Automata-Theoretic Modeling of Fixed-Priority Non-Preemptive Scheduling for Formal Timing Verification

Matthias Kauer[1], Sebastian Steinhorst[1], Reinhard Schneider[2],
Martin Lukasiewycz[1], Samarjit Chakraborty[2]

[1] TUM CREATE, Singapore, Email: matthias.kauer@tum-create.edu.sg
[2] TU Munich, Germany, Email: samarjit@tum.de

***Abstract*—The design process of safety-critical systems requires formal analysis methods to ensure their correct functionality without over-sized safety margins and extensive testing. For architectures with state-based events or scheduling, such as load-dependent frequency scaling, model checking has emerged as a promising tool. It formally verifies timing behavior of real-time systems with minimal over-approximation of the worst case delays. In this context, Event Count Automata (ECAs) have become a valuable modeling approach because they are specifically designed to handle typical arrival patterns and integrate well with analytic techniques.**

In this work, we propose an extension of the ECA framework's semantics and use it in a Fixed-Priority Non-preemptive Scheduling (FPNS) model that correctly abstracts the intra-slot behavior in the slotted-time model of the ECA. This is challenging because straightforward implementations cannot capture the full behavior of event-triggered scheduling with such a time model that the ECA shares with most model checking based methods. In a case study, we obtain bounds via model checking a basic model and then our proposed model. We compare these bounds with a SystemC simulation. This shows that the bounds from the basic model are too optimistic – and exceeded in practice – because it does not capture the full behavior, while the bounds from the proposed extended model are both safe and reasonably tight.

I. Introduction

Modern embedded systems are becoming increasingly distributed and heterogeneous at the same time. Typically, they are running two kinds of applications. Best effort applications (such as streaming) need to meet a certain average quality criterion and often require high throughput. Safety-critical tasks, on the other hand, interact with the real world via sensors and actuators and therefore operate under hard real-time requirements. A typical example is the distributed control application in Fig. 1. Here, the sensor reading of the state x_k is transmitted over a bus before the controller computes an appropriate input and communicates it to the actuator which applies it to the system. On the one hand, the sensor-to-actuator delay needs to fulfill a strict threshold, but, on the other hand, high cost sensitivity (especially in the automotive industry) can lead to progressively shrinking safety margins.

Thus, during the design phase, the behavior of the system has to be analyzed in order to guarantee its timing requirements. Since the individual components and shared resources follow diverse scheduling and arbitration policies, this analysis becomes challenging. At the same time it is impossible to give guarantees using simulation, as the worst-case pattern is often obscure and might only rarely occur in practice. In a complex system, it is therefore never certain that the behavior has been sufficiently simulated. As a remedy, the system can be analyzed using formal verification approaches such as model checking. The results of such a formal analysis can yield conservative, guaranteed bounds for the timing behavior with minimal over-approximation. There exist many formalisms and associated model checking tools to represent communication architectures. The applied modeling formalisms, however, determine whether the system behavior can be fully captured in the model.

This work was financially supported in part by the Singapore National Research Foundation under its Campus for Research Excellence And Technological Enterprise (CREATE) programme.

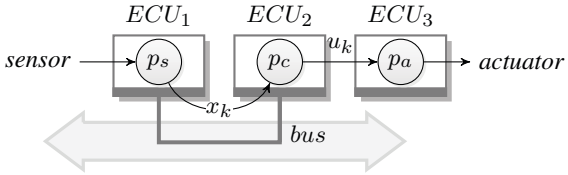

Fig. 1: A distributed control application (p_s, p_c and p_a are sensor, controller and actuator task, respectively).

Usually, there is a trade-off between time-granularity and scalability of the modeling. Timed Automata (TAs), for instance, intend to track the system's behavior with high precision and often result in models that cannot be solved in practice. Event Count Automata (ECAs), on the other hand, have been developed for streaming applications and their inherent uncertainty. As a result, they naturally model data arrival patterns that need to be scheduled and integrate well with analytic approaches. With their slotted-time model – common among most model checking based approaches – they are well-suited for time-triggered systems.

Since communication architectures are often mixed, it is of interest to use ECAs as effectively as possible for event-triggered communication policies such as FPNS. The application of ECAs to these systems has not been investigated yet, however. Towards this, we construct an ECA that is aware of all the possible scenarios that FPNS can lead to under different, but indistinguishable arrival patterns within a time slot.

Our contributions. In this work, we focus on the analysis of FPNS scheduling and, in particular, on one of its most common implementations, the Controller Area Network (CAN) bus, which is the most widely used bus in the automotive industry and other domains.

We (1) analyze why a basic FPNS model does not work if it relies on a slotted timing assumption, using the ECA framework as an example. We then (2) extend the semantics of the ECA framework to explicitly keep track of newly arrived messages and allow non-deterministic update functions. Finally, we (3) propose a FPNS model that utilizes our extended framework to accurately capture all intra-slot scenarios. We implement both the basic and the newly proposed model in the model checking tool Symbolic Analysis Laboratory (SAL) [1]. We then compare the bounds obtained by formally verifying both implementations to an analytic solution and a SystemC simulation. These experiments show that only the proposed model leads to conservative bounds without requiring increased time granularity and therefore helps to avoid state space explosion.

Overview of the paper. The remainder of this paper is organized as follows. Related approaches to delay analysis are discussed in Section II. We then give the problem description in Section III and the basics of the ECA framework in Section IV. This serves as

978-1-4799-2816-3/14/$31.00 ©2014 IEEE

background information for the introduction of a straightforward FPNS model in Section V. After we analyzed its shortcomings, we then extend the ECA's semantics and propose advanced FPNS models in Section VI. Finally, we demonstrate their value with a case study in Section VII. Section VIII concludes.

II. Related Work

Commonly studied schedulability analysis techniques such as those based on demand bound or response time analysis, e.g., [2] for CAN, become pessimistic for the non-preemptive case. Furthermore, they are not compositional in nature. This has led to more general compositional models and techniques such as the Real-Time Calculus (RTC) framework [3], [4] – a purely analytic technique built upon the interaction of upper and lower arrival curves with corresponding service curves that contain information of the system from an interval-based, sliding window perspective. Often, RTC delivers precise results, but it cannot accurately capture *state* information. Efforts in this direction have been the analysis of correlated streams in [5] and lightweight state mechanics in [6]. A FPNS model has been proposed for RTC in [7], but it cannot directly interact with state-based subsystems.

The need for state-based and non-preemptive scheduling has motivated automata-based methods, e.g., [8], that can be used to obtain timing guarantees by delegating to powerful, established model checking tools originating from the digital design area. These tools find an arrival pattern that violates a certain bound if it cannot be guaranteed. Their results are therefore tight by design within the precision of the model. However, they all suffer from state space explosion once the architecture under consideration becomes too large.

Approaches to FPNS analysis using TAs from [9] have been presented in [10] and [11]. Because these approaches employ a continuous time model, they track the evolution of the system in a much more fine-grained fashion than necessary in most cases. In addition, they are difficult to interface with analytical methods, such that scalability often becomes an issue. For instance, the technique from [11] is not applicable when arrival times are given in intervals.

Consequently, previous efforts have supplemented the RTC framework with a model checking based extension, the ECA [12]. It strikes a balance between the high precision tracking of the TA and the scalability of RTC. Large parts of the hardware architecture under consideration can be modeled in plain RTC and then composed with an ECA-modeled part [13]. This considerably improves scalability. Modeling state-based scheduling becomes straightforward as with all automata-based frameworks and they have shown to be a good model for capturing the timing properties of streaming applications. At the same time, they can test and verify richer interfaces [14], as they might arise from feedback control systems [15].

However, ECA models of FPNS have not been examined yet. In order to enable FPNS modeling with ECA, we capture behavior arising from the slotted-time model by taking all possible scenarios into account and delegating the decision to the model checker. Hence, our proposed model does neither require an overly fine time scale nor complicated message tracking mechanics.

III. Problem Description

We consider a distributed control application as shown in Fig. 1 that is communicating over a shared FPNS bus. It is operating with a period T, that acts as a deadline for the *sensor-to-actuator delay*. This kind of feedback control loops typically fulfill safety-critical tasks in physical systems such as cars or airplanes. Since *the physical world does not wait* for the control system to finish processing, the consequences of a missed deadline can be severe.

We focus our analysis on the FPNS bus and try to ensure that the message x_k needs to wait a maximum of T_m there, with T_m being sufficiently small to guarantee that p_s and p_a can be executed in time. This analysis integrates well with larger ECA networks and carries over to more flexible requirements, such as m, k-firm deadlines as discussed in [14].

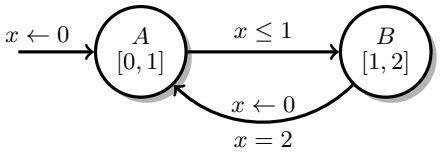

Fig. 2: Example ECA showing the framework's basic functionality.

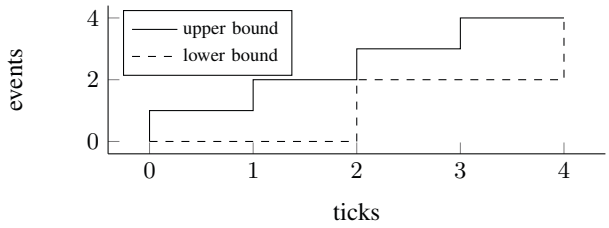

Fig. 3: Upper and lower bounds for the events of the example ECA in Fig. 2.

IV. Event Count Automata

Before we describe the issues surrounding an event-triggered system such as FPNS modeled in discrete time and the proposed FPNS implementation to capture them, we briefly introduce the basic ECA behavior. We illustrate the formal definition using a simple example in Section IV-A, describe how they combine to form larger networks in Section IV-B, and show how we can derive a message's maximum delay from an ECA network in Section IV-C. A complete description of ECAs can be found in [12].

A. Individual ECA

Individual Event Count Automata (ECAs) are given as tuples

$$\mathcal{A} = (S, s_{in}, X, V_{in}, Inv, \rho, \rightarrow). \tag{1}$$

In this formulation:

- S is a set of states and s_{in} is the initial state.
- X is a set of count variables.
- V_{in} is the initial valuation of the count variables.
- $Inv : S \rightarrow \Phi(X)$ is the Invariant Constraint Function that assigns constraints to states with

  $$\Phi(X) = x \leq K | x < K | x \geq K | x > K | \varphi_1 \wedge \varphi_2$$

  where K is a lower/upper bound enforced on a count variable x and $\varphi_l \wedge \varphi_2$ allows combinations of such constraints.
- $\rho : S \rightarrow \mathbb{N} \times \mathbb{N}$ is the rate function.

  $$\rho(s) = [l, u]$$

  Every state is assigned an upper bound u and a lower bound l for the arrival or service rate in that state.
- $\rightarrow \subset S \times \Phi(X) \times 2^X \times S$ is the transition relation. Here $\Phi(X)$ indicates that transitions can have guards of the same form as the invariant constraints of the states. 2^X indicates the set of count variables that will be reset on a transition, typically written as $x \leftarrow 0$.

Consider the simple example depicted in Fig. 2. It has a single count variable $X = \{x\}$. It further consists of two states $S = \{A, B\}$ with associated rate intervals $\rho(A) = [0, 1]$ and $\rho(B) = [1, 2]$. In other words, while in B, one or two events can occur.

The states are connected by transitions with guards such as $x = 1$ and $x \leq 1$ and reset actions $x \leftarrow 0$. The ECA starts in the initial configuration $(s_{in}, V_{in}) = (A, [0])$ where V corresponds to the valuation of the count variable x. Since all transitions are considered urgent by default, we then move to either $(B, [0])$ or $(B, [1])$. In B, the subsequent guard enforces an amount of events

such that $x = 2$ before the automaton resets. This models a jitter in the system.

The transitions in the system deliver the event counts $c \in [C] \stackrel{\text{def}}{=} \{1, \ldots, C\}$ (with C being a finite upper bound that is required to keep the automaton tractable). These automaton-wide event counts increment all the count variables. They represent data arrival or processing during a single step and form the alphabet for the language of the ECA : $[C]$. It contains sequences (or "strings") of event counts $\sigma = (c_1 c_2 c_3 \cdots)$ that represent arrival or processing patterns.

Fig. 3 shows the upper and lower bound on the events that can arrive in this automaton. Note that they are in the time domain, but in this case they could easily be converted to the interval-based interpretation of RTC. In general, ECAs allow more complicated patterns than RTC's service and arrival curves can accurately capture.

B. ECA Networks

Individual ECAs can be connected to form an ECA network which is a structure

$$\mathcal{N} = \left(\{\mathcal{A}_p\}_{p\in\mathcal{P}}, \{U_p\}_{p\in\mathcal{P}}, \mathcal{B}, b, C, IN, OUT\right) \tag{2}$$

In this definition:

- $\mathcal{P}$ is a finite set of nodes that the various ECAs $\mathcal{A}_p$ are associated with.
- U_p is the update function that defines how an ECA consumes and deposits items from its buffers.
- $\mathcal{B}$ is a finite set of buffers of capacity $B_{\max}$.
- C is the maximum number of items that any ECA in the network can handle in a single step. This needs to be finite for the state space underlying the ECA that we need to explore to remain finite.
- $IN : \mathcal{P} \to 2^{\mathcal{B}}$ and $OUT : \mathcal{P} \to 2^{\mathcal{B}}$ link the buffers to the ECA. It is assumed that every $\mathcal{A}_p$ has at least one buffer and that its input and output buffers are disjoint. Furthermore, buffers cannot be shared and merging of streams has to be handled in the automatons.

Consider the following, more intuitive description: Arrival ECAs deposit messages into initial buffers and service ECAs process messages from their input to their output buffers. Each ECA $\mathcal{A}_p$ is characterized by an associated update function U_p that describes from which input to which output it processes data. With N_p, the total amount of buffers attached to $\mathcal{A}_p$, separating into $N_p = N_p^{in} + N_p^{out} \stackrel{\text{def}}{=} |\text{IN}(p)| + |\text{OUT}(p)|$, $U_p : \mathbb{N}^{N_p} \to \mathbb{N}^{N_p}$ maps input and output buffer levels to *changes* in input and output buffer levels.

Data is preserved within individual ECAs. It must therefore hold for all vectors of associated buffer levels $\mathbf{b} = \left[(b_i)_{i\in\text{IN}(p)} \quad (b_i')_{i\in\text{OUT}(p)}\right] \in \mathbb{N}^m$, event counts $c \in [C]$ and states $s \in S$:

$$\sum_{i\in\text{IN}(p)} U_p^{(i)}(s, c, \mathbf{b}) = \sum_{j\in\text{OUT}(p)} U_p^{(j)}(s, c, \mathbf{b})$$

Data can however be overwritten by the buffer mechanics in (3) where b_i is the fill level of the i-th buffer and the $(\cdot)^+$ operator is a temporal next operator[1].

$$b_i^+ = \max\left[0, \min\left(b_i + \sum_{p:i\in\text{OUT}(p)} U_p^{(i)} - \sum_{p:i\in\text{IN}(p)} U_p^{(i)}, B_{\max}\right)\right] \tag{3}$$

[1]More precisely, $b_i = b_i^k$ refers to the content of the buffer at the end of slot k, $b_i^+ = b_i^{k+1}$ refers to the content one time step later. This is a convenient notation because there is no interaction between more distant time steps and it translates well into the implementation language for SAL[1] – a well-known Binary Decision Diagram (BDD)-based model-checking tool that has often been selected for ECA verification.

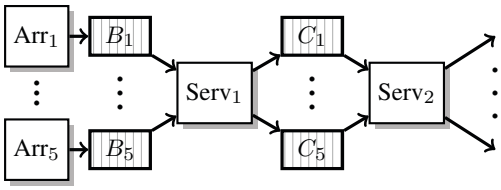

Fig. 4: Example ECA network demonstrating the interconnection using buffers.

Consider the example in Fig. 4. It begins on the left with a set of arrival ECAs. These automata use the trivial update function that deposits all the events into the only attached buffer:

$$U(s, c, \mathbf{b}) = c$$

From the first buffers, the data is further processed by Serv_1 and Serv_2. These service automata could be implementing a Time Division Multiple Access (TDMA) scheduling policy, i.e., they could be ECAs with a state for every TDMA slot and associated update function

$$U^{(i)}(s_i, c, \mathbf{b}) = U^{(i')}(s_i, c, \mathbf{b}) = c$$

for the input buffer b_i and the output buffer b_i' that correspond to the current state s_i and

$$U^{(j)}(s_i, c, \mathbf{b}) = U^{(j')}(s_i, c, \mathbf{b}) = 0$$

otherwise for $j \neq i$.

C. Delay calculation

Once the ECA network is set up, the maximum delay for a certain message stream can be calculated in the following way. Initialize an integer array $D_j = 0$. D_j tracks how many messages have been in the system for j units of time or more. Update D_j to D_j^+ – cf. b_i^+ in equation (3) – as follows[2]:

$$D_j^+ = \begin{cases} \text{inc}(b_1) + D_1 - \text{dec}(b_{\text{last}}) & \text{, if } j = 1 \\ \max\left(0, D_{j-1} - \text{dec}(b_{\text{last}})\right) & \text{, otherwise} \end{cases}$$

The maximum delay is then $\max\{j : D_j > 0 \text{ at any time}\}$ and can be obtained by initializing $d = 0$ and updating d via:

$$d^+ = \begin{cases} d & \text{, if } D_{d+1} = 0 \\ d + 1 & \text{, otherwise} \end{cases}$$

Note that this model does not delete buffer overwrites from the delay count. Any overwrite will therefore cause an unbounded worst case delay for the concerned message stream which is the correct interpretation in most cases.

V. Straightforward FPNS Model

We will now describe a basic FPNS model implemented in the ECA framework. Its mechanics are very close to those of a simulation implementation in, e.g., SystemC. To avoid state space explosion, the time granularity of an ECA model must be chosen much coarser than that of a simulation. Therefore, this intuitive model can not capture all the intricacies of a real-world CAN bus. In this section, we will explore these effects and see how they can lead to overly optimistic bounds in detail. The presented modeling might be used in a typical verification scenario. Relying on its resulting bounds, however, can and will lead to deadline misses and the aforementioned, potentially severe consequences in practice.

[2]$\text{inc}(b_1)$ refers to the increment of the first buffer, i.e., the new arrivals; $\text{dec}(b_{\text{last}})$ to the decrement of the last buffer, i.e., the messages leaving the system. Both are to be regarded with respect to one specific message stream.

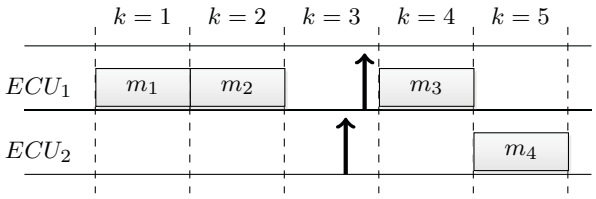

Fig. 5: Perfectly aligned timing as assumed by the straightforward FPNS model ignores blocking by lower-priority message. Table I shows the corresponding ECA trace. The arrows denote the actual message arrival in the send-buffer.

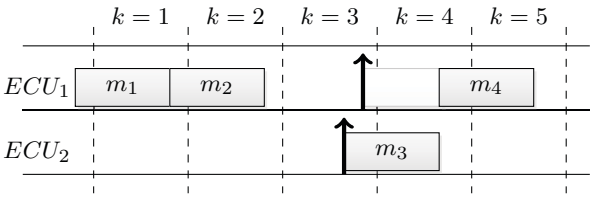

Fig. 6: Timing diagram illustrating how a higher priority-message can be blocked by a lower-priority one in FPNS scheduling. Fig. 5 and Table I show the corresponding sequence in the basic ECA implementation that fails to capture this effect.

A. Implementation Description

The most basic implementation of FPNS, that is closest to how a simulation tool would implement an abstract CAN bus, would employ a constant service automaton (one state A with constant rate $\rho(A) = [c, c]$) and combine it with a priority-based update function where $\pi(b)$ is the priority of a certain message buffer. We obtain $U^{(i)}(s_i, c, \mathbf{b}) = U^{(i')}(s_i, c, \mathbf{b}) = g_i$ with

$$g_i = \max\left(0, \min\left(b_i, c - \sum_{\substack{j \in \mathrm{IN}(p) \\ \pi(b_j) > \pi(b_i)}} b_j\right)\right) \tag{4}$$

This ensures that the Processing Element (PE) handles all the data in the higher priority buffers before moving on to lower priority items. Note that for the highest priority buffer it holds $U = \min(c, b_i)$, meaning it always gets the full service unless it doesn't have enough data in the buffer.

B. Timing Issues of the Straightforward FPNS Implementation

The straightforward FPNS model described in Section V-A assumes that the messages on the FPNS bus are aligned to the ticks of the ECA as illustrated in Fig. 5. The arbitration in the model takes place at the beginning of every tick and it is impossible to resolve timing issues within a slot. More precisely, if two messages arrive in the same slot, it is impossible to say which one was first. The alignment of the ECA's ticks to the real life timing of, e.g., the CAN bus is an overly simplistic assumption that cannot hold in practice, however. When the bus is idle, it does not wait for the next time slot to begin before transmitting once a message is in the buffer ready to be sent. Such a behavior might make verification easier, but it is certainly undesirable when optimizing for throughput.

It can therefore occur in the real CAN bus that higher priority messages are blocked by lower priority messages. This is not captured by the straightforward model and leads to over-optimistic bounds when analyzing it in a model checker. To see this issue in more detail, let us assume for simplicity that there are only two message streams and that they are indexed in order of descending priority (1 being the highest priority). Further let the underlying constant service automaton process one message per time slot, i.e., $c = 1$.

As shown in Fig. 6, after two messages, m_1 and m_2 from the buffer for ECU_1 are sent, all the buffers are empty and the bus idles at the beginning of $k = 3$. Then, a message arrives for ECU_2 (indicated by the black arrow) and the bus starts processing it immediately since ECU_2 is the highest priority accessing it at that moment. When ECU_1 wants to transmit data shortly after (black arrow), the bus is already utilized and ECU_1 is blocked (transparent) until ECU_2 finishes its uninterruptible message m_3. Only then is m_4 transmitted. This blocking is a well-known phenomenon and forms the basis of, e.g., the FPNS analysis for RTC [7]. To the best of our knowledge, it has till now not been addressed in the context of ECA or other discrete-time approaches, however.

Let us compare this to how the straightforward FPNS model would process this arrival pattern. Fig. 5 contains the same arrivals, but the messages are processed according to the straightforward model. The evolution of the two buffers b_1 and b_2 under this model is further detailed in Table I. It starts out just like the real pattern and handles the two items in the buffer of ECU_1. At the end of $k = 2$, the buffers are empty, so nothing is processed in $k = 3$ but there are two arrivals.

The model starts to differ from the observed pattern in step $k = 4$. Here, it would just process the higher priority message because it performs arbitration at the beginning of the slot only. This has been marked in bold in Table I. The bus, however, would have started processing the message of ECU_2 already before the beginning of slot $k = 3$ as described above. This would correspond to $[b_1 \quad b_2] = [1 \quad 0]$ at the end of $k = 4$. Calculating the maximum delay for ECU_1 from this model, e.g., by using the method from Section IV-C, and using it as a bound would therefore be overly optimistic. As we will see in the case study in Section VII, such a bound will be violated by some of the messages.

VI. Proposed FPNS model

Blocking by lower-priority messages as described in Section V-B does occur and needs to be taken care of at the ECA modeling level. We will start by showing how a model could look like for an FPNS element that processes one message per tick (or less, e.g., one message every 3 steps) and then extend it to a multi message model.

A. Single Message Model

In this section, we will introduce a model that takes the possible blocking by lower priority messages into account when a transmission starts before the slot it ends in. Towards this, we need to consider the various paths the intra-slot behavior could take due to the underlying slotted-time mechanics. These paths then need to be included into the update function that governs the arbitration between the individual message streams. The example from Fig. 6 and Table I showed that we cannot just take the highest priority from the buffer and process it. Instead, we have to consider that any newly arrived message with even higher priority could take its place. Fig. 7 illustrates this in greater detail. With one message in the buffer for ECU_3 at the end of $k = 1$ and one message arriving for both ECU_1 and ECU_2 during $k = 2$, the discrete-time semantics allow for the scenarios discussed in the following. Each scenario consists of two arrivals (arrows) and one message (labeled a, b, c) and is further described in the following paragraphs:

k	0	1	2	3	4	5
b_1	2	1	0	1	**0**	0
b_2	0	0	0	1	**1**	0

TABLE I: Buffer fill levels at the end of individual steps k showing the behavior of the basic FPNS model for the arrival pattern in Fig. 5 and Fig. 6.

(a) The message from ECU_1 arrives before the transmission of m_1 ends and it therefore gets to send next. It is irrelevant when the message of ECU_2 arrives in this case.
(b) The message from ECU_2 arrives before m_1 is entirely transmitted and ECU_1 is ready to send only afterwards. ECU_2 therefore sends next.
(c) Both ECU_1 and ECU_2 get ready to send after the transmission of m_1 finishes. ECU_3 therefore transmits next.

Since the slotted time does not hold enough information to resolve which message arrives first within the time slot, all the possibilities need to be accounted for.

To calculate these possibilities, we need to track which messages were already present at the beginning of the slot and which have just arrived during the last step. We therefore extend the basic buffer semantics from Eq. (3) where only the total buffer content was tracked. In addition to b_i, the total elements in the buffer at the end of the slot, we now track n_i, the elements that newly arrived during this slot. This results in the following equations:

$$\begin{aligned} n_i^+ &= \sum_{p:i\in \mathrm{OUT}(p)} U_p^{(i)} \\ b_i^+ &= \max\left(0, b_i + \sum_{p:i\in\mathrm{OUT}(p)} U_p^{(i)} - \sum_{p:i\in\mathrm{IN}(p)} U_p^{(i)}\right) \end{aligned} \tag{5}$$

The elements that remained in the buffer after all the messages from the last step were sent are then given by

$$r_i^+ = b_i^+ - n_i^+. \tag{6}$$

With this additional and more fine-grained information we can now select the message i_0 that the bus starts sending at the end of the last step and finishes in the current step. As shown in Fig. 7, this can be either the message with the highest priority that was already in the buffer and whose transmission therefore would have started immediately after the bus was idle (scenario (c)) or any of the newly arrived messages that have higher priority (scenarios (a) and (b)). This sums up to the following options:

$$i_0 \in \{i : \underbrace{b_i > 0}_{\text{has message to send}} \quad \text{AND} \quad \underbrace{r_j = 0 \quad \forall j > i}_{\text{no higher priority is waiting}} \} \tag{7}$$

With regard to the update function, after setting

$$g_i \overset{\text{def}}{=} \begin{cases} 1 & \text{, if } i = i_0 \\ 0 & \text{, otherwise} \end{cases} \tag{8}$$

we then let $U^{(i)}(s, c, \mathbf{b}) = U^{(i')}(s, c, \mathbf{b}) = g_i$, where i and i' refer to the in- and output buffer of stream i, respectively.

The single message model extends well into a more fine-grained time discretization, where one message is transmitted every couple of time steps. Here, one would just change the underlying service automaton and leave the arbitration as described when there is service to distribute. Additional delay could be modeled by introducing new buffer stages that the message has to travel through.

B. Multi Message Model

As discussed in Section VI-A, extension of the single message model towards more fine-grained timing is straightforward. Establishing a similar model for even less fine-grained timing where multiple messages are to be sent during a single time slot still poses a challenge, however. One possibility for modeling the behavior in this case is to iterate the single message model from Section VI-A.

We start by assuming that i_0 was chosen in the previous time step using the rule from Eq. (7). The next message to be transmitted faces a situation that is similar to the single message case shown in Fig. 7. The difference in the selection of i_1, the ECU to send after i_0, lies in the available messages. Firstly, the messages that were in the buffer at the end of slot $k = 1$ are available if they were not sent already, i.e., minus i_0. Additionally,

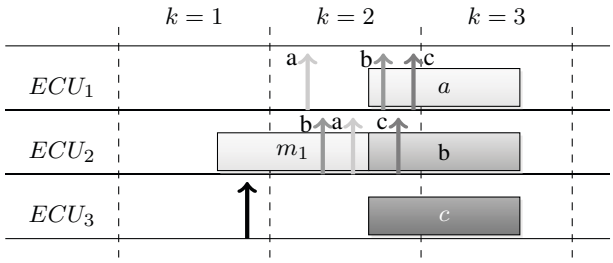

Fig. 7: Three scenarios (a,b,c) with identical arrival signature from the ECA's point of view lead to different outcomes when the CAN's arbitration is not aligned to the ECA's ticks in the single message case.

all the messages with higher priority that might have arrived by then could be ready to send. Since, again, it is impossible to distinguish arrivals within a single time step, this refers to all the messages arriving in the respective slot. Therefore i_1 has to be selected as

$$i_1^+ \in \{i : \underbrace{b_i + n_i^+ - \delta_{ii_0} > 0}_{\text{potentially has message}} \text{ AND } \underbrace{b_j - \delta_{ji_0} = 0 \quad \forall j > i}_{\text{no higher priority}}\}$$

where δ_{ij} refers to the Kronecker delta:

$$\delta ij = \begin{cases} 0 & \text{, if } i \neq j \\ 1 & \text{, if } i = j \end{cases} \tag{9}$$

After i_1 has been determined, further i_l can then be calculated iteratively:

$$\begin{aligned} i_l^+ \in \{i : \quad & b_i + n_i^+ - \sum_{m=0}^{l-1} \delta_{ii_m} > 0 \\ & \text{AND } b_j - \sum_{m=0}^{l-1} \delta_{ii_m} = 0 \quad \forall j > i\} \end{aligned} \tag{10}$$

VII. Case Study

To demonstrate the advantages of the model developed in Section VI-A and extended in VI-B, we performed a case study using Symbolic Analysis Laboratory (SAL) [1] and SystemC. We converted the abstract ECA networks from both the straightforward model and the proposed extended models to their transition system representation. We then implemented them in the modeling language of SAL. Finally, we employed its BDD-based model-checker to verify the respective system behavior. We calculated bounds on the delay for every stream via the method outlined in Section IV-C. These will be referred to as *StraightForw*, *SingleMsg* and *MultiMsg*, respectively. Additionally, we constructed a SystemC model for the CAN bus and obtained the average delay *Avg* and the maximum delay *Max* as observed in a simulation for $T = 10^7$ms ≈ 2.7h. Finally, we performed a manual analysis of the worst case scenario where all messages start at the same time and are initially blocked by the longest lower priority message. This is denoted by *Manual*. It is both tedious and error-prone and unlike ECAs and SystemC, this analysis cannot be applied to state-based or even just larger systems.

We assumed 4 message streams numbered #1 to #4 from highest to lowest priority with periods

$$P = [1\text{ms} \quad 2\text{ms} \quad 4\text{ms} \quad 5\text{ms}]$$

and a common jitter of $j = 0.5$ms where jitter refers to an arrival in the interval $[k \times P_i, k \times P_i + j]$.

The message streams share a CAN bus with transmission time of 0.5ms per message. No state dependence was modeled in order to have readily available reference solutions for comparison.

Runtime Evaluation. The model checking runs for the following case study took 16s for the single message model and 85s for the multi message model on an Intel i7 at 3.4GHz with 16GB RAM. Note that while the multi message model took longer to verify in this case, it might be the preferred or only choice in case another PE dictates a certain slot time.

Single message model. To test the single message model, we set the slot time to 0.5ms in accordance with the transmission time. The resulting bounds created by the formal verification of the ECA models via the model checking tool SAL are listed in Table II as *Straightforw* and *SingleMsg* respectively.

Subsequently, we took the same architecture and modeled it in SystemC. We simulated the model and recorded the delays of the individual messages. Fig. 8 shows the delay distribution of Stream #3. It illustrates that some messages do in fact incur a larger delay than 2.5ms, the bound calculated using the straightforward model. In contrast, no message exceeded 4.5ms, the bound guaranteed by SAL when verifying the newly developed single message model.

Because of their similarity, we do not show histograms for the other message streams. Instead, we have summed up their average and maximum delay in Table II. All the bounds from the single message model shown there prove to be tight within the resolution of the ECA. To clarify, consider stream #3 in Table II as an example. There, we observed a maximum delay of 3.63ms (equivalent to 4ms or 8 slots, rounded up for the ECA) and our manual analysis gave a worst-case delay of 4.0ms that can occur in practice. Verifying the ECA model with SAL leads to an upper bound of 8 slots. This corresponds to the same 4.0ms and would therefore be tight as well. However, we additionally account for the fact that this worst-case message could have arrived early in slot $k = k_0$ and was transmitted late in slot $k = k_0 + 8$, increasing the upper bound *SingleMsg* by an additional slot to 4.5ms.

Multi message model. We used the same case study with the multi message model. Toward this, we changed the slot time to 1ms, such that two messages can be transmitted per slot. By transferring this model to SAL and running its BDD-based model checker again, we obtained the bounds *StraightForw* and *MultiMsg* from Table III. These bounds appear to be very conservative with respect to the simulation results and the analysis from Table II. This is due to the higher implied jitter in the multi message case. Since the slot time is 1ms, the model now assumes that all the messages suffer from a jitter of 1ms as well. Repeating the simulation and the analysis with this jitter in mind, we obtain the columns *Avg*, *Max* and *Manual* in Table III. Again, we observe that the straightforward model is overly optimistic. In contrast, the proposed model successfully bounds the transmission delay and it is tight within the selected resolution of the ECA framework when compared to the analytical worst case. We could, however, not observe the worst case for all message streams in the SystemC simulation.

VIII. Concluding remarks

This paper points out and thoroughly analyzes the challenges inherent in modeling FPNS on top of a slotted-time model. A basic implementation does not account for blocking by lower priority messages and can thus lead to bounds that do not hold in practice. Instead of simply refining the time scale to capture these effects – a futile approach due to state space explosion – we propose a FPNS model that can be created with reasonable modifications to the semantics of the well-established ECA framework. We compare the bounds obtained by formally verifying a straightforward model as well as our proposed model with results from a SystemC simulation. This case study demonstrates the importance of the improved modeling, as the bounds from the straightforward approach are in fact violated by a significant amount of messages.

With the proposed extensions to the ECA semantics and the developed FPNS model, an accurate and efficient verification of timing bounds becomes possible.

	Delay from SystemC		Analytic Delay Bounds		
Stream #	*Avg*	*Max*	*Manual*	*Straightforw*	*SingleMsg*
1	0.66ms	0.98ms	1.0ms	1.0ms	1.5ms
2	0.83ms	1.83ms	2.0ms	1.5ms	2.5ms
3	1.10ms	3.63ms	4.0ms	2.5ms	4.5ms
4	1.17ms	3.65ms	4.0ms	4.5ms	4.5ms

TABLE II: Single message model case study results.

	Delay from SystemC		Analytic Delay Bounds		
Stream #	*Avg*	*Max*	*Manual*	*Straightforw*	*MultiMsg*
1	0.65ms	1.4ms	1.5ms	2ms	2ms
2	0.78ms	2.3ms	2.5ms	2ms	4ms
3	1.02ms	4.5ms	5.5ms	3ms	7ms
4	1.14ms	5.45ms	7.5ms	5ms	9ms

TABLE III: Multi message model case study results.

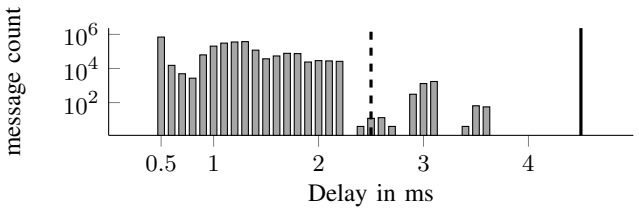

Fig. 8: Histogram of delays experienced by message stream #3 as measured in the SystemC simulation. The bound from the straightforward model (dashed, Section V) is exceeded by a significant part of the messages while the bound from the single message model (solid, Section VI-A) holds.

References

[1] L. de Moura, S. Owre, H. Rue, J. Rushby, N. Shankar, M. Sorea, and A. Tiwari, "SAL 2," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2004, vol. 3114, pp. 251–254.

[2] R. I. Davis, A. Burns, R. J. Bril, and J. J. Lukkien, "Controller Area Network (CAN) Schedulability Analysis: Refuted, Revisited and Revised," *Real-Time Systems*, vol. 35, no. 3, p. 239272, 2007.

[3] L. Thiele, S. Chakraborty, and M. Naedele, "Real-time Calculus for Scheduling Hard Real-Time Systems," in *ISCAS*, 2000.

[4] E. Wandeler and L. Thiele, "Real-Time Calculus (RTC) Toolbox," `http://www.mpa.ethz.ch/Rtctoolbox`, 2006.

[5] K. Huang, L. Thiele, T. Stefanov, and E. Deprettere, "Performance Analysis of Multimedia Applications Using Correlated Streams," in *DATE*, 2007.

[6] A. Bouillard, L. Phan, and S. Chakraborty, "Lightweight Modeling of Complex State Dependencies in Stream Processing Systems," in *RTAS*, 2009.

[7] D. B. Chokshi and P. Bhaduri, "Modeling Fixed Priority Non-Preemptive Scheduling with Real-Time Calculus," in *RTCSA*, 2008.

[8] K. G. Larsen, P. Pettersson, and W. Yi, "Compositional and Symbolic Model-Checking of Real-Time Systems," in *RTSS*, 1995.

[9] R. Alur and D. L. Dill, "A Theory of Timed Automata," *Theoretical computer science*, vol. 126, no. 2, p. 183235, 1994.

[10] J. Krakora and Z. Hanzalek, "Timed Automata Approach to CAN Verification," in *INCOM*, 2005.

[11] E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi, "Schedulability Analysis of Fixed-Priority Systems Using Timed Automata," *Theoretical Computer Science*, vol. 354, no. 2, p. 301317, 2006.

[12] S. Chakraborty, L. Phan, and P. Thiagarajan, "Event Count Automata: a State-Based Model for Stream Processing Systems," in *RTSS*, 2005.

[13] L. Phan, S. Chakraborty, P. Thiagarajan, and L. Thiele, "Composing Functional and State-Based Performance Models for Analyzing Heterogeneous Real-Time Systems," in *RTSS*, 2007.

[14] M. Kauer, S. Steinhorst, D. Goswami, R. Schneider, M. Lukasiewycz, and S. Chakraborty, "Formal Verification of Distributed Controllers using Time-Stamped Event Count Automata," in *ASP-DAC*, 2013.

[15] G. Weiss and R. Alur, "Automata Based Interfaces for Control and Scheduling," *HSCC*, 2007.