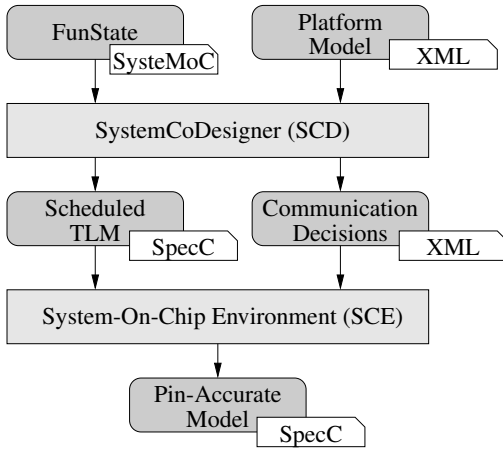


Fig. 8. Tool flow

V. DESIGN FLOW COUPLING

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

Neglecting intermediate steps, Fig. 8 sketches the resulting tool flow and input/output models. The input model for SCD is a FunState model, given as executable representation using the SystemMoC library [20]. SCE's input model is in SpecC [21] format. Constraints are fed into the tools using XML-files.

One feature of SystemMoC is the capability to automatically extract finite state machines, describing the behavior of the actors, and the overall structure of the model. Additionally, functions and other information (such as variable names) are extracted using the Karlsruhe SystemC Parser Suite (KaSCPar) [22]. Following the methodology and model definitions described in Section IV together with computation decisions, this information is used to automatically generate the Scheduled TLM in SpecC format. After generating an implementation oriented model based on the formal FunState model, design decisions made by the exploration framework from SCD are encoded as XML-file. Taking the Scheduled TLM and decisions, SCE refines abstract memory access and synchronization transactions in the model down to bus transactions and interrupts. As a result, SCE generates both an Architecture TLM and a PAM. After briefly describing modeling concepts for SystemMoC in the following subsection, we explain how the SpecC model is generated from the SystemMoC model, incorporating design decisions.

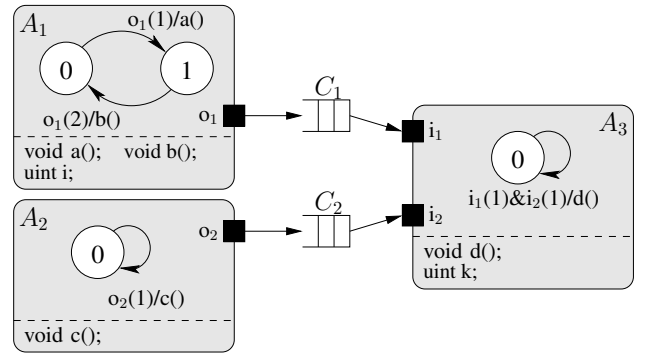


Fig. 9. SystemMoC model

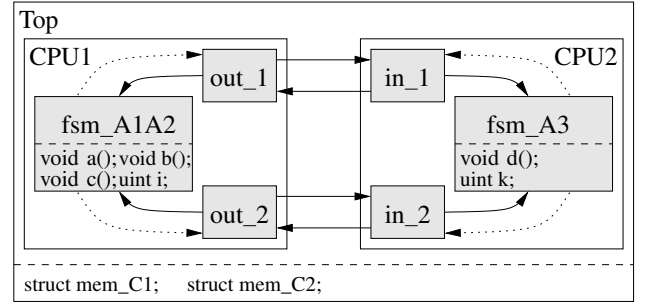


Fig. 10. SpecC model

A. SystemMoC

SystemMoC is a C++-library for implementing formal, executable FunState based representations of embedded systems. An illustrating SystemMoC model for the running example introduced in Fig. 3 is shown in Fig. 9 in more detail. Every actor contains a finite state machine (FSM) with one or more states, shown above the 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.

B. SpecC Generation

In this section, we explain how to generate SpecC programs from SystemMoC descriptions and computation decisions, using the running example. That is, the transformation from the model shown in Fig. 4(a) to the model 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 SysMoC port is implemented as an additional interface process and behavior.
- For each SysMoC channel, memory structures for data, read pointer, and write pointer are instantiated.
- Actions are transformed so that port accesses in the SysMoC 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 for no transition in the current state the activation pattern is fulfilled; Changes on a connected channel provoke reevaluation. As described in Section IV-A, asynchronous communication using abstract queues is implemented with 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. 10. 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.

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 SysMoC example in Fig. 9, actors A1 and A2 are mapped onto a CPU1 whereas actor A3 is partitioned onto CPU2. As a result, a Scheduled SpecC TLM as shown in Fig. 10 is generated. Here, the FSMs of actors A_1 and A_2 are implemented within a single behavior, internally using a round-robin scheduling scheme.

VI. RESULTS

To demonstrate the seamless applicability of the described methodology using the tool flow as shown in Fig. 8, we automatically synthesized a SysMoC 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 original model consists of 14 actors and 22 channels.

System-level synthesis was performed for different mappings of the JPEG application onto an ARM-based multi-processor platform. These implementations result from an

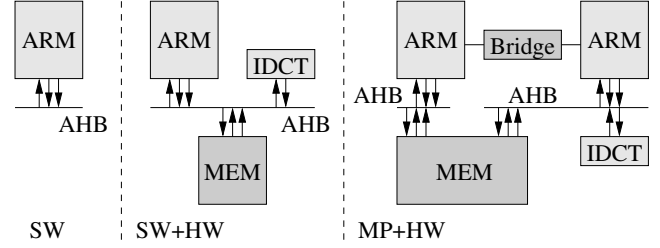


Fig. 11. Implementation architectures

TABLE II
SYNTHESIS AND SIMULATION RESULTS

	Design	TLM			
		App.	Sch.	Arch.	PAM
Simulation time (Hr:Min:Sec)	SW	0:02	0:07	0:14	0:21
	SW+HW	0:02	0:09	3:28	21:23
	MP+HW	0:02	0:09	36:01	4:57:00
Code lines	SW	10945	14882	20835	20165
	SW+HW	10945	15304	21991	21381
	MP+HW	10945	17104	28899	27780

automatic design space exploration, ranging from a software-only implementation on a single processor to an MPSoC architecture using additional hardware accelerators. These design points represent fundamental design decisions for cheap or fast implementations. Exemplary, three resulting architectures are sketched in Fig. 11. The design decisions for the three design points are as follows:

- 1) **SW**: All actors are mapped to a single ARM processor and implemented in software using dynamic scheduling with a round-robin strategy. All communication and synchronization is mapped to the local ARM memory and internal semaphores.
- 2) **SW+HW**: The Inverse Discrete Cosine Transformation (IDCT) actor is implemented as a custom hardware accelerator. Communication queues between the IDCT accelerator and the ARM are implemented in shared memory using interrupt-based synchronization.
- 3) **MP+HW**: Reset and JPEG picture source is implemented on a master ARM processor. Main decoding functionality resides on a second ARM assisted by an IDCT hardware accelerator. All queue buffers are mapped to a dual-ported shared memory and synchronization is implemented via an interrupt bridge between the two ARM processors.

All designs were automatically synthesized down to Architecture TLMs and PAMs from a single, initial SysMoC model and platform template. 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 II shows synthesis and simulation results. We can observe a typical exponential growth of simulation time with

increasing level of detail and accuracy. Runtimes are thereby proportional to the amount of simulated inter-module communication. In a pure software implementation, all communication is local to the ARM. By contrast, the MP+HW design maps all queues to a shared memory and every queue access results in several read or write transactions over the system busses. Compared to the initial model, significant amounts of code and implementation detail, such as middleware, bus drivers, and interrupt handlers, are automatically synthesized within each refinement step. The code size of the synthesized PAM is doubled compared to the Application TLM (App. TLM). Note that Architecture TLMs (Arch. TLM) 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 streaming 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 and platform template only, which becomes more and more common practice in industrial design flows (e.g., using executable specifications).

VII. CONCLUSIONS AND FUTURE WORK

In this paper, we presented an approach towards a comprehensive system-level synthesis solution—to automatically and seamlessly integrate the four basic sub tasks 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 model 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. 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.

With the present work, we proved that comprehensive system-level synthesis of general, optimized MPSoC implementations from abstract, formal application models is possible. In the future, we will extend the DSE engine to better suit the communication refinement capabilities of the SCE design flow, further improving synthesis results. In addition, we plan to investigate tighter coupling of tools, which will enable exploiting pin-accurate models in design space exploration.

REFERENCES

- [1] A. Gerstlauer, C. Haubelt, A. Pimentel, T. Stefanov, D. Gajski, and J. Teich, "Electronic system-level synthesis methodologies," *IEEE Trans. Computer-Aided Design Integr. Circuits Syst.*, vol. 28, no. 10, pp. 1517–1530, Oct. 2009.
- [2] 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, 2007.
- [3] E. A. Lee and A. Sangiovanni-Vincentelli, "A framework for comparing models of computation," *IEEE Trans. Computer-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. Computer-Aided Design Integr. Circuits Syst.*, vol. 27, no. 10, pp. 1701–1713, 2008.
- [5] F. Ghenassia, *Transaction-Level Modeling with Systemc: TLM Concepts and Applications for Embedded Systems*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006.
- [6] K. Keutzer, A. R. Newton, J. M. Rabaey, and A. Sangiovanni-Vincentelli, "System-level design: Orthogonalization of concerns and platform-based design," *IEEE Trans. Computer-Aided Design Integr. Circuits Syst.*, vol. 19, no. 12, pp. 1523–1543, 2000.
- [7] 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 TODAES*, vol. 14, no. 1, pp. 1–23, 2009.
- [8] R. Dömer, A. Gerstlauer, J. Peng, D. Shin, L. Cai, H. Yu, S. Abdi, and D. Gajski, "System-on-Chip Environment: A SpecC-based Framework for Heterogeneous MPSoC Design," *EURASIP JES*, vol. 2008, no. 647953, p. 13, 2008.
- [9] Open SystemC Initiative (OSCI), "Transaction Level Modeling (TLM) Library, Release 2.0," <http://www.systemc.org>.
- [10] 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, 2006.
- [11] R. Marculescu, U. Y. Ogras, and N. H. Zamora, "Computation and communication refinement for multiprocessor SoC design: A system-level perspective," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 11, no. 3, pp. 564–592, 2006.
- [12] E. Bondarev, M. R. V. Chaudron, and E. A. de Kock, "Exploring performance trade-offs of a JPEG decoder using the DeepCompass framework," in *Proc. of the 6th International Workshop on Software and Performance*. New York, NY, USA: ACM, 2007, pp. 153–163.
- [13] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli, "Metropolis: An integrated electronic system design environment," *IEEE Computer*, vol. 36, no. 4, pp. 45–52, April 2003.
- [14] 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 TODAES*, vol. 12, no. 3, pp. 1–25, 2007.
- [15] 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. of the International Conference on Hardware-Software Codesign and System Synthesis*, 2007, pp. 9–14.
- [16] T. Kangas et al., "UML-based multi-processor SoC design framework," *ACM TECS*, vol. 5, no. 2, pp. 281–320, May 2006.
- [17] M. Lukaszewicz, M. Streubühr, M. Glaß, C. Haubelt, and J. Teich, "Combined system synthesis and communication architecture exploration for MPSoCs," in *Proc. of the Conference on Design, Automation and Test in Europe*, Nice, France, Apr. 2009, pp. 472–477.
- [18] 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, 2001.
- [19] J. Gladigau, C. Haubelt, B. Niemann, and J. Teich, "Mapping actor-oriented models to TLM architectures," in *Proc. of Forum on Specification and Design Languages*, Sep. 2007, pp. 128–133.
- [20] <http://www12.cs.fau.de/research/scd/systemoc.php>.
- [21] <http://www.cecs.uci.edu/~specc/>.
- [22] <http://www.greensocs.com/projects/KaSCPar>.