Actor-oriented Modeling and Simulation of Cut-through Communication in Network Controllers

Liyuan Zhang, Michael Glaß, Martin Streubühr, Jürgen Teich
Hardware/Software Co-Design
University of Erlangen-Nuremberg
{liyuan.zhang,glass,streubuehr,teich}@cs.fau.de

Andreas v. Schwerin, Kai Liu
Industry Sector
Siemens AG
{andreas.schwerin,kai-liu}@siemens.com

Abstract

In many networked embedded systems, crucial factors for correct functionality are the network’s real-time capability, determinism, and reliability. Besides these requirements, a high network performance in terms of throughput and latency is desired. One of the applied techniques to improve performance is cut-through communication. It allows to forward messages as soon as the packet header is processed, decreasing latency in the controller and, hence, increasing throughput of the overall network. This technique, however, is not directly supported by state-of-the-art ESL actor-oriented modeling approaches. These and their underlying formal models typically rely on FIFO-based communication which corresponds to a store-and-forward communication paradigm. In this paper, we propose and evaluate approaches to extend actor-oriented modeling and simulation to consider cut-through communication in network controllers. Based on the Ethernet variant PROFINET from the industrial automation domain, we evaluate the proposed techniques regarding modeling effort and accuracy compared to standard store-and-forward modeling.

1. Introduction

Modern embedded systems are more and more implemented as distributed systems with computational units, sensors, and actuators that interact with each other to a large extent. Opposed to traditional Systems-on-a-Chip (SoCs), the communication infrastructure and its interaction with the implemented applications is a crucial factor for a reliable and high-performance system service. To satisfy requirements like reliable and deterministic communication that can achieve hard real-time, many bus architectures in the networked embedded system domain like FlexRay [Fle] from the automotive area or PROFINET [IEC10] and EtherCat [Eth] from the industrial automation area rely on time-triggered scheduling paradigms with, at least partially, static schedules. The aspect of communication infrastructure, however, significantly increases the complexity when designing such systems.
Nowadays, Electronic System Level (ESL) design methodologies are developed that aim at predicting the impact of early design decisions on the final product. Starting on a high abstraction level with an executable specification, ESL approaches enable early evaluation and verification of design decisions in design flows for SoCs and Multi-Processor SoCs (MPSoCs). These executable specifications are veritably applicable for modeling data-streaming applications from the digital signal processing or multimedia domain [KSS+09].

Extending these techniques to networked embedded systems is a non-trivial task. As mentioned before, many important bus systems include time-triggered activation that contrasts data-driven activation as incorporated in classic ESL approaches. While a solution to this problem is proposed, e.g., by Kutz et al. in [KSH+11], the work at hand aims at considering another important aspect of modern bus systems: Cut-through communication, which is a well-known technique to decrease the delay of a message within a communication hop by forwarding the message as soon as the header information with the destination of the message is processed. While the technique has been applied in large network routers in the past, cf. [AD89], it is nowadays used in industrial bus systems like EtherCat and PROFINET, but also in communication like InfiniBand [Inf10], to keep performance high while ensuring reliability and real-time capabilities. However, existing ESL approaches typically have formal specifications with a Model of Computation (MoC) that is tailored to signal processing and multimedia applications. Their FIFO-based paradigm to exchange data does not natively support cut-through communication but resembles a store-and-forward-based communication.

The work at hand proposes and evaluates an approach to extend an actor-oriented ESL modeling technique based on SystemC [GLMS02] to consider cut-through communication in network controllers. Using the Ethernet variant PROFINET from the industrial automation domain as a case study, the proposed technique is evaluated with respect to modeling effort and accuracy compared to standard store-and-forward modeling.

The remainder of the paper is outlined as follows: Related work is discussed in Section 2. In Section 3, fundamentals for actor-oriented modeling in ESL design approaches are introduced. The proposed approach to consider cut-through communication is outlined in Section 4. Section 5 introduces the Industrial Ethernet variant PROFINET as a case-study, presents a high-level model of a PROFINET controller, and compares the ESL simulation results with a real PROFINET installation from the literature with respect to modeling accuracy. Section 6 concludes the paper.

2. Related Work

In [FHT06], Falk et al. present a library for modeling and simulating actor-oriented models termed SysteMoC. SysteMoC is based on SystemC, the de facto standard for system-level modeling, and adds a Model of Computation (MoC) to develop an analyzable executable specification language. Other approaches similar to SysteMoC are for example the California Actor Language (CAL) [EJ03] and the Extended Codesign Finite State Machines (ECFSMs) [SVSL00].

In [PACT02], the authors propose the system-level modeling of a network switch SoC using SystemC. They model virtual cut-through routing by explicitly considering individual flits. As discussed in Section 4, a similar modeling approach for the bus techniques considered here would result in a tremendous number of individual tokens and events, respectively, leading to significantly increased simulation times. In the Network-on-Chip (NoC) domain, cut-through is subject to several studies, see [KLM06] for a comprehensive survey, but is not considered in high-level
actor-oriented models. To the best of our knowledge, no other high-level actor-oriented modeling approach explicitly targets cut-through communication in real-time bus systems.

3. Actor-oriented Modeling Fundamentals

In actor-oriented models, actors, which encapsulate the system functionalities, are potentially executed concurrently and communicate over dedicated abstract channels. Thereby, actors produce and consume data (so called tokens), which are transmitted by those channels. Actor-oriented models may be represented as bipartite graphs, consisting of channels $c \in C$ and actors $a \in A$.

**Definition 1 (Network Graph)** A network graph is a directed bipartite graph $G_n = (A, C, E)$, containing a set of actors $A$, a set of channels $C$, and directed edges $E \subseteq (C.O \times A.I) \cup (A.O \times C.I)$ between actor output ports $A.O$ and channel input ports $C.I$, as well as channel output ports $C.O$ and actor input ports $A.I$.

**Definition 2 (Channel)** A channel is a tuple $c = (I, O, n, d)$ containing channel ports partitioned into a set of channel input ports $I$ and a set of channel output ports $O$, its buffer size $n \in \mathbb{N}_{\infty} = \{1, 2, 3, .., \infty\}$, and also a possibly empty sequence $d \in D^*$ of initial tokens, where $D^*$ denotes the set of all possible finite sequences of tokens.

We use SysteMoC, a SystemC based library for modeling and simulating actor-oriented models. The basic SysteMoC model uses a FIFO channel to present unidirectional point-to-point connection between an actor output port and an actor input port. Actors are only permitted to communicate with each other via channels, to which the actors are connected by ports. In a SysteMoC actor, the communication behavior is separated from its functionality, which is a collection of functions that can access data on channels via ports.

**Definition 3 (Actor)** An actor is a tuple $a = (I, O, F, R)$ containing actor ports partitioned into a set of actor input ports $I$ and a set of actor output ports $O$, the set of functions $F$ and the Finite State Machine (FSM) $R$.

The functions encapsulated in an actor are partitioned into actions and guards and are driven by the finite state machine (FSM) $R$ that also represents the communication behavior of the actor (i.e., the number of tokens consumed and produced in each actor activation). An action produces outputs, which are used by the firing FSM to generate tokens for the FIFO channels connected to the actor output ports. A guard just returns a Boolean value and does not have the ability to change the functionality state.

**Definition 4 (Actor FSM)** The actor FSM is a tuple $R = (Q, q_0, T)$ containing a finite set of states $Q$, an initial state $q_0 \in Q$, and a finite set of transitions $T$.

**Definition 5 (Transition)** In an FSM $R = (Q, q_0, T)$, a transition is a tuple $t = (q, k, f, q') \in T$ containing the current state $q \in Q$, an activation pattern $k$, the associated action $f \in a.F$, and the next actor state $q' \in Q$. The activation pattern $k$ is a Boolean function that decides if transition $t$ can be taken or not.
A graphical representation of a SysteMoC actor is given in Figure 1. The actor Store-and-forward models processing of a data frame using a store and forward paradigm. It contains one input port \( \text{in} \) and one output port \( \text{out} \). Thereby, tokens read from \( \text{in} \) and written to \( \text{out} \) represent entire data packets. The FSM of the actor uses two transitions representing the two phases: (1) storing and processing the frame; (2) forwarding the packet. The activation pattern \( \text{in}(1) \) checks if there are enough tokens on the input queue. The activation pattern \( \text{waitFor}(\text{forwardEvent}) \) is used to model execution delay by waiting for a delay that depends on the frame size. Consequently, the actor is either in a state \textit{ready} to accept incoming frames or in a state waiting for the processing delay of the currently processed frame.

4. Considering Cut-through Communication

In the networking domain, there are two basic transmission methods for packet switching, see Figure 2: (1) Store-and-forward switching makes a forwarding decision on a data packet after receiving the whole frame and checking its integrity. (2) Cut-through switching starts the forwarding process soon after examining the destination MAC address of an inbound packet. In theory, a cut-through switch receives and examines only the first 6 bytes of the frame, which carries the destination MAC address. However, to decrease delays by applying cut-through switching, several benefits of store-and-forward have to be sacrificed. In particular, cut-through switches do not have the ability to apply cyclic redundancy check (CRC) to detect bit errors before frame transmission starts. If the CRC check detects an error during transmission, dedicated protocols or strategies may be applied to reduce the wasted bandwidth, see for example [KKP04]. Given these strategies are typically depending on the respective protocol and implementation, the work at hand neglects this aspect.

In state-of-the-art actor-oriented modeling, a data frame, which may be abstracted as a single token, is transmitted and processed in a store-and-forward manner. Therefore, cut-through communication cannot be modeled directly. However, neglecting cut-through communication in a high-level model may result in a significant error with respect to timing and/or functionality. In the following, \footnote{The \texttt{waitFor} mechanism can be implemented using SystemC’s dynamic sensitivity or using the event channel mechanism as suggested in [KSH+11].}
Figure 2: Comparison of store-and-forward (1) and cut-through transmission (2): A data frame consists of a header (black) and a body (gray). Depicted are the following delays: $t_1$ frame transfer time, $t_2$ propagation delay, $t_3$ store-and-forward switch delay, $t_4$ end-to-end latency, $t_5$ cut-through switch delay, and $t_6$ end-to-end latency.

four modeling possibilities to consider cut-through communication are discussed and illustrated in Figure 3.

**Frame Store-and-forward** As mentioned before, a possible abstraction from cut-through communication is relying on store-and-forward. In this approach, the complete data frame is received and processed before the token, i.e., the data frame for the succeeding actor, see Figure 3 (a). The advantage of this approach is that it does not require any changes, neither in the model, nor in the analysis environment typically used together with the model. The drawback is a significant overapproximation of the resulting packet delays. This drawback may, in case of time-triggered bus systems, also result in incorrect functional behavior on the succeeding component since the high-level model assumes the frame will arrive later than it does in the real system. Consequently, a real systems static schedule cannot be applied to the high-level model. This may possibly change orders in buffers and have other side effects.

**Header Store-and-forward** A possibility termed header store-and-forward here still employs store-and-forward modeling but only stores and forwards the headers of the messages. The advantage is, again, that the model and the analysis environment remains unchanged. This technique resolves the overapproximation problem of frame store-and-forward, but implies an underapproximation that depends on the body size since it cannot consider the delay for the frame body in the last actor, i.e., the destination. Moreover, it resolves the problem of frames arriving too late on a succeeding component. However, by not considering the body of the frame, the component itself does not consider the required time to process the body anymore. Thus, incorrect functional behavior is again introduced on the component itself since it becomes ready to process the next frame too early, see Figure 3 (b).
Figure 3: Depicted are three solutions to model cut-through behavior: (a) Frame store-and-forward, (b) header store-and-forward, and (c) blocking state. The actor on the left receives a frame on the input port and changes its state from ready to sleep while invoking the action calculation. Calculation determines the processing delay of the frame encoded by the token and waits for a processing delay. After the delay, a token is produced on port out to forward the frame to the next actor. With this FSM, both frame store-and-forward as well as header store-and-forward behavior on the actor are depicted in (a) and (b), respectively. Note that the functional behavior for (a) on the actor itself is correct and incorrect for (b), while the cut-through behavior of the header is incorrect in (a) but correct in (b). The actor on the right has the same functionality with improved timing behavior compared to the left one. However, it remains in sleep just until the processing time of the header passed. Then it transits to a blocking state and produces a token on port out. During the transition, the actor is blocked until the real body would have been correctly processed. This technique ensures both, correct functional behavior of the actor as well as correct cut-through behavior of the frame.

Frame Splitting Another option for cut-through communication, also used in [PACT02], is to split a frame into several modeling units (in our case tokens). Thus, it is possible to model tokens representing (parts of) the header of a frame and presenting (parts of) the body. A very fine-grained possibility is to represent each byte of a frame as an individual token. This, however, immediately results in a significant increase in the number of tokens in the system and, hence, events in a simulation. It is prospected that this technique will seriously sacrifice simulation speed. A coarse-grained option would be to introduce two tokens per frame: (1) one representing the header (2) and one token representing the body. With this technique, the header can be forwarded correctly and the component can be blocked until the complete frame is processed. However, both fine and coarse-grained frame splitting require a special treatment adding one state to each actor’s FSM. As
a remedy, we propose the following technique that only requires one additional state in the FSM without at least doubling the number of tokens.

**Blocking State**  The cut-through solution favored by the authors is termed *blocking state*. In the FSM of the actor, one additional state is introduced, see Figure 3 (c). Here a token corresponds to an entire frame. After the frame token is received, a sleep state consumes the processing time for the header and forwards a token to the succeeding actor, ensuring a correct cut-through behavior of the frame header. From the sleep state, a transition is taken to a blocking state. This blocking state prevents the component from accepting another frame and completely resolves the drawbacks of the header store-and-forward technique while modeling a correct cut-through behavior. The only disadvantage of this approach is, of course, the change in the model. However, explicitly adding real cut-through functionality with only a single additional state is reasonable.

5. Case Study

The proposed cut-through modeling techniques are tested and evaluated in a network model based on the Ethernet variant PROFINET [IEC10]. First, the fundamental aspects of PROFINET and a developed actor-based high-level model of a PROFINET switch are briefly introduced. Afterwards, the cut-through modeling techniques are comparing with each other based on end-to-end transmission latency. The developed test bench consists of eight PROFINET IO switches with a line topology.

5.1. Actor-oriented PROFINET Switch Model

PROFINET IO is one perspective in the PROFINET standard, which modifies standard Ethernet to realize *Isochronous Real-Time* (IRT), *Real-Time* (RT), and *Non-Real-Time* (NRT) communications. In IRT, cycle times are below 1 ms with a jitter below 1 μs [IEC10]. These values are required for clock synchronized applications in the automation domain like controlling the position and acceleration of a robotic arm (motion control). PROFINET IO separates the data exchange process into communication cycles, see Figure 4, each communication cycle is further divided into

![Figure 4: A PROFINET communication cycle is divided into a reserved interval for isochronous real-time (IRT) communication and an open interval for real-time (RT) and non-real-time (NRT) communication.](image-url)
intervals (or periods) for different kinds of transmissions. The red interval is used for IRT communication within a subnet with maximum precision in a determined order that is specified offline. No communication other than the scheduled one is allowed. Within the orange interval, all RT frames can be transmitted without an a priori schedule. During the green interval, all NRT frames and RT frames queued in the switch are transmitted according to their respective priorities. Within the yellow interval, frames from all queues (except IRT-frames) are transmitted according to their priorities if and only if the transmission of the respective frame ends within this interval. The actor-oriented high-level model of a dual-ported PROFINET network switch is depicted in Figure 5. A frame is received by the receiver actor RX and forwarded to the actor Demux. The actor Demux transmits the frame either to actor RED Relay or MAC Relay, and hereby transmits the frames according to the rules concerning the message type, i.e., NRT, RT, and IRT, and the current communication interval, see Figure 4. The switching of the communication interval is controlled by the actor Scheduler, following a predefined schedule. After the frame is processed by the RED Relay or the MAC Relay actor, the following actor Mux transfers the frame to the actor Switch Fabric, where the switching operation is performed. Afterwards, the frame is transmitted to the actor Queue Handler. The Queue Handler has two possibilities: (1) It can send the frame immediately to the transmittter if the interval is correct and enabled by the protocol, e.g., sending a frame in yellow interval. (2) It can buffer the frame in the sending queues according to the frame’s type and priority, cf. IEEE 802.1Q. The actor TX models the sending of the data frame on the actual physical medium.

5.2. Experimental Setup and Results

As outlined before, abstracting cut-through behavior by standard store-and-forward techniques results in both functional as well as timing errors. In this section, we evaluate the introduced timing error with respect to end-to-end latencies. Moreover, we will compare the simulation results of our
Figure 6: Test bench setup: Eight PROFINET switches connected to each other with full-duplex communication are aligned in a line topology. The controller sends frames periodically to its seven connected devices. After the processing, each device sends data frames back to controller.

The evaluated network consists of eight PROFINET switches that are connected to each other with full-duplex communication in line topology, see Figure 6. One controller sends frames periodically to its seven connected devices. After the processing, each device sends data frames back to controller.

To compare the proposed techniques, the end-to-end latencies that consist of link delays, switch delays, and frame transmission delays are employed. The switch delay can be further divided into transmit-port PHY delay \(t_{ps}^r\) and receive-port PHY delay \(t_{ps}^q\), processing delay \(t_{ps}^p\), and queuing delay \(t_{ps}^q\). Processing delay is the time the switches processes the frame header. Queuing delay is the time a frame waits in a queue until it can be forwarded. Transmission delay is the time required to transmit all bits of a frame to the physical medium and, hence, a function of the frame size. \(t_{bit}\) denotes the single-bit transmission delay. The link delay is the traveling time of a frame from a source port to the destination device’s port through the physical medium and, hence, a function of the cable length. \(t_{bit}^p\) denotes the single-bit propagation delay per meter. Additionally to the delays, \(s_h\) denotes the frame header size while \(s_b\) denotes the frame body size.

Let \(n\) be the number of switches (between two end ports) and \(l\) the cable length. The link delay \(T_L\), the switch delay \(T_S\), and the end-to-end latency \(T_D\) are calculated as follows:

\[
T_L = l \times t_{bit}^p \\
T_S^1 = t_{ps}^r + t_{ps}^p + t_{ps}^q + t_{bit}^r \times (s_h + s_b) + t_s^r \\
T_S^2 = t_{ps}^r + t_{ps}^p + t_{ps}^q + t_{bit}^r \times s_h + t_s^r \\
T_D = t_s^r + t_{bit}^r \times (s_h + s_b) + T_L + n \times T_S + t_s^r
\]

Here, Equation (2) calculates the latency for a store-and-forward switch while Equation (3) calculates the latency for a cut-through switch.

We calibrate our actor-oriented high-level model with reference data from a real PROFINET installation [Zde09]: The average switch delay of the real hardware is 3.342 \(\mu s\). The switch delay inside our PROFINET device (with integrated switch) is, in contrast, set to a typical value that serves as an approximation in early design stages and is 3 \(\mu s\), see [JSW07]. The average single-bit transmission delay is 10 ns, the single-bit propagation delay for Fast Ethernet (i.e. 100 Mbit/s)

\(^2\)Note that this change is made to investigate the error that comes from applying approximated data in early design phases.
is 5 ns (per meter), the average transmit-port delay is 373.5 ns, and the average receive-port delay is 1198 ns. However, $t_w^t$ and $t_r^r$ are set to 0 in our simulation since they are already included in the approximated switch delay according to [JSW07].

On this experimental setup, the cut-through modeling techniques frame store-and-forward (frame SaF), header store-and-forward (header SaF), and blocking state (BS) are compared. Two different frame sizes are investigated: (1) the minimum Ethernet frame size of 64 bytes and (2) the maximum frame size of 1518 bytes. The controller sends frames to device $g$, see Figure 6, with largest distance from the controller. The cable length is given as $l = 7 * 50$ m. The simulation result are given in Table 1. As discussed in Section 4, frame SaF results in an over approximation between about 77% for the smallest frame up to about 488% for the largest frame. On the other hand, header SaF results in an under approximation between about 14% up to about 81% which is surprisingly large. As discussed, the blocking state should result in no error at all, if calibrated with the same values as the real installation. Thus, we decided to include a consideration of approximations in early design phases into the experiments by including an approximated switch delay. For the smallest possible frame, the error introduced by the approximation is relatively large with about 11.5%. However, for the largest possible frame, the approximation in the switch delay is less significant and delivers a low error of only 2.5%.

In summary, the experiment shows a significant error if cut-through communication is not properly reflected in actor-oriented high-level models. Moreover, the experiment shows that the actor-oriented high-level models of a PROFINET switch is capable of delivering decent accuracy, even when calibrated with approximated values.

### 6. Conclusion

In this paper, we proposed and evaluated approaches to consider cut-through communication in actor-oriented high-level modeling and simulation for network controllers. We discussed four variants to adapt and/or overcome standard FIFO-based communication, which corresponds to a store-and-forward communication paradigm, to realize cut-through transmission with minimum modifications in the model. In particular, we favor a variant termed blocking state where only one additional state has to be introduced into the FSM of an actor, enabling correct cut-through behavior without requiring further changes neither in the model, nor in the employed analysis environment. Based on a case study, the proposed techniques are evaluated with respect to their timing accuracy. The case study consists of a presented high-level model of the industrial Ethernet variant PROFINET, where adequate timing evaluation in early design stages is a crucial factor for correct system behavior and efficient offline scheduling. It is shown that approximations based on

<table>
<thead>
<tr>
<th>frame size [byte]</th>
<th>objective</th>
<th>real network</th>
<th>frame SaF</th>
<th>header SaF</th>
<th>BS</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>latency [μs]</td>
<td>31.37</td>
<td>55.59</td>
<td>26.73</td>
<td>27.75</td>
</tr>
<tr>
<td></td>
<td>error</td>
<td>77.20%</td>
<td>14.79%</td>
<td>11.53%</td>
<td></td>
</tr>
<tr>
<td>1518</td>
<td>latency [μs]</td>
<td>147.69</td>
<td>869.83</td>
<td>26.73</td>
<td>144.07</td>
</tr>
<tr>
<td></td>
<td>error</td>
<td>488.94%</td>
<td>81.90%</td>
<td>2.45%</td>
<td></td>
</tr>
</tbody>
</table>
standard store-and-forward come at very high under (up 81%) or over approximations (up to 488%) regarding end-to-end latency and sacrifice the correct functional behavior. Moreover, a comparison of early design approximations regarding the switch delay with a real PROFINET implementation shows that the overall model already has a decent accuracy in terms; highlighting the applicability of high-level modeling in the network controller domain. As future work, we will investigate possibilities to model cut-through behavior solely in the analysis environment, in particular, in the simulation framework, to avoid any modifications in the high-level model.

7. Acknowledgements

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

References


