I. INTRODUCTION

Driven by the rapid development of microelectronics technology, the functionality and the design complexity of modern distributed embedded systems are continuously increasing. To cope with these challenges, Electronic System Level (ESL) [1] design methodologies introduce higher abstraction and model the complete embedded system as an executable specification at system level. At this level, the decisions such as the partition of functional units to software or hardware are yet to be made. Therefore, Design Space Exploration (DSE) [2] allows for an early evaluation of design decisions and searches for the best implementation candidates. After DSE, the next step is to synthesize an implementation candidate either for the concrete implementation of the product or to validate the design as early as possible by means of a virtual prototype. Currently, there exist numerous synthesis techniques in this field like software compilation, behavioral synthesis, or communication scheduling. However, these are only suitable for parts or subdomains of the system. Existing synthesis flows, thus, typically require the manual disassembling of the system into subsystems, suitable for the synthesis tools. This step is followed by a cumbersome integration process that has to consider the interaction of the subsystems with respect to both computation and communication. The manual system integration becomes a major bottleneck for virtual prototyping in ESL design methodologies.

In this work, we propose a communication-driven disassembling and integration of the system to enable ESL-based automatic virtual prototyping as part of an ESL design flow depicted in Fig. 1. We observe that existing synthesis tools are suitable for particular components like processors or hardware accelerators and a particular communication protocol like shared-memory communication. In the complete system, numerous communication domains exist: Signal-based communication, on-chip and shared memory buses, or even field buses like LIN, CAN, or Ethernet. The key idea of the proposed communication-driven virtual prototyping is to consider the points at which data is transferred between different communication domains with respective communication protocols. In the actual system implementation, these points correspond to real hardware or software components, serving as bridges between the protocols. Our main contributions are as follows: (I) We explicitly introduce these bridges already in the ESL model to consider their characteristics like cost and delay, and allocate the bridges only when data is routed across them. (II) Based on the bridges, we automatically disassemble the system into subsystems, each suitable for an available synthesis tool. (III) After all subsystems are synthesized, we perform system integration where the allocated bridges are specifically synthesized to translate between the respective communication protocols of the subsystems. In order to realize the proposed bridge concept we have developed several synthesis compiler tools integrated into our ESL design flow. As a proof of concept, we apply our proposed design flow to a distributed control application to investigate its applicability and resulting productivity gain.

The remainder of this paper is structured as follows: Section II discusses related work. In Section III, an overview of the ESL design flow is given. Section IV presents the proposed communication-driven virtual prototyping methodology. The results of the virtual prototyping of a networked control system are presented in Section V before the paper is concluded in Section VI.

II. RELATED WORK

The use of virtual prototyping in ESL design methodologies is increasing. Most existing virtual prototyping tools, such as CoWare and CoMET-MET from Synopsys [3], OVPsim from Imperas [4], etc, support the integration of transaction level models written in SystemC as the prototypes. However, none of them allows the automatic code generation from a
A virtual communication and interconnect modeling is proposed in [8] to allow first-hand evaluation of the communication architecture. This work concentrates on enabling accurate performance estimation of both communication architecture and communication mapping in the early MPSoC design stages. In [9], an automatic layer-based refinement approach is proposed to enhance system-level communication design. This approach supports a two-stage design flow using automated model refinement towards custom heterogeneous communication networks, which generates tailored network models at various levels of abstraction. Other works like [10] [11] [12] try to synthesize communication implementations from TLM to universal RTL interfaces. To us, a network is just a subsystem connected with other subsystems through bridges. Therefore, instead of communication design, we are focusing on acceleration of virtual prototyping through automatic system decomposition and composition via bridges.

There are works that concentrate on the generation of interfaces between components of a system. The work in [13] offers modularity, flexibility and scalability in the design flow, in which architectural components like processors and hardware IP-blocks are instantiated from a high-level system specification. The presented design flow focuses on the generation of hardware/software wrappers that adapt the processor to the communication network, but it does not support to enable a virtual prototyping environment that contains components crossing different abstraction levels. In contrast, our framework allows simulation of a system at different abstraction levels.

There exist many works targeting virtual prototyping for embedded applications. In [14], a methodology for the automatic generation of heterogeneous MPSoC virtual prototypes is proposed. Based on a given mapping, the communication within the application model is refined to transactions in the virtual prototype, which is generated by assembling cycle-accurate processor models, memory models, and bus models. This so-called two-step design approach lacks the support for systems that have hardware accelerators and is mainly designed for streaming application models. A complete coverage of the ESL design flow is claimed by some approaches, such as the SER (Specify-Explore-Refine) methodology [15], which accepts algorithmic specification models written in SpecC and enables DSE with HW/SW partitioning, network topology design, bus protocol selection, and bus interface design. In contrast, our system-level model has a well-defined model of computation and uses SystemC as modeling language, which is the industry standard for system-level design.

### III. ESL Design Methodology

This section gives a short introduction of the ESL design methodology used in this work. We use SystemC as modeling language, since SystemC has been established as the de facto industry standard language for system-level modeling with numerous industrial and academic tools supporting it. The overall ESL design flow is shown in Fig. 1:

1) Functional modeling and simulation on the system level results in an executable specification of the design in the SystemC environment. From this executable model, a problem graph is automatically extracted.
2) Generation of hardware accelerators for some or all SystemC modules using high-level synthesis tools to generate application-specific resources.
3) Defining and specifying an architecture graph that include all possible resources and connections of resources under consideration. Combining the problem graph and the architecture graph into an exploration model by adding possible bindings of the tasks in the problem graph to the resources.
4) Design Space Exploration for finding optimized implementation candidates from the exploration model.
5) Code generation for virtual prototyping. All contributions in this work are aimed to automatize this step.

**Actor-based Design**

We use actor-based design [16] to construct the executable models. This is realized by building a library upon the standard...
SystemC language. So, instead of using SystemC modules directly, we have actors to represent the basic computation units. Actors are concurrent components that are connected with others through channels (e.g., FIFOs). Functions of an actor are triggered by its Finite State Machine (FSM). The key advantage of actor-based design is that the interaction between actors follows some kind of communication pattern, called Model of Computation (MoC) [17]. A certain MoC is given by a predefined type of communication behavior and a scheduling strategy for the actors. Separating actor computation and actor communication gives the designer the ability to refine the communication at different abstraction levels [18]. Thus, the model is ideal for ESL design, since at the system level, the decision about the hardware/software partitioning has yet to be made.

Design Space Exploration

The goal of DSE is to find an optimized implementation solution through exploration of implementation candidates. The input of DSE is an exploration model which is defined by the problem graph derived from the executable model and the architecture graph derived from the architecture provided by the system designer, as well as mapping constraints between the two graphs. From this exploration model, various implementations can be derived by determining the allocation of the resources, the bindings of the actors and channels to the different resources, and the routing of the communication. For each implementation, multi-objective optimization [19] is used to evaluate the implementation quality. The results of DSE is a set of high quality candidates as optimized implementation solutions.

Given is a problem graph \( G_p(P, E_p) \) with circles representing actors and black dots representing FIFO channels. The edges are used to represent data communication. In Fig. 2, there are 4 tasks, in which \( p_1 \) and \( p_2 \) are computation tasks and \( p_{1-2} \) and \( p_{2-1} \) are communication tasks. This problem graph also contains 4 communication edges. The architecture graph is given as a directed graph \( G_R(R, E_R) \). \( R \) presents resources such as processors, memories, hardware accelerators, buses, and, in particular, the mentioned bridges \( R_B \subset R \). \( E_R \) indicates available communication connections between two resources. Additionally, a set of mapping edges \( E_M \subseteq P \times R \) denotes possible executions of tasks on resources. Note that bridges are initially placed into the architecture by the system designer during the creation of the architecture. For example, if two connected resources belong to two different domains (e.g., software subsystem and hardware subsystem), one bridge component must be inserted between them (see \( r_b \) in Fig. 2).

DSE searches optimized implementation candidates, with an implementation being defined as following:

**Definition 1 (Implementation Candidate):** An implementation candidate (Fig. 3) consists of (i) the original problem graph \( G_p \), (ii) an induced allocation graph \( G_R = (R, E_R) \subseteq G_R \) of the allocated resources with \( R_R \subseteq R \) and \( E_R \subseteq E_R \), as well as (iii) a binding \( E_M \subseteq E_M \) that maps each task to a resource.

Deriving an implementation candidate generally involves allocating resources, binding tasks, and routing transactions. If the system is distributed, network exploration is also performed, which decides (1) the communication pattern, (2) synchronization between different communication nodes (some buses have built-in synchronization protocols), and (3) other requirements, e.g., scheduling parameters for real-time communication.

IV. METHODOLOGY OF AUTOMATIC VIRTUAL PROTOTYPING

Based on the results of DSE, the designer can select one implementation candidate to start automatic virtual prototyping. Given an implementation candidate \( (G_p, G_R, E_M) \), the question arises how to apply code generation directly from the system level with minimum manual effort. In this section,
we first present automatic system disassembling that decomposes the ESL model into subsystems. Afterward, automatic code generation for each subsystem is performed by invoking subsystem synthesis. Finally, the bridge components are once again used to integrate the subsystem to the complete virtual prototype. All of these steps are integrated into the ESL design framework presented in the previous section.

(1) System Disassembling

The first challenge of an automatic realization of an ESL model is the automatic disassembling of the system into subsystems which are the suitable input for the respective synthesis tools. The principle for disassembling a system into subsystems is a communication-driven decomposition, which defines tasks belonging to different subsystems, only if (i) there is communication between these tasks, (ii) these tasks are mapped onto different resources, and (iii) they are handled by different synthesis tools. In Fig. 3, these two tasks along with their assigned resources must belong to two subsystems. Furthermore, the bridge between these two resources must be divided into two instances so each subsystem has one instance.

Given the graph-based implementation candidate outlined before, we define a subsystem as follows:

**Definition 2 (Subsystem):** A subsystem consists of i) an induced sub-graph $G'_R = (R', E_{R'}) \subseteq G_R$ of the allocation graph $G_R = (P, E_R)$ and ii) an induced sub-graph of the problem graph $G'_P = (P_s, E_{P_s}) \subseteq G_P$, where it holds $\forall p \in P_s : (p, r) \in E_M$ with $r \in R'$, i.e., all tasks mapped to the resources in the subsystem.

Following the proposed idea of a communication-driven disassembling, we employ the bridges $R'_B \in R'$ in the allocation graph to automatically derive all subsystems of a given implementation candidate. This procedure consists of three steps:

1. Removing all bridges $R'_B$ from the allocation graph $G'_R$ results in an allocation graph that decays into isolated sub-graphs.
2. For each isolated sub-graph in the allocation graph, we derive the induced problem graph, resulting the creation of a subsystem.
3. Re-introducing each bridge $b \in R'_B$ from the original allocation graph $G'_R$ into each subsystem that had a connection to the respective bridge before the disassembling.

For the application shown in Fig. 3, two subsystems are derived in Fig. 3 (1) according to the proposed procedure. During this process, there is no manual work involved to divide the system into two parts, since system disassembling is just an automatic process after DSE. Note that the third step of this procedure results in the same bridge being present in several subsystems at the same time. However, a synthesis tool will not generate a complete bridge that synthesizes all communication protocols required for the bridge. Hence, it will only create a stub of the bridge to the considered subsystem and its communication protocol. Therefore, we will synthesize each actual bridge in the integration step by combining all respective stubs.

(2) Subsystem Synthesis

In order to offer fully automatic subsystem synthesis, several compiler tools developed by us are combined with domain specific third-party synthesis tools to realize subsystem synthesis in our ESL design flow. Subsystem synthesis is divided into four phases.

**Platform Synthesis** The first phase of subsystem synthesis is the generation and the configuration of the virtual prototype platform by our platform synthesis compiler tool. This tool (1) instantiates the necessary TLM components from the component library, (2) configures these components (e.g. size of memory) based on the specification of the architecture, and (3) invokes software synthesis. The key part of our platform synthesis is an algorithm that translates routing information into addresses used by the TLM interconnect. This translation reserves for each communication task an address range used for storage of the data (tokens) in the memory onto which the communication task is mapped. Furthermore, the TLM Bus components are parameterized in such a way that the computation tasks access memory on the CPU components according to the routing information specified by the DSE. For simulating the virtual platform, we use the Open Virtual Platform (OVP) [4] from Imperas which provides instruction set simulators (ISS) for processor types like ARM, MIPS, MicroBlaze softcore, etc. If multiple programs are assigned to the same processor, scheduling strategies have to be given.

Example 1 Automatic generated implementation for subsystem1 after platform synthesis. Each component is connected through TLM sockets with configured routing information.

```java
Example 1
```

```
Example 1
```
Example 2 Automatic generated synthesizable SystemC for subsystem2. Components are connected via hardware signals.

class bridge_Sub2 : public sc_core :: sc_module {
  smoc_port_in < type2 > inport;
  smoc_port_out < type2 > outport;
  smoc_fifo < smoc_port_in, smoc_port_out > fifo_fromHW;
  smoc_fifo < smoc_port_in, smoc_port_out > fifo_toHW;
  sc_in < bool > clk;
  sc_in < bool > rst;
  ...
};
class smoc_port_in : public sc_core :: sc_module {
  sc_signal < bool > rd_enable;
  sc_signal < type2 > rd_data;
  sc_signal < sc_unit > rd_offset;
  sc_signal < sc_unit > rd_tokens;
  sc_signal < sc_unit > rd_commit;
  ...
  void reset() { /* reset port (tokens) */ }
  void get() { /* reads the data */ }
  void tokens_event() { /* indicates the change of the number of tokens */ }
  int token() { /* return the number of tokens */ }
  void commit() { /* commit the changes and invoke FIFO */ }
};
class bridge_Sub2 : public sc_core :: sc_module {
  ... similar to smoc_port_in */
};
class smoc_fifo : public hardware FIFO */

class HW : public sc_core :: sc_module {
  sc_in < bool > clk;
  sc_in < bool > rst;
  smoc_port_in < type2 > inport;
  smoc_port_out < type2 > outport;
  ...
};
class subsystem2 : public sc_core :: sc_module {
  smoc_port_in < bool > clk;
  smoc_port_in < bool > rst;
  hw::hw;
  bridge_sub2 b2;
  hw::clk(clk);
  hw::rst(rst);
  hw::import(b2::fifo_toHW);
  hw::output(b2::fifo_fromHW);
  ...
};


tem1 in Fig. 3.

Software Synthesis The goal of software synthesis is to produce software code (e.g. C/C++) which can be simulated by an ISS inside a processor model. Our software synthesis compiler tool is invoked as part of the platform synthesis. The key part of software synthesis is a source-to-source compiler that applies several transformations on the actor-based ESL model such as: the replacement of SystemC FIFO ports of an actor by pointers to software FIFO interfaces; the finite state machine is encoded as a switch-case statement; all the SystemC data types have to be converted. After transformation, the code for each task is assigned to the corresponding resource (according to DSE) by the platform synthesis framework. For example, in Fig. 3 (2) the code for computation task p1 is stored in the memory MEM. The two communication tasks p1→2 and p2→1 inside the bridge r1b are synthesized to two software FIFOs, which are available in our component library.

High-level Synthesis To implement a certain subsystem to hardware implementation, one can use high-level synthesis tools. Since our ESL models are built upon SystemC using actor-based design, we use a two-step approach to generate a hardware implementation: The first step is to synthesize an actor-based model to synthesizable SystemC using the compiler introduced in [20]. Then, we use the Forte Synthesizer [5] from Forte Design System to generate the hardware implementation. This tool accepts systems built in SystemC and automatically generates SystemC/Verilog code at RTL or behavior level. A key feature of this tool is that modules written at behavior level communicate with the rest of the system through auto-generated interfaces. This gives the designer the ability to freely switch the level of abstraction for the hardware simulation, which enables functional or cycle-accurate simulation for hardware implementation. We utilize this functionality by synthesizing a bridge that is located at the hardware side to behavior-level SystemC. This gives us the ability to later simulate the hardware implementation with the rest of the system at different implementation levels.

For the example shown in Fig. 3, subsystem2 is first converted to a synthesizable SystemC implementation (Example 2) after the first step of the two-step approach. The computation task p2 is synthesized from actor-based SystemC to synthesizable SystemC HW. The two communication tasks p1→2 and p2→1 inside the bridge r1b are synthesized to two hardware FIFOs, which are available in our component library. This implementation now serves as the input for the second step of the two-step approach, from which the RTL code can be automatically generated by the third-party synthesis tools.

Communication Synthesis Since the target applications in this work are mostly networked, therefore, an ESL model must contain some kind of network model. Based on the decision of DSE, the network of a system is either available in the component library as hardware IP or designed by the designer as custom communication networks. For hardware IPs, wrappers have to be provided. A custom network model must go through subsystem synthesis to become a part of the virtual prototype. As shown in Section II, there exist several works focusing on communication synthesis, which involves not only the code generation, but also the configuration of the network controllers, such as the frame size, address table, communication pattern, etc.

(3) System Integration

In this work, we are not aiming to fully automatize the system integration process, rather to utilize bridges with synthesized interfaces, so that integrating subsystems together becomes less time-consuming. To explain how integration works, see the example shown in Fig. 3 (2). After subsystem synthesis, two bridges r1b (Example 1) and r2b (Example 2) are partially synthesized, which means each bridge has a configured port connected with the respective subsystem. In other words, the only challenge for sending data between subsystem1 and subsystem2 is to determine how to transfer data between the two interfaces (configured ports, e.g. import). Note that the interfaces in the synthesized bridges r1b and r2b already provide high-level APIs (Fig. 3 (2)) for data transmission.

The bridges r1b and r2b can be merged to a single file (see Example 3) to form the final bridge component. The core of a bridge component is the glue logic that transfers data from one port to the other port. The glue logic can be partially pre-programmed for selected communication protocol as templates, although one still has to make changes in each application to customize the characteristics of the current
Example 3 Final implementation of the bridge (see Fig. 3 (3)) for system integration. The glue logic enables data transmission between software FIFO and hardware FIFO.

```c
class bridge : public sc_core { public:
    void process_to_Sub1() {
        // polling enabled */
        while (true) {
            // poll from hw_FIFO
            // construct the right type for Subsystem1 */
            type1 = covert(in);
            c_fifo_toCPU.write(value); /* write to software FIFO */
            import.commit(); /* update hardware FIFO */
        }
    }
    void process_to_Sub2() {
        // polling enabled */
        while (true) {
            // poll from hw_FIFO
            // construct the right type for Subsystem2 */
            type2 = covert(in);
            while (!outport.tokens()) { /* checking available space */
                wait(outport.tokens_event());
            } /* send data to hardware FIFO */
            outport.put(value); /* update hardware FIFO */
        }
    }
};
```

communication, e.g. conversion between big-endian and little-endian. The implementation of the glue logic depends on the communication type chosen for this system. There are four communication types: (1) polling write and read; (2) polling read and interrupt write; (3) polling write and interrupt read; (4) interrupt write and read.

For each communication type, different glue logic can be applied. For example in Example 3, polling glue communication is active. The glue logic has a procedure `process_to_Sub1()` to read data at the interface of `r1`. If subsystem2 sends something to subsystem1, this procedure (1) reads the data from the hardware FIFO by invoking the APIs of the interface, (2) converts the data type suitable for subsystem1, (3) invokes the APIs of the interface of `r1`, to write the data to the software FIFO. The glue logic has another procedure `process_to_Sub2()`. If subsystem1 sends something to subsystem2, this procedure (1) reads the data from the software FIFO, (2) converts the data type suitable for subsystem2, and (3) writes the data to the hardware FIFO and updates its state. If the designer wants to use interrupt in communication, the selected processor model must support Interrupt Service Routine.

After bridges are customized, a working virtual prototype is at hand. To check the correctness and the performance of the generated code, the virtual prototype is simulated in its supporting simulator. In this work, the selected processor model from the OVP library runs in the OVPsim simulator from Imperas. All other parts of the prototype (e.g. bridges) are simulated directly by the SystemC simulation kernel.

### V. Case Study

As a proof of concept, the proposed prototyping methodology is applied to a distributed embedded control system. This system is used to solve the mathematical puzzle Tower of Hanoi. The objective of the puzzle is to move the entire stack of disks to another rod while obeying the rules. The system is designed as follows: Two robotic arms perform the disk lifting and dropping actions. These two arms are governed by two controllers that exchange information through a network and run in parallel (cf. Fig. 4). During system initialization, the two arm controllers will receive movement lists made by the planner. When the process starts, the two arms will follow their movement lists to make the move. We use Matlab/Simulink to model the plant that consists of the mechanical robotic arms and the tower of hanoi. Each arm is connected to the ceiling and has a two-dimensional degree of freedom. At each joint, there is a DC motor to apply torque to the arm body. The robotic arm has a magnetic hand as the end effector to lift or drop a disk by switching the magnetic force on or off.

Since the two arms run in parallel and the sensors on each arm only observe its own position, there is a possibility of collision. For example, arm left tries to give a disk to arm right, at the same time arm right tries to lift a disk from the central rod. To avoid collision, the two arms have to update the information about their position periodically. The duration of the period depends on the mechanical model of the arms and the accuracy of the system.

The control system is modeled using our actor-based modeling language in SystemC (cf. Fig. 6). The following design decisions are made after DSE:

- The computation tasks are carried out through three ECUs. Each ECU contains a processor, a bus, and a local memory. The chosen processor OpenRISC 1000 (OR1K), which has a Harvard architecture, needs two TLM sockets: one for program code and one for data transfer.
- The three ECUs are connected through a network that is based on the industrial Ethernet bus system.
In this work, we propose communication-driven automatic virtual prototyping for networked embedded systems. Our approach is designed to cope with the difficulty of automatic synthesis directly from an ESL model. The key of the proposed approach lies in (1) the automatic disassembling of the ESL model into subsystems suitable for existing synthesis tools, and (2) the semi-automatic assembling of the synthesized subsystems to form the virtual prototype. To accomplish this, we introduce so-called bridges already in the ESL model. These bridges are specifically synthesized to exchange data between the respective communication protocols of the subsystems, e.g. from hardware signals to TLM sockets, or vice versa. As a proof of concept, a distributed control application is presented to give evidence of the proposed technique’s applicability and productivity gain. The selected application is a typical example of a Cyber-Physical System (CPS) which includes mechanics modeling in Simulink, control system modeling and network design in SystemC. The performance of the final virtual prototype is tested via a co-simulation environment. The results of the experiment show that it needs less manual work to generate a virtual prototype directly from the ESL. This case study shows, to enable automatic system decomposition and composition the designer has only to make 4.6% of the total effort for building the initial system.

VII. ACKNOWLEDGEMENTS

The work has been partially supported by EFRE funding from the Bavarian Ministry of Economic Affairs (Bayernisches Staatsministerium für Wirtschaft, Infrastruktur, Verkehr und Technologie) as a part of the 'ESI Application Center' project.

REFERENCES


Since a higher bandwidth yields a lower communication latency but increases system cost.
Fig. 6. Overview of the virtual prototype. (a) Tasks shown in Fig. 4 are mapped on the selected resources followed by DSE. (b) According to communication-driven decomposition, 6 subsystems are formed.


