

Efficient High-Level Modeling in the Networking Domain

Christian Zebelein, Joachim Falk,
Christian Haubelt, Jürgen Teich
University of Erlangen-Nuremberg, Germany

Rainer Dorsch
IBM Research & Development GmbH, Germany

Abstract — Starting Electronic System Level (ESL) design flows with executable High-Level Models (HLMs) has the potential to sustainably improve productivity. However, writing good HLMs for complex systems is still a challenging task. In the context of network controller design, modeling complexity has two major sources: (1) the functionality to handle a single connection, and (2) the number of connections to be handled in parallel. In this paper, we will propose an efficient actor-oriented modeling approach for complex systems by (1) integrating hierarchical FSMs into dynamic dataflow models, and (2) providing new channel types to allow concurrent processing of multiple connections. We will show the applicability of our proposed modeling approach to real-world system designs by presenting results from modeling and simulating a network controller for the Parallel Sysplex architecture used in IBM System z mainframes.

I. INTRODUCTION

Moving design flows for increasingly complex systems to a higher level of abstraction allows many forms of verification to be performed much earlier in the design process, thus reducing time to market, and lowering cost by discovering problems earlier. The International Technology Roadmap for Semiconductors (ITRS) has identified executable High-Level Models (HLM) as a key factor to sustainably improve design productivity [1]: Starting from the system specification, an executable HLM is generated. Executable HLMs, if done right, have the advantage to provide a common reference for architectural exploration, hardware and software design flows, and integration & verification processes. For architectural exploration, the HLM is typically augmented with architecture information, i.e., number and type of processing and communication resources including buses and memories. By specifying the HLM mapping to this architecture, and by using estimates for execution times, so called *Virtual Architecture Models* can be generated, permitting a combined functional and timed simulation, and hence, an early performance estimation [2], [3]. After selecting the platform (i.e., an architecture instance) for implementation, the HLM can be reused in various scenarios: In hardware design, the HLM serves as golden model; in software design, parts of the executable HLM are integrated into a virtual prototype, allowing software development to start without the need for a hardware prototype. Moreover, as the HLM has to undergo functional verification, the created verification environment can be reused in hardware, software, and system verification, even alleviating the integration process. As the requirements

differ with each scenario, the development of a suitable HLM is a challenging task.

The high simulation speed makes C/C++ models a first choice for virtual prototypes in the software development, cf. [4]. However, the lack of concurrency particularly prohibits the reuse of such a model in architectural exploration or hardware verification. On the other hand, hardware description languages like VHDL and SystemVerilog are biased towards implementation and make high-level modeling painful by missing object-oriented features.

System description languages like SystemC [5], the de-facto industry standard, enable design and verification at the system level. However, SystemC allows an indefinite number of ways to write an HLM, which makes the reuse of the HLM for different design tasks cumbersome. Restricting SystemC to certain *Models of Computation* (MoC) allows transformations and optimizations to be performed automatically. In this context, system description languages integrating finite state machines (FSMs) with dataflow (DF) models have successfully been used for high-level modeling in ESL tools [6].

However, when applied to designs from the networking domain, such modeling approaches fail due to complexity reasons, of which the most critical ones can be characterized as follows: (1) non-determinism inherent to the arrival of packets from the physical layer and commands from the application, as well as non-determinism caused by concurrent connections, and (2) the complexity of next generation transport layer [7] protocols [8], [9] for a single connection.

In this paper, we propose a high-level modeling approach based on SystemC that solves both; problem (1) by providing *multiport FIFO channels* which can be used to explicitly model non-determinism, and problem (2) by integrating *hierarchical FSMs* with DF models in order to reduce the modeling effort for a single connection. Furthermore, it can often be observed that a high percentage of connections are idle at any given moment in time. Our proposed modeling approach supports this fact by providing *virtual FIFO channels* that allow an efficient mapping of the HLM to a hardware architecture.

The applicability of the proposed approach to real-world designs will be illustrated by modeling and simulating a network controller for the Parallel Sysplex architecture used in IBM System z mainframes.¹

The remainder of the paper is organized as follows: Related

¹Parallel Sysplex and System z are trademarks of International Business Machines Corporation (IBM).

work will be discussed in Section II. In Section III, the basic modeling approach presented in [6] will be reviewed. We will identify some shortcomings of this approach in Section IV. Subsequently, the extensions to the modeling approach will be discussed in Section V. In Section VI, the Parallel Sysplex case study will be presented.

II. RELATED WORK

One of the first modeling approaches integrating Finite State Machines (FSMs) with dataflow (DF) models is **charts* (pronounced "star charts") [10]. The **charts* approach allows an arbitrary nesting of dataflow graphs and FSMs by either refining dataflow actors by FSMs or by refining states of an FSM by dataflow graphs. For this purpose, **charts* assume a "black box" approach, i.e., on each level of hierarchy, a set of connected actors is described, while actors are treated as black boxes. Some Model of Computation (MoC) is selected to govern the actor interaction on each hierarchy level. This, however, requires that actor interfaces conform to some standard accepted by the outer MoC. Scheduling dataflow subgraphs by FSMs is thus very hard to describe.

Other modeling approaches integrating FSMs with dataflow models are SystemMoC [6], the California Actor Language (CAL) [11], and Extended Codesign Finite State Machines (ECFSMs) [12]. SystemMoC is a limitation of the FunState Model of Computation [13] to the integration of FSMs with dataflow models. In FunState, controlling the execution of actors through FSMs is the basic modeling concept. FunState also defines the integration of hierarchical FSMs. On the other hand, FunState is only a coordination language which is not executable by itself. SystemMoC, however, is based on SystemC which permits the simulation of the HLMs.

In contrast to CAL and ECFSMs, SystemMoC [6] splits the state of an actor into two parts: While one part is represented by a certain assignment of the member variables of the actor, the other part is the explicit actor FSM state. This distinction enables the mathematical expression of the separation of *computation* (i.e., the manipulation of member variables and the calculation of token values produced) and *communication* (i.e., the number of tokens consumed and produced), allowing design tools to analyze and optimize the model.

III. INTEGRATING FSMs WITH DATAFLOW MODELS

SystemMoC is a high-level modeling approach based on integrating FSMs with dataflow models. It allows high-level performance estimation, automatic hardware and software generation, and prototyping [6]. Thus, this modeling approach satisfies many of the requirements for HLMs, which is why we use it as starting point for our proposed modeling approach.

In SystemMoC, an application is modeled by a network of *actors*, which are connected by *channels*. Actors are only allowed to communicate with each other by means of these channels.

Definition III.1 (Channel) A *channel* is a tuple $c = (I, O, n, \mathbf{d})$ containing channel ports *partitioned* into channel input ports I and channel output ports O , its *buffer size*

$n \in \mathbb{N}_\infty = \{1, 2, 3, \dots, \infty\}$, and also a possibly empty *sequence* $\mathbf{d} \in D^*$ of initial tokens.²

In the basic SystemMoC model, only ordinary FIFO channels are available. An instance c of this channel type is restricted to have a single reader and single writer, i.e., $|c.I| = |c.O| = 1$.

The basic entity of data transfer are *tokens* carrying data values of some data type, which are transmitted over these channels. Consequently, an actor contains *actor ports* partitioned into *actor input ports* and *actor output ports* which are connected to channel ports and enable the actor to consume and produce tokens.

An actor may contain member variables, contributing to the actor's state. In order to access or modify these variables, two special types of member functions can be used: (i) *guard functions*, which query (but can't modify) the variables and thus represent general predicates on the state of an actor, and (ii) *action functions*, which may also modify the variables.

The communication behavior of an actor (i.e., the number of tokens consumed and produced in each actor activation) is controlled by the *actor FSM*, which also invokes the guard functions and action functions of an actor:

Definition III.2 (Actor FSM) The *actor FSM* is a tuple $R = (Q, q_0, T)$ containing a finite set of states Q and an initial state $q_0 \in Q$. A transition $t = (q, q', \text{req}, \text{cons}, g_{\text{guard}}, f_{\text{action}}) \in T$ from a state $q \in Q$ to a state $q' \in Q$ contains the enabling rate $\text{req} : I \cup O \rightarrow \mathbb{N}_0$ which maps each input/output port to the number of tokens/places required in order to execute the transition, and the consumption rate $\text{cons} : I \cup O \rightarrow \mathbb{N}_0$ ($\forall p \in I \cup O : \text{cons}(p) \leq \text{req}(p)$) which maps each input/output port to the number of tokens/places consumed when the transition is executed. The guard function g_{guard} is a boolean-valued expression over the values of the tokens required on the inputs and the member variables of the actor. Finally, the action function f_{action} determines the values of the tokens which are to be produced on the outputs and may modify the member variables of the actor.

If represented graphically (cf. Fig. 1), $p(\text{req}(p), \text{cons}(p))$ is annotated to a transition. If $\text{req}(p) = \text{cons}(p)$, only $p(\text{req}(p))$ is annotated. If $\text{req}(p) = \text{cons}(p) = 0$, the annotation is omitted.

The operational semantics of an actor FSM can be divided into three phases: First, the current state is checked for an enabled transition t , i.e., sufficient tokens/places must be available on the input/output ports as specified by $t.\text{req}$ ³, and the guard function $t.g_{\text{guard}}$ must evaluate to true. Next, a single enabled transition is selected non-deterministically and executed, i.e., the action function $t.f_{\text{action}}$ is invoked which calculates the values of tokens to be produced based on the

²We use D^* to denote the set of all possible *finite sequences* of tokens $d \in D$, i.e., $D^* = \bigcup_{n \in \{0, 1, \dots\}} D^n$.

³We use the \cdot -operator, e.g., $a.I$, for member access of tuples whose members have been explicitly named in their definition, e.g., member I of actor $a \in A$ from Definition ???. Moreover, this member access operator has a trivial extension to sets of tuples, e.g., $A.I = \bigcup_{a \in A} a.I$, which is also used throughout this document.

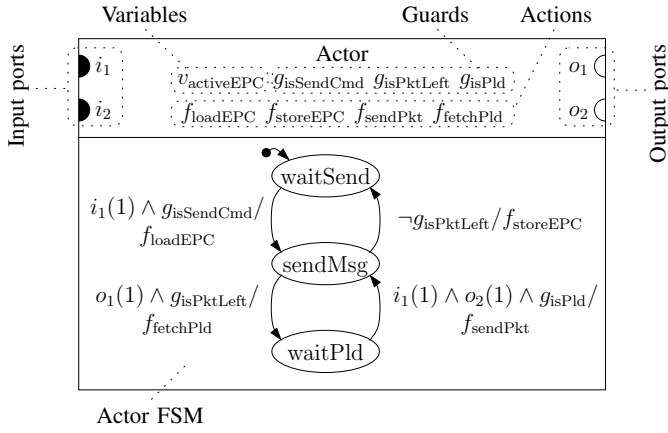


Fig. 1. Actor consisting of input ports, output ports, variables, guards, actions and an actor FSM controlling the communication behavior.

member variables and input tokens. Finally, tokens/places are consumed from input/output ports as specified by $t.cons$.

Consider for example transition $waitSend \rightarrow sendMsg$ in Fig. 1 which is enabled, if the current state of the actor FSM is $waitSend$, one token is available on input port i_1 , and the guard function $g_{isSendCmd}$ evaluates to true. When the transition is executed, the associated action function $f_{loadEPC}$ is invoked, subsequently consuming the token from i_1 , and transitioning the FSM into the $sendMsg$ state.

IV. HIGH-LEVEL DESIGN METHODOLOGY

With only the basic model available, a high-level model for complex network protocol engines could be derived in principle, as we already showed in [2] by modeling and simulating an InfiniBand Host Channel Adapter (HCA).

In order to identify why this attempt has some serious shortcomings, we will first introduce some basic concepts. We assume that the communication flow between network controllers is tied to connections between *endpoints* (EP) configured by the user (comparable to sockets used by TCP/IP, or queue pairs used by InfiniBand [8]). In order to identify the EP that should handle an incoming packet, each EP is assigned a locally unique *endpoint number* (EPN). The information needed by an EP in order to maintain its state (e.g., mode of operation, destination EPN, number of packets left to send and buffer addresses) is called *endpoint context* (EPC).

Usually, the behavior of a single endpoint is specified in detail by means of a *finite state machine* (FSM). For example, Fig. 1 shows how the sending of a message could be specified for an endpoint of a simple network controller:

- Initially, the FSM is in the $waitSend$ state waiting for a send command issued by the application. Upon arriving of such a packet, the transition $waitSend \rightarrow sendMsg$ is executed, whose action function $f_{loadEPC}$ determines the EPN transmitted in the packet, and marks the corresponding EPC as *active* by loading it into the member variable $v_{activeEPC}$.
- The active EPC will be used for subsequent transitions which are not triggered by incoming packets, but depend only on the EPC. In our example, the guard function

$g_{isPktLeft}$ annotated to transition $sendMsg \rightarrow waitPld$ determines whether more packets have to be send for the current message (segmentation) or not, by evaluating the active EPC.

- Assuming that incoming payload packets do not carry EPNs, the EPC in our example will be finally written back by the action function $f_{storeEPC}$ of the transition $sendMsg \rightarrow waitSend$, i.e., when the $sendMsg$ state is reached, and no more packets have to be sent.

Two sources of complexity can be identified that make it difficult to pursue this modeling approach. First, the FSM for a single EP itself can become very complex. Obviously, hierarchical FSMs as known from Statecharts [14] are a solution, whose integration into the basic model will be presented in Section V-B.

Another source of complexity stems from the fact that more than one endpoint must be supported. However, even the simple FSM presented above deadlocks if EPs are operating concurrently: As the lifetime of the active EPC may span several transitions, other EPs are blocked from execution during this interval. In particular, tokens on channels not determined for the active EP may not be consumed, hiding tokens required by the active EP in order for the actor FSM to make progress. Alleviating this problem by inserting transitions into the FSM such that tokens not determined for the active EP are consumed while the FSM is blocked in a state is no adequate solution either, as this ad-hoc approach is error-prone (especially for large FSMs), and possibly multiple active EPCs have to be managed.

From a modeling point of view, the network controller can be seen as a composite state consisting of N AND components executing concurrently, as known from Statecharts [14], with each component containing the FSM of a single EP, parametrized with the corresponding EPN (cf. Fig. 2a). An advantage of this approach is that the FSM for a single endpoint has to be written only once, and can be subsequently instantiated N times.

However, this approach mixes the concurrency model with the automata semantics, i.e., the question is: when do the N concurrent EP FSMs make state transitions relative to each other? Normally, the answer is chosen from one of the following two extremes: (1) true parallelism for concurrent

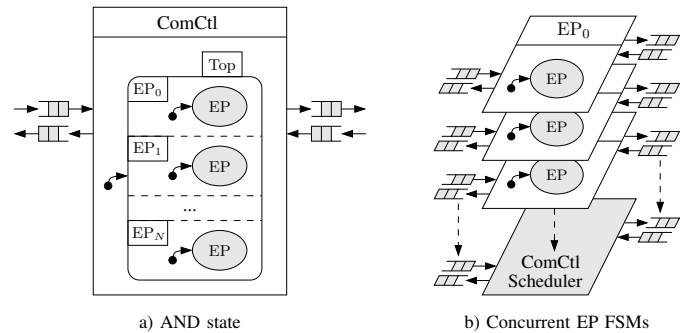


Fig. 2. Network controller modeled by a) an AND composite state, and b) concurrent EP FSMs.

transitions as known from hardware description languages and (2) sequential execution for concurrent transitions, as known from the execution of synchronous/reactive languages compiled into software. Both alternatives represent extremes for the available system resources: (1) corresponds to a number of execution engines $M \geq N$, and (2) to $M = 1$. In the general case, i.e., $1 < M < N$, neither (1) nor (2) applies. Furthermore, the semantics of (1) implies that the execution time of the parallel execution of concurrent transitions is equal to the execution time of the longest concurrent transition. This constraint demands that the functionality is subdivided into transitions with roughly equally execution times in order to improve hardware utilization. However, this approach is problematic, as the execution times for different virtual architectures can vary significantly requiring different decompositions of the functionality for different architectures. This violates the separation of architecture and functionality.

By decoupling the concurrency model from the FSM semantics [10] these problems can be cleanly solved. We choose the sequential semantics of (2) for FSM execution, and add parallelism via a well defined concurrency model like dataflow. Architecture-specific behavior like timing annotation [15] is implemented by schedulers representing the hardware platform onto which these FSMs are mapped (cf. Fig. 2b).

V. MODELING EXTENSIONS

In order to support the modeling approach outlined in the previous section, we will introduce two additional communication primitives in the following, which can be used along with the FIFO channels already available.

A. Communication Primitives

In order to explicitly model non-determinism, we extended the basic SystemMoC model by a channel type called *multiport FIFO channel*. An instance c of this channel type can have more than one reader and writer, i.e., $|c.I| \geq 1$ and $|c.O| \geq 1$ (cf. Fig. 3b). In contrast to the FIFO channels available in the basic model, *conflicts* as known from Petri nets may occur if multiple transitions requiring tokens or places from the same multiport FIFO channel are enabled at the same time. In order to resolve these conflicts, a scheduler may be provided (e.g., by the implementation of the underlying hardware platform), which controls the visibility of tokens/places for each port connected to the channel. For example, a limited resource can be modeled by placing $N \geq 1$ initial tokens on a multiport FIFO c . Subsequently, the resource can be requested by requiring a token from c . When the resource is available, the transition requesting the resource will become enabled, and subsequently be executed. Finally, when the resource is no longer needed, the actor produces a token onto c .

As outlined in Section I, systems in the networking domain usually have to process interleaved streams of tokens belonging to specific connections. On the other hand, our design methodology presented in Section IV proposes that concurrent connections should be modeled by concurrent FSMs, each implementing the behavior of a single endpoint (cf. Fig. 2b). In this case, each port of an endpoint could be

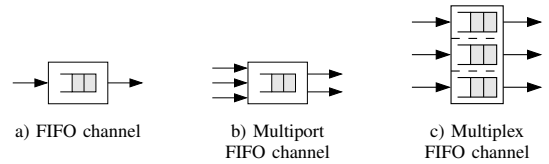


Fig. 3. Available communication primitives

connected to its own FIFO channel, which then contains only tokens belonging to the corresponding connection. While this approach reduces the modeling effort as token streams are no longer interleaved, it can often be observed that a high percentage of connections are idle at any given moment in time. Therefore, in the general case, the underlying hardware platform will not provide separate FIFOs for the maximum number of endpoints. In order to efficiently map the HLM to a hardware architecture, the modeling infrastructure must support this interleaving of token streams.

For this reason, we extended the basic SystemMoC model by a channel type called *multiplex FIFO channel*. An instance c of this channel type provides n virtual FIFO channels c_1, c_2, \dots, c_n , i.e., $|c.I| = |c.O| = n$ (cf. Fig. 3c with $n = 3$). A token produced onto a virtual FIFO channel c_i is assigned the index i , and is subsequently multiplexed in a way deemed suitable by the underlying hardware platform. For example, the scheduler provided for a multiplex FIFO channel could query its input ports in a round-robin fashion, multiplexing new tokens into a single stream of tokens representing a *single* hardware FIFO. On the other hand, the output port of a virtual FIFO c_i will only see the tokens annotated with the index i , retaining the advantage of non-interleaved token streams.

B. Hierarchical FSMs

In order to cope with the complexity found in the specification for a single endpoint, hierarchical FSMs are a solution. These are characterized by the fact that states can contain other automata. We extended the basic SystemMoC model by the two most important super-states known from Statecharts, namely XOR and AND states. In the case of XOR decomposition, the FSM is in exactly *one* of its substates, whereas in the case of AND decomposition, the FSM is in *all* of its substates.

Compared to transitions of the basic actor FSM (cf. Definition III.2), transitions in the hierarchical actor FSM are extended in the following way:

- The target state $t.q' \in Q$ is replaced by a *set of target states* $t.Q' \in Q^*$ (with Q^* denoting the set of all states of the hierarchical actor FSM). This allows for the specification of a certain combination of active states, if an AND state is entered. If an AND state is entered and no specific substate is specified for a component, the initial state of the component will be chosen as active state.
- An additional element, $t.Q_{\text{cond}}^+ \in Q^*$ is added to t , representing a *set of conditional states*. In this case, the FSM must be in each $q \in t.Q_{\text{cond}}^+$ in order to enable the transition.
- Another element, $t.Q_{\text{cond}}^- \in Q^*$ is added to t , also representing a *set of conditional states*. However, in this

case, the FSM must not be in any $q \in t.Q_{\text{cond}}^-$ in order to enable the transition.

The operational semantics of the hierarchical actor FSM is defined in terms of the non-hierarchical actor FSM from Definition III.2. Please note that the semantics of an AND state in our model is purely sequential, i.e., transitions of different components will always be executed sequentially in the flattened actor FSM.

VI. CASE STUDY

In this section, we will show how a complex network controller for the Parallel Sysplex architecture was modeled using the design methodology presented in Section IV. Subsequently, some experimental results will be presented.

A. Parallel Sysplex Overview

A Parallel Sysplex cluster [16], [17] consists of a number of servers which appear as a single system to users or applications. Each system contained in the cluster runs one or more instances of an operating system (OS), and is connected to a coupling facility (CF) that provides shared data structures (e.g., cache structures, list structures and lock structures), which can be used to synchronize the systems. In order to achieve a high degree of fault tolerance, two CF instances are typically employed in such a cluster (cf. Fig. 4a).

If an OS wants to read or modify a shared data structure in the CF, it has to send a *primary message* to the CF. After completion of the primary message, the CF may in turn send *secondary messages* to the OS instances determined to have interest in the data structure that has been accessed or modified.

Fig. 4b shows the processing of a primary message with write data: First, upon receiving a WRITE command, the OS transfers a *message command block* (MCB) to the CF.

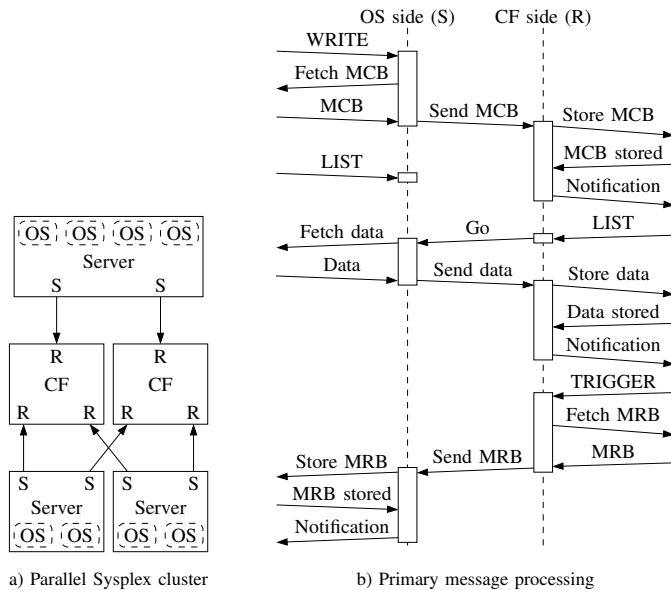


Fig. 4. a) Parallel Sysplex cluster consisting of three servers and two coupling facilities, and b) processing of a primary message

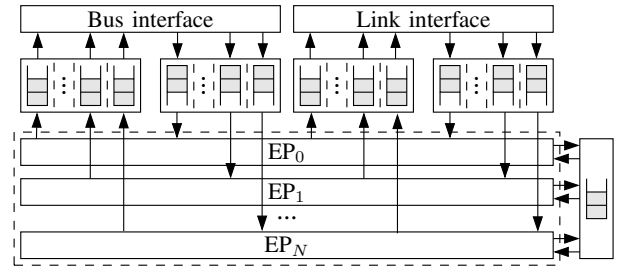


Fig. 5. HLM of the Parallel Sysplex protocol engine

Subsequently, when both sides have received LIST commands specifying where the data should be fetched/stored, the data will be sent from the OS to the CF. After data transfer is complete, the CF issues a TRIGGER command which sends a *message response block* (MRB) back to the OS.

In contrast to primary messages, secondary messages are initiated by the CF and transmit only MCB and MRB, i.e., no data is transmitted by this message type.

An endpoint in this case can be individually configured to send or receive primary or secondary messages, i.e., there are four different operating modes available for each EP. The number of endpoints to be instantiated can be specified by the user when the simulation is started.

B. Results

The (simplified) structure of the HLM is depicted in Fig. 5. It can be seen that the Parallel Sysplex protocol engine has been modeled using the design methodology presented in Section IV, i.e., by N concurrent EP FSMs represented by actors EP₁ to EP_N. The resulting description of the EP actor consists of approximately 3k physical source lines of code (SLOC) [18], and has successfully been applied to RTL hardware verification and firmware development. The corresponding VHDL model consists of approximately 18k SLOC. In contrast to the InfiniBand HLM which has been modeled using only the basic model [2], we did not encounter any deadlocks during the verification of the Parallel Sysplex HLM in the hardware verification environment.

Multiplex FIFOs are used to connect each EP actor with the bus interface actor and link interface actor, respectively. Multiplex FIFOs are used for the modeling of limited resources representing functional constraints which have to be met by the EP actors: For example, each packet sent to the bus interface for which a response packet will be generated must contain a 5-bit unique tag such that the bus interface can determine the target EP upon arrival of the response packet. In our HLM, this is modeled by a multiplex FIFO connected to all EP actors (cf. Fig. 5), which has a size of 32 and contains initial tokens numbered from 0 to 31. An actor which wants to send e.g., a DMA read request consumes one of these tokens in order to obtain the tag which has to be stored in the packet. When the read response arrives, the actor releases the tag by producing a token with the corresponding number onto the multiplex FIFO.

Although no interactions between different EP FSMs have to be modeled, the resulting hierarchical FSM for a single EP as implemented consists of 206 states partitioned into 147

leaf states, 38 XOR states, and 21 AND states, as well as 241 transitions attached to these states.

In SystemC, the simulation phase is preceded by the so-called *elaboration phase*, during which the module hierarchy is created, and ports are bound to channels. During this phase, hierarchical actor FSMs (if any) are flattened in order to obtain the corresponding basic actor FSM which is used during the simulation.

In case of the hierarchical EP FSM, the corresponding basic actor FSM consists of 1,142 states and 14,537 transitions. Thus, the resulting model consists of more than 140,000 states and nearly 2,000,000 state transitions. On an Intel Core2 Duo E6600 CPU, the flattening for a single EP FSM takes approx. 0.1 seconds, resulting in the following elaboration times:

EPs	1	2	4	8	16	32	64	128
Time [s]	0.09	0.19	0.37	0.75	1.55	3.11	6.28	12.78

Note that, in the general case, the FSM for an actor cannot be reused, even if the same actor is instantiated multiple times, as the actor FSM can be constructed dynamically during the elaboration phase, and thus may be different from instance to instance.

Fig. 6 shows the simulation performance for different numbers of EPs instantiated in the model. Each point in the diagram has been obtained by averaging five consecutive runs with the same parameters, i.e., with the same number of instantiated and utilized endpoints, respectively.

The increase of simulation time for a given number of utilized endpoints can be explained by two facts: First, instantiated but unused EPs still require some initial processing until they are quiesced. Second, the larger performance decrease between 8 and 16 instantiated EPs is probably caused by suboptimal cache utilization in the case of more EPs.

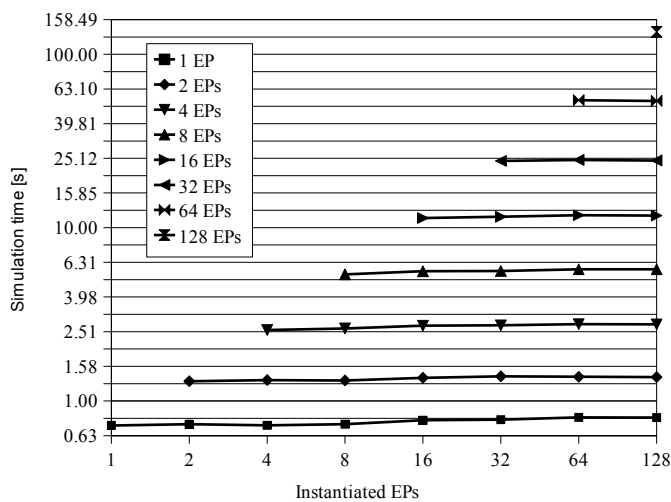


Fig. 6. Simulation times resulting from the transmission of 100 primary messages per EP, each consisting of a 1 KB MCB/MRB and 10 KB payload data. As only 256 bytes of payload are sent in a packet, 48 packets are transmitted per message. Note that in order to utilize a given number of N EPs, at least N EPs must be instantiated in the model.

VII. CONCLUSIONS

Given a system specification, an executable HLM is required for architectural exploration, hardware and software design flows, and integration & verification processes. We proposed a design methodology for systems from the networking domain, which abstracts from the hardware platform by modeling the endpoint behavior by concurrent FSMs, thus avoiding the introduction of deadlocks into the model. In order to support the methodology, we extended the specification language presented in [6] by communication primitives and hierarchical FSMs. We showed the applicability of our proposed modeling approach by implementing and simulating a HLM for a complex Parallel Sysplex network controller.

REFERENCES

- [1] ITRS, "International Technology Roadmap for Semiconductors – Design," ITRS, Tech. Rep., 2005, <http://www.itrs.net/>.
- [2] M. Streubühr, J. Falk, C. Haubelt, J. Teich, R. Dorsch, and T. Schlipf, "Task-Accurate Performance Modeling in SystemC for Real-Time Multi-Processor Architectures," in *Proceedings of DATE*. Munich, Germany: IEEE Computer Society, Mar. 2006, pp. 480–481.
- [3] K. Huang, S. I. Han, K. Popovici, L. Brisolara, X. Guerin, L. Li, X. Yan, S.-I. Chae, L. Carro, and A. A. Jerraya, "Simulink-based MPSoC design flow: case study of Motion-JPEG and H.264," in *Proceedings of DAC*, 2007, pp. 39–42.
- [4] L. Baldez, S. de Pena, and J. Vidal, "The Unified Models Methodology: Applications to Inkjet Printing," in *Proceedings of FDL*, 2007, p. 6.
- [5] T. Grötter, S. Liao, G. Martin, and S. Swan, *System Design with SystemC*. Kluwer Academic Publishers, 2002.
- [6] 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 Transactions on Design Automation of Electronic Systems*, vol. 14, no. 1, pp. 1–23, 2009.
- [7] J. D. Day and H. Zimmermann, "The OSI Reference Model," *Proceedings of the IEEE*, vol. 71, no. 12, pp. 1334–1340, 1983.
- [8] ITA, *InfiniBand Specification*, InfiniBand Trade Association, 2002, <http://www.infinibandta.org>.
- [9] M. Ko, D. Eisenhauer, and R. Recio, "A case for convergence enhanced ethernet: Requirements and applications," in *ICC*. IEEE, 2008, pp. 5702–5707.
- [10] A. Girault, B. Lee, and E. A. Lee, "Hierarchical Finite State Machines with Multiple Concurrency Models," *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems*, vol. 18, pp. 742–760, 1999.
- [11] J. Ecker and J. W. Janneck, "CAL Language Report," University of California at Berkeley, Tech. Rep., Dec. 2003. [Online]. Available: <http://embedded.eecs.berkeley.edu/caltrop/docs/LanguageReport/>
- [12] A. Sangiovanni-Vincentelli, M. Sgroi, and L. Lavagno, "Formal Models for Communication-based Design," in *Proceedings of CONCUR*, August 2000.
- [13] K. Strehl, L. Thiele, D. Ziegenbein, R. Ernst, J. Teich, and M. Gries, "Symbolic Scheduling Based on The Internal Design Representation FunState," *IEEE Trans. on VLSI Systems*, vol. 9, no. 4, pp. 522–544, 2001.
- [14] D. Harel, "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming*, vol. 8, no. 3, pp. 231–274, Jun. 1987.
- [15] M. Streubühr, J. Gladigau, C. Haubelt, and J. Teich, "Efficient Approximately-Timed Performance Modeling for Architectural Exploration of MPSoCs," in *Proceedings of FDL*, Sophia Antipolis, France, Sep. 2009.
- [16] J. M. Nick, J.-Y. Chung, and N. S. Bowen, "IBM System/390 Division: Overview of IBM System/390 Parallel Sysplex - A Commercial Parallel Processing System," in *Proceedings of IPDPS*, Los Alamitos, CA, USA, 1996, pp. 488–495.
- [17] J. M. Nick, B. B. Moore, J.-Y. Chung, and N. S. Bowen, "S/390 cluster technology : Parallel Sysplex," *IBM systems journal*, vol. 36, pp. 172–201, 1997.
- [18] D. Wheeler, "Sloccount," 2009. [Online]. Available: <http://www.dwheeler.com/sloccount/>