Performance Analysis of FlexRay-based ECU Networks

Andrei Hagiescu[1] Unmesh D. Bordoloi[1] Samarjit Chakraborty[1]
Prahladavaradan Sampath[2] P. Vignesh V. Ganesan[2] S. Ramesh[2]
[1]Department of Computer Science, National University of Singapore
[2]General Motors R&D - India Science Laboratory, Bangalore
{hagiescu, unmeshdu, samarjit}@comp.nus.edu.sg, {p.sampath, prasannavignesh.ganesan, ramesh.s}@gm.com

ABSTRACT

It is now widely believed that FlexRay will emerge as the predominant protocol for in-vehicle automotive communication systems. As a result, there has been a lot of recent interest in timing and predictability analysis techniques that are specifically targeted towards FlexRay. In this paper we propose a compositional performance analysis framework for a network of electronic control units (ECUs) that communicate via a FlexRay bus. Given a specification of the tasks running on the different ECUs, the scheduling policy used at each ECU, and a specification of the FlexRay bus (e.g. slot sizes and message priorities), our framework can answer questions related to the maximum end-to-end delay experienced by any message, the amount of buffer required at each communication controller and the utilization of the different ECUs and the bus. In contrast to previous timing analysis techniques which analyze the FlexRay bus in isolation, our framework is fully *compositional* and allows the modeling of the schedulers at the ECUs and the FlexRay protocol in a seamless manner. As a result, it can be used to analyze large systems and does not involve any computationally expensive step like solving an ILP (which previous approaches require). We illustrate our framework using detailed examples and also present results from a Matlab-based implementation.

Categories and Subject Descriptors

C.3 [**Special-purpose and application-based systems**]: Real-time and embedded systems; C.4 [**Performance of systems**]: Design studies and modeling techniques

General Terms

Performance, Design

Keywords

Automotive electronics, Bus protocols, FlexRay

1. INTRODUCTION

Since the last two decades there has been a phenomenal increase in the use of electronic components in automotive systems, resulting in the replacement of purely mechanical or hydraulic-implementations of many functionalities. The main motivation behind this stems from lower cost, reduced weight, new and innovative functionalities and the need for faster design cycles. In earlier designs, different functions were implemented as stand-alone electronic control units (ECUs), with each ECU consisting of one or more microcontrollers and a set of sensors and actuators. However, with the rapid increase in the complexity of the different functionalities, it became imperative to have distributed implementations, where different parts of a task are implemented on different ECUs with messages and signals being exchanged between them. For example, an ECU implementing *crash preparation* needs inputs from *wheel rotation sensors*, *radars*, and ECUs implementing tasks such as *object detection*, *data fusion* and *object selection*. Today, in high-end cars, it is common to have around 70 ECUs exchanging upto 2500 signals between them [1]. Hence, it is infeasible to connect the different ECUs with point-to-point links. This has led to the development of bus-based ECU networks, where communications between multiple ECUs are multiplexed over one or more shared buses. Consequently, this also gave rise to the need for different communication protocols specifically targeting automotive communication systems.

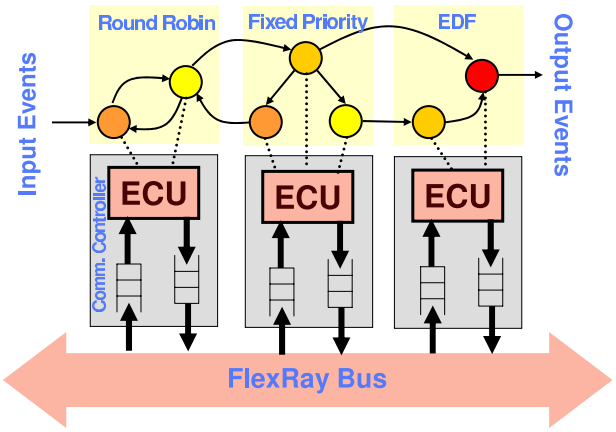

Figure 1: A FlexRay-based network of ECUs, with an application partitioned and mapped onto multiple ECUs.

Today, the most commonly used protocols [12] include the Controller Area Network (CAN) [2], the Local Interconnection Network (LIN) [10] and the J1850 from the Society for Automotive Engineers (SAE) [9, 11]. The different protocols can be classified into two major groups: (i) time-triggered, and (ii) event-triggered. Communication activities in the latter class are triggered by the occurrence of specific events and the protocol defines a policy for resolving the contention for the shared bus when messages from multiple ECUs or tasks are ready at the same time. For example, in the case of CAN, data is segmented into *frames* and each frame is labeled with a priority which is used to resolve bus contention. Time-triggered protocols, on the other hand, schedule communication activities or frame transfers at predetermined points in time, which are commonly referred to as *slots*. The sequence of slots and their lengths for different message types are statically defined and the resulting schedule repeats itself infinitely.

DAC 2007, June 4–8, 2007, San Diego, California, USA.

Event-triggered protocols are clearly more efficient in terms of communication bandwidth usage and allow incremental system design (i.e. new ECUs or tasks can be added without redesigning the system from scratch). However, they are difficult to analyze because of their dynamic nature. Hence, verifying timing properties and detecting faults often become problematic. This poses a serious hindrance to their deployment when the functions involved are safety-critical and require hard real-time guarantees. On the other hand, time-triggered protocols are highly predictable in terms of their temporal behaviour, but suffer from poor communication bandwidth utilization and are inflexible. The addition of new ECUs, or the modification of any tasks require a complete redesign and reevaluation of the entire system.

As a result, recently there has been a lot of emphasis on hybrid protocols, that combine the time-triggered and event-triggered paradigms. Protocols in this class include TTCAN [18], FTT-CAN [7] and FlexRay [8]. FlexRay is currently backed by many major automotive companies and will most likely become the de-facto standard for automotive communication systems very soon. This has led to a lot of recent interest in timing and predictability analysis techniques and tool-support targeting FlexRay-based designs.

Our contributions and related work: This paper is in line with these efforts and proposes an analytical framework for compositional performance analysis of a network of ECUs that communicate via a FlexRay bus. Given a specification of the applications running on the system, their partitioning and mapping on the different ECUs, their activation rates and the mapping of the resulting messages onto the different FlexRay slots along with the message priorities (see Figure 1), our framework can be used to answer various performance analysis-related questions. These include the maximum end-to-end delay experienced by the different message types, the amount of buffer space required within a communication controller associated with an ECU and the utilizations of the different ECUs and the FlexRay bus. Our framework can also be used for deriving the parameters of the FlexRay protocol (e.g. lengths of the static and dynamic segments and priorities of the messages mapped onto the dynamic segment). Further, it can help in resource dimensioning (e.g. designing the various ECUs) and determining optimal scheduling policies for multitasking ECUs.

In the FlexRay protocol, a communication cycle consists of a combination of a time-triggered or static (ST) segment and an event-triggered or dynamic (DYN) segment. Such a communication cycle is repeated in a periodic fashion. The ST segment uses a time-division multiple access (TDMA) scheme and the DYN segment uses—what is often referred to as—*Flexible TDMA*. The ST segment has all the virtues of a time-triggered paradigm, i.e. the timing properties of messages mapped onto this segment are highly predictable. But it is mostly suited for periodic messages and has low communication bandwidth utilization. The DYN segment compensates this drawback, but suffers from the usual shortcomings of an event-triggered paradigm. As a result, most of the current implementations of FlexRay heavily lean towards using only the ST segment, with the DYN segment being unutilized. The only advantage of FlexRay that is being exploited in this process is its high bandwidth. To fully utilize the benefits of this protocol, it is important that suitable analysis techniques be developed that can provide timing and performance guarantees for messages mapped onto the DYN segment as well. This is complicated because of two reasons: (i) the DYN part of the protocol is more complex than the ST part, and (ii) the potential messages targeted for the DYN segment tend to be more irregular (e.g. high-volume multimedia data) than those mapped onto the ST segment (the DYN segment has been specifically designed for such messages).

Commercially available design tools for FlexRay-based systems (e.g. those from dSPACE [6] and DECOMSYS [5]) today mostly rely on simulation. As a result, they are time consuming to use and cannot provide formal performance guarantees, which are important in the automotive domain. Although formal timing analysis techniques have been proposed for protocols such as CAN [15, 17] and TTP [13], none of them seem to extend in a straightforward manner to model the DYN segment FlexRay.

Very recently, the first attempt to formally model the behaviour of the DYN segment was reported in [14]. Given the arrival rates of the different message streams mapped onto the DYN segment, [14] computes the worst-case delay experienced by any message due to blocking by the ST segment and contention from higher priority messages. Computing this worst-case delay was shown to be similar to a bin covering problem [4] and was solved using an integer linear programming (ILP) formulation. Further, computationally efficient (but pessimistic) heuristics were also presented to bound this delay. Although, this certainly represents an important step towards formally analyzing the FlexRay protocol, it suffers from certain drawbacks which might hamper its application to real-life problems. The first, and most important of these being that [14] analyzes the FlexRay bus in isolation, i.e. requires the input rates or periods of the arriving messages and computes the worst-case delay due to transmission over the bus. A system designer, on the other hand is typically interested in computing the worst-case end-to-end delays of messages originating from a sensor, passing over multiple ECUs and the FlexRay bus, and finally activating an actuator (see Figure 1 for an illustration). In this process, a message stream arriving at the FlexRay bus need not be purely periodic and might get modified depending on the scheduling policies on the different ECUs.

The framework we present in this paper addresses this concern. It is fully compositional and models both the ECUs and the FlexRay bus in a seamless manner. Hence, it does not make any a priori assumption on the timing properties of the message streams arriving at the bus. Further, in contrast to [14]—which is only restricted to computing the worst-case response times of messages—our framework can be used to answer a wider variety of performance-related questions and will also be helpful for synthesizing a FlexRay schedule (i.e. determine the slot sizes and message priorities) when maximum end-to-end delays are provided as design constraints. Lastly, our approach does not involve any computationally expensive step like solving an ILP and would hence scale to real-life settings. We have implemented our framework using a combination of Java and Matlab, which can be used as a stand-alone design tool, or can serve as a plugin to standard tool suites (e.g. DECOMSYS Tools [5]). Such a plugin can be used to obtain hard performance guarantees, which can then be cross-validated using simulation.

Organization of the paper: The rest of this paper is organized as follows. In the next section we briefly discuss the FlexRay protocol. In Section 3 we give an overview of our basic framework and the challenges in modeling the DYN segment of FlexRay. This is followed by a formal performance model for FlexRay, which is the main result of this paper. A case study is presented in Section 5, followed by possible directions for future work in Section 6.

2. OVERVIEW OF FLEXRAY

As mentioned in the previous section, each FlexRay communication cycle is partitioned into a ST and a DYN segment. The lengths of these segments need not be equal, but are fixed over the different cycles (hence these lengths are among the parameters that need to be determined when the FlexRay schedule is synthesized). The ST

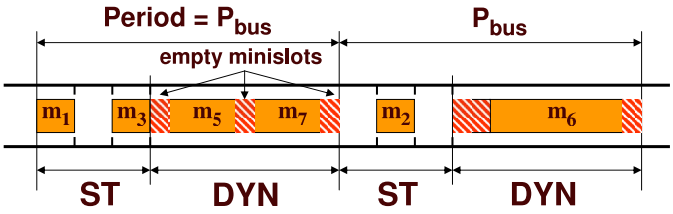

Figure 2: Two typical FlexRay communication cycles.

segment is further partitioned into a fixed number of equal-length slots. Each slot is allocated to a specific task and a task is allowed to send a message only during its allocated slot. If a task has no messages to send, then its slot goes empty (i.e. other tasks are not allowed to use it).

The DYN segment is also partitioned into equal-length slots, but each slot size is much smaller and is referred to as a *minislot*. Tasks which send messages on the DYN segment are assigned fixed priorities. At the beginning of each DYN segment, the highest priority task is allowed to send a message. The length of such a message can be arbitrarily long (i.e. can occupy an arbitrary number of minislots), but has to fit within one DYN segment. However, if the task has no message to send, then only one minislot goes empty. In either case, the bus is then given to the next highest-priority task and the same process is repeated till the end of the DYN segment. Further, when its turn comes, a task is only allowed to send a message if it fits into the remaining portion of the DYN segment. For further details of this protocol, we refer the reader to the excellent description in [14] or to the full specification [8].

As an example, consider eight tasks $T_1, \ldots, T_8$ mapped onto different ECUs, which send messages on the FlexRay bus. Any message sent by a task T_i is labeled as m_i. Tasks T_1, T_2 and T_3 send messages over the ST segment and T_4 to T_8 over the DYN segment. For the DYN segment, the priorities of the tasks decrease from T_4 to T_8. Figure 2 shows two consecutive FlexRay communication cycles resulting from this mapping. In the first cycle, task T_2 has no message to send (hence the corresponding slot in the ST segment is empty) and in the second cycle T_1 and T_3 have nothing to send.

Similarly, in the first cycle, tasks T_5, T_6 and T_7 have messages to send, but not T_4 and T_8. Hence, there is one empty minislot corresponding to T_4 in the DYN segment, followed by the message m_5. The size of m_6 is bigger than the remaining length of the DYN segment, hence it is not sent; instead there is one empty minislot in its place. This is followed by m_7 and another empty minislot resulting out of no message from T_8. In the second cycle, T_4 and T_5 have no messages to send, which results in two empty minislots. These are followed by m_6 which could not be sent in the first cycle. The DYN segment ends with one empty minislot which might either be because T_7 had nothing to send or its message was longer than one minislot.

It may be noted that (i) the ST and DYN segments are independent of each other, and (ii) techniques for analyzing the timing behaviour of the ST segment are already known (because it uses a TDMA scheme) [13, 16]. Hence, from now on we will only focus on modeling the behaviour of the DYN segment (however, we will of course take into account the blocking effects of the ST segment).

3. BASIC FRAMEWORK

In this section we give an overview of our basic modeling framework and the challenges faced in modeling the DYN segment of FlexRay. In the next section we show how these challenges are addressed. Our modeling techniques are motivated by [3], where a mathematical framework was presented for analyzing the timing properties of multiprocessor embedded systems. Our main contribution in this paper lies in appropriately modifying this framework to model the FlexRay protocol, which turns out to be a non-trivial task, as we show in this section.

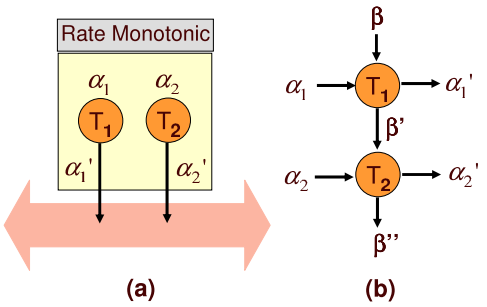

Figure 3: (a) Rate monotonic scheduling of two tasks. (b) Corresponding scheduling network.

The system architectures we are interested in consist of multiple ECUs communicating via a FlexRay bus. One or more applications are partitioned into tasks, which are then mapped onto different ECUs. ECUs running multiple tasks use a scheduler to share the available processing resources as shown in Figure 1. Each task is activated at a certain rate or is triggered by an output from another task. Once activated, it needs to be processed and hence consumes a fixed number of processor cycles from the ECU on which it is running.

The activation rate of any task is upper- and lower-bounded by two functions $\alpha^u(\Delta)$ and $\alpha^l(\Delta)$. Let $R(t)$ be the total number of times a task is activated during the time interval $[0, t]$. Then $\alpha^l(\Delta) = \min_{t \geq 0}\{R(t+\Delta) - R(t)\}$ for any Δ. Similarly, $\alpha^u(\Delta) = \max_{t \geq 0}\{R(t+\Delta) - R(t)\}$. Hence, $\alpha^u(\Delta)$ and $\alpha^l(\Delta)$ denote the maximum and minimum number of times the task can be activated within *any* interval of length Δ. Similarly, let $\beta^u(\Delta)$ and $\beta^l(\Delta)$ be upper and lower bounds on the *service* available to this task. Let $S(t)$ be the number of activations of this task that were serviced during the time interval $[0, t]$. Then, $\beta^l(\Delta) = \min_{t \geq 0}\{S(t+\Delta) - S(t)\}$ for any Δ, and $\beta^u(\Delta) = \max_{t \geq 0}\{S(t+\Delta) - S(t)\}$. If there are multiple tasks running on an ECU, the service bounds β^u and β^l available to any task will clearly depend on the scheduling policy used by the ECU. Further, if $\beta^u(\Delta)$ and $\beta^l(\Delta)$ are expressed in terms of the maximum and minimum number of *processor cycles* available within any time interval of length Δ, then they can be easily converted to represent the service in terms of the number of task activations that can be serviced, by scaling them with the execution requirement of an activation.

Now, let each serviced activation of a task generate a message and $\alpha^{u\prime}(\Delta)$ and $\alpha^{l\prime}(\Delta)$ denote upper and lower bounds on the number of such messages generated within *any* time interval of length Δ. Such messages can activate other tasks on the same ECU, or might be transferred over the FlexRay bus to trigger tasks running on other ECUs. It can be shown that:

$$\alpha^{l\prime}(\Delta) = \min\{\inf_{0 \leq \mu \leq \Delta}\{\sup_{\lambda > 0}\{\alpha^l(\mu+\lambda) - \beta^u(\lambda)\} + \beta^l(\Delta - \mu)\}, \beta^l(\Delta)\}$$

$$\alpha^{u\prime}(\Delta) = \min\{\sup_{\lambda > 0}\{\inf_{0 \leq \mu < \lambda + \Delta}\{\alpha^u(\mu) + \beta^u(\lambda + \Delta - \mu)\} - \beta^l(\lambda)\}, \beta^u(\Delta)\}$$

Similarly, the bounds on the *remaining service* after processing the activations of this task is given by:

$$\begin{aligned} \beta^{l\prime}(\Delta) &= \sup_{0 \leq \lambda \leq \Delta}\{\beta^l(\lambda) - \alpha^u(\lambda)\} \\ \beta^{u\prime}(\Delta) &= \max\{\inf_{\lambda > \Delta}\{\beta^u(\lambda) - \alpha^l(\lambda)\}, 0\} \end{aligned}$$

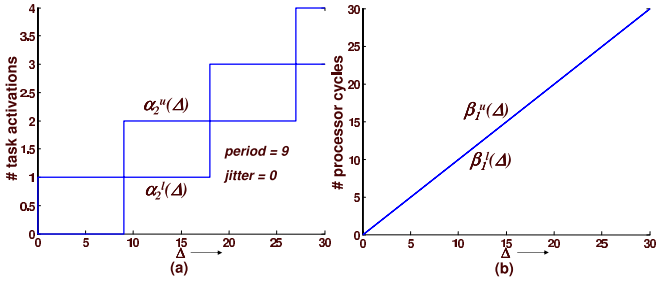

Figure 4: (a) α^u and α^l corresponding to a periodic activation. (b) β^u and β^l of an unloaded processor.

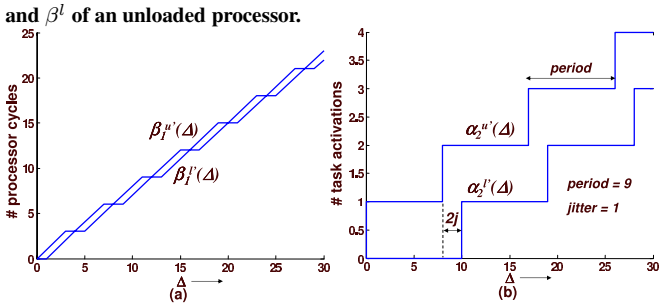

Figure 5: (a) Upper and lower bounds on the *remaining* service after processing task T_1. (b) Bounds on the messages generated by T_2.

To see the utility of these bounds, consider the setup shown in Figure 3(a). It shows two tasks T_1 and T_2, which are being scheduled using a rate monotonic scheduler. Both T_1 and T_2 are activated periodically, with T_1's period being 4 time units and T_2's period being 9 time units. Each activation of T_1 and T_2 requires 1 and 2 processor cycles respectively to process. The upper and lower bounds on the activation of T_2 (i.e. α_2^u and α_2^l) are shown in Figure 4(a). They are similar for T_1, except for the difference in the length of the period. The upper and lower bounds on the service offered by the unloaded ECU (in terms of the number of processor cycles available over any time interval) is shown in Figure 4(b) (note that these bounds coincide for obvious reasons). Since T_1 has a smaller activation period, it has a higher priority (because of rate monotonic scheduling) and hence the full service offered by the unloaded ECU is made available to it.

As discussed above, using α_1^u, α_1^l and β_1^u, β_1^l (the service bounds for the unloaded processor), we can compute $\beta_1^{u\prime}$ and $\beta_1^{l\prime}$, which are bounds on the *remaining* service (that is left over after processing T_1). This remaining service is now available to the lower-priority task (i.e. T_2). This concept is illustrated in the form of a *scheduling network* for rate monotonic (or any fixed priority) scheduler in Figure 3(b).

$\beta_1{}'$ is used for servicing task T_2 (see Figure 5(a)), which along with α_2 can be used to compute upper and lower bounds on the messages generated by each serviced activation of T_2 (β and α often refer to the tuples β^u, β^l and α^u, α^l). These bounds are shown in Figure 5(b). From this figure, note that this message stream is periodic with a period of 9 time units and a jitter of 1 time unit. It is straightforward to see that the distance between $\alpha_2^{u\prime}$, $\alpha_2^{l\prime}$ is equal to twice the jitter of the message stream.

α_1' and α_2' can now be used for computing the load on the bus and the same technique can be applied to compute the timing properties of the transmitted messages, which can then trigger tasks on other ECUs. Further, given α^u, α^l and β^u, β^l, it is possible to compute the maximum *delay* experienced by a task before its activation is serviced and the maximum number of *backlogged* activations. These are given by: delay $\leq \sup_{t\geq 0}\{\inf_{\tau\geq 0}\{\alpha^u(t) \leq \beta^l(t+\tau)\}\}$

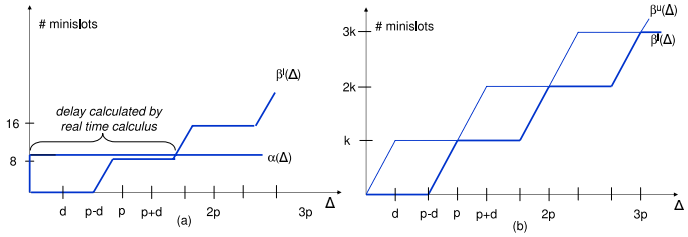

Figure 6: (a) Computing maximum delay from α^u and β^l. (b) Total service offered by the DYN segment.

and backlog $\leq \sup_{t\geq 0}\{\alpha^u(t) - \beta^l(t)\}$.

It should be clear by now that although we have used this framework to analyze a processing element (i.e. tasks executing on an ECU), the same technique is also applicable to communication resources (e.g. buses). For scheduling policies other than fixed priority, the scheduling network would be different from the one shown in Figure 3(b). For example, for TDMA, a fraction of β would be available to T_1 and and another fraction to T_2. These fractions would depend on the TDMA slot sizes, but not on the functions α_1 and α_2.

3.1 Difficulties in Modeling FlexRay

Recall from Section 2 that in the DYN segment of FlexRay, tasks are given access to the bus in decreasing order of their priorities. In other words, the task with the highest priority is offered access to the bus at the start of the DYN segment. Further, once given access to the bus, a task can occupy it till the end of the current DYN segment. Hence, the most straightforward approach would be to model this protocol as a fixed priority scheduler, as shown in Figure 3(b). Here, β would be used to model the total service offered by the DYN segment and successive β's would be computed from the message sizes and message generation rates of the different tasks. However, this approach does not work because of the following properties of FlexRay: (i) A task is only allowed to send a message if it fits into the remaining portion of the DYN segment, i.e. a message cannot straddle two communication cycles. (ii) Once a task misses its turn in the DYN segment (because there were no ready messages at the beginning of its communication slot), it has to wait till the next communication cycle before it can access the bus (which is the TDMA-like property of the DYN segment). (iii) A task can send at most one message in each DYN segment (where the maximum length of the message can be equal to the length of the DYN segment). (iv) One minislot is consumed from the available service each time a task is not ready to transfer a message, before the next task is allowed to send its message on the bus.

The modeling framework presented above does not incorporate these restrictions when representing the service availability of a resource using the upper and lower bounds $\beta^u(\Delta)$ and $\beta^l(\Delta)$. To see this, consider Figure 6(a), which shows α^u corresponding to the arrival of a single message (of length equal to 10 minislots) that is to be transmitted over the DYN segment (of length 8 minislots). Here, the length of each communication cycle (or period) is assumed to be p time units and the length of the DYN segment is equal to d time units. The lower bound on the service β^l corresponding to the DYN segment is also shown in this figure. Note that over time intervals Δ of length less than or equal to $p - d$, no service might be available from the DYN segment due to the blocking by the ST segment.

Since the length of the message in this case is longer than the length of the DYN segment, this message will never get transmitted. However, the framework we described above models the mes-

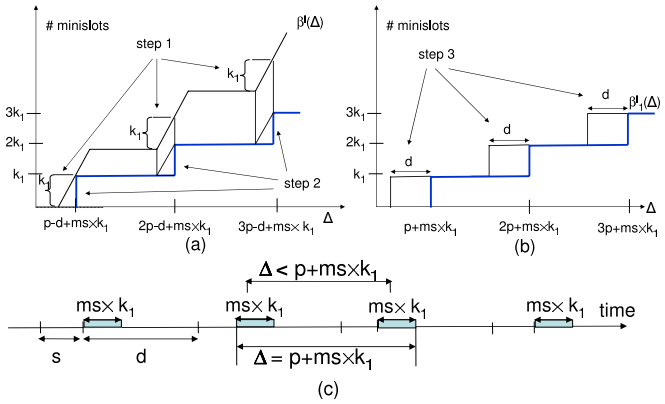

Figure 7: (a) Steps 1 and 2 for transforming β^l. (b) Shifting the resulting service bound. (c) Blocking time.

sage to be transmitted over two communication cycles, thereby incurring a delay equal to the maximum horizontal distance between α^u and β^l (see Figure 6(a)).

4. MODELING FLEXRAY

FlexRay – as described in Section 3.1 – restricts the amount of available service that can actually be used. Hence, while the service bounds $\beta^u(\Delta)$ and $\beta^l(\Delta)$ capture the limits on the total service available to the DYN segment, we need to model how much of this service can actually be used.

Towards this, assume that tasks $T_1, \ldots, T_n$ send messages over the DYN segment with any message from task T_i being denoted by m_i and has a length of k_i minislots. The length of the DYN segment is assumed to be equal to k minislots (or d time units) and the length of a communication cycle, as before, is equal to p time units. Each minislot is assumed to be MS time units long.

Let $\beta^l(\Delta)$ be the lower bound on the service (expressed in terms of number of minislots) offered by the unloaded DYN segment to all the tasks. Further, let β_i^l be the service offered by the DYN segment to task T_i.

It can be shown that each "increasing" segment of the service curve corresponds to an additional communication slot with a minimum length being the increase in the service value. Therefore, the transformations applied to each slope of the service curve capture the worst-case increase in the availability of transmission time for T_i corresponding to the additional available slot contained in the extension of the interval.

To obtain β_1^l, the function β^l needs to be algorithmically transformed. The service curve is trimmed to fit the message size requirments. Afterwards, it is delayed to ensure the full availability of the slot in the specified time interval. The following steps are used:

1. Extract k_1 minislots of service during each communication cycle from β^l. This is because during any communication cycle at most k_1 minislots are available to T_1 (since a task can send at most one message). Nullify the communication cycles containing less then k_1 minislots.

2. Discretize the service bound obtained from Step 1, i.e. convert it into a step-function. This is to model that a message cannot straddle two communication cycles. Steps 1 and 2 are shown in Figure 7(a).

3. The resulting service bound is shifted by d time units. This is again to model that a message has to be completely sent within a single DYN segment, and must start at the beginning of the communication slot. Note from Figure 7(c) that any interval Δ of length less than $p + \texttt{MS} \times k_1$ can be positioned to straddle two communication cycles. Hence, the minimum service available from the DYN segment over intervals of such length is equal to 0. The shifted service bound in Figure 7(b) reflects this. It also reflects the property that once a task misses its turn in the DYN segment, it has to wait for the next communication cycle.

4. A minislot is lost even when a task does not transmit any message. This is accounted by subtracting one minislot from each communication cycle corroborated with an adjusted message size of $k_1 - 1$ minislots in the subsequent analysis of service consumption for messages transmited by task T_1.

The resulting service bound, which we denote as β_1^l correctly represents the minimum or guaranteed service from the DYN segment that is available to messages from T_1. This β_1^l can now be plugged into the framework outlined in Section 3 to compute the maximum delay suffered by any m_1, the maximum number of backlogged m_1s and the timing properties of the transmitted messages (which might trigger other tasks). Towards this $\alpha_1^u(\Delta)$ is used as an upper bound on the number of messages generated by T_1 within any interval of length Δ.

The service available to the lower priority tasks (i.e. $T_2, \ldots, T_n$) is made up of two components: (i) The remaining service left after performing transformation 1 (i.e. the service that was *unavailable* to T_1). This is given by $\bar{\beta}^l(\Delta) = \sup_{0 \le \lambda \le \Delta}\{\beta^l(\lambda) - \beta_1^l(\lambda)\}$. (ii) The service that was *unutilized* by T_1. This can be computed from β_1^l and α_1^u and is denoted by $\beta_1^{l\prime}$. However, $\beta_1^{l\prime}$ cannot be directly added to $\bar{\beta}^l$ because it is specific to messages from task T_1 (i.e. incorporates message size dependent adjustments such that the respective service can be consumed just according to the FlexRay restrictions). So it first needs to be transformed by applying the "inverse" of Steps 2 and 3 that were applied to β^l, and the resulting function is added to $\bar{\beta}^l$. This sum then represents the service available to the lower priority tasks, which is transformed in the same way as β^l, but using information specific to messages from task T_2. This procedure is then repeated for all the tasks $T_3, \ldots, T_n$.

To illustrate this scheme, consider the architecture shown in Figure 8(a), consisting of two tasks T_1 and T_2 running on two different ECUs. T_1 generates a periodic stream of messages (denoted by m_1) which are transmitted over the DYN segment of a FlexRay bus. The transmitted messages trigger T_2 on ECU_2, which in turn generates a stream of messages m_2 which are also transmitted over the DYN segment of the same bus to an actuator. m_1 is assigned a higher priority than m_2 and both m_1 and m_2 cannot fit into one DYN segment. Figure 8(b) shows an overview of our scheme. Here, α_1 bounds the arrival rate of m_1 at the bus and β is the service offered by the unloaded bus. β_1 is the service available to m_1. β' is the service remaining from β (i.e. *unavailable* to m_1). $\beta_1{}'$ is the service that is *unutilized* by m_1 (from what was available to it). The sum of β' and $\beta_1{}'$ is the service available to m_2. Finally, the triggering rate of T_2 (which is equal to the arrival rate of m_2 at the bus) is bounded by $\alpha_1{}'$, that is computed from α_1 and β_1.

5. CASE STUDY

We implemented our proposed framework using Matlab as the front-end, which is used for specifying the system architecture along with the relevant parameters. The back-end, implemented in Java, handles all the function transformations. In this section we present a case study based on an Adaptive Cruise Control application (ACC).

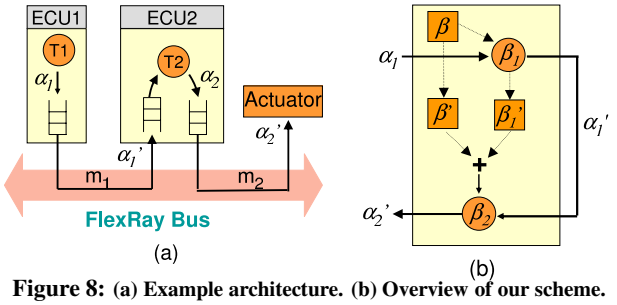

Figure 8: (a) Example architecture. (b) Overview of our scheme.

Bus		ECUs	
Message	*# Minislots*	*Task*	*WCET*
m_1	64	Data Fusion	19.7 ms
m_2	64	Object Selection	1 ms
m_3	15	Adaptive Cruise Control	4.3 ms
m_4	40	Arbitration	17.6 ms
		Actuator control	9 ms
		Object detection	12.5 ms

Table 1: System parameters for an Adaptive Cruise Control.

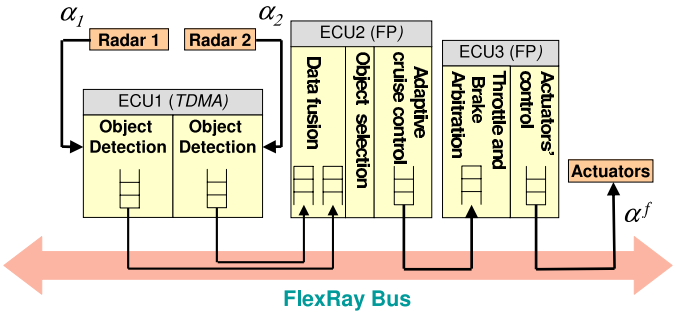

Figure 9: An Adaptive Cruise Control subsystem.

The ACC subsystem consists of three ECUs communicating via a FlexRay bus (with a communication cycle of 10 ms and DYN segment of 8 ms, consisting of 72 minislots). ECU_1 receives data from two radar sensors periodically (each with a 30 ms period). The data from each radar is processed by an *Object Detection* task running on ECU_1. The processed data streams (m_1 and m_2) are sent over the bus to a *Data Fusion* task running on ECU_2, along with *Object Selection* and *Adaptive Cruise Control* tasks. The resulting data stream (m_3) is again transmitted over the bus to ECU_3 running two other tasks and the final output (m_4) is sent to an actuator, again over the same bus. The scheduling policies for each ECU are indicated in Figure 9. All the messages are mapped onto the DYN segment of the FlexRay bus. The values of the relevant system parameters may be found in Table 1.

Figure 10(a) shows the bounds on the service offered by the unloaded bus and the remaining service for messages from other subsystems, after processing $m_1 - m_4$. Figure 10(b) shows bounds on the data arrival rates from a radar (α), the lower bound on the message arrival rate from ECU_2 onto the bus (α^l_{ACC}), and the message arrival rate at the actuator (α^l_f). From these bounds the maximum end-to-end delay (from radar to actuator) turns out to be 393 ms in contrast to a (wrong) back-of-the-envelope calculation from Table 1, which is only 60 ms.

6. CONCLUDING REMARKS

In this paper we presented a compositional performance model for a network of ECUs communicating via a FlexRay bus. Our main contribution was a formal model of the protocol governing the DYN segment of FlexRay. Throughout this paper we have assumed all messages from a specified task to be of constant (worst-case) length. Relaxing this constraint to account for variable length messages will require certain modifications to our framework which would be interesting to explore. We are also in the process of exploring possibilities of integrating our implementation into standard tools for designing FlexRay-based systems.

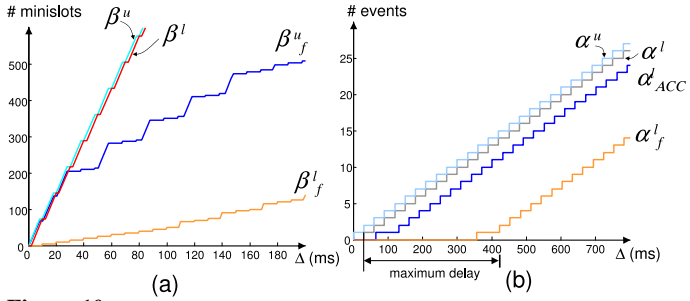

Figure 10: (a) Bounds on the service for the architecture in Figure 9. (b) Bounds on the arrival rates of messages.

7. REFERENCES

[1] A. Albert. Comparison of event-triggered and time-triggered concepts with regard to distributed control systems. In *Embedded World*, Nürnberg, Germany, 2004. `www.semiconductors.bosch.de/pdf/embedded_world_04_albert.pdf`.

[2] CAN Specification, Ver 2.0, Robert Bosch GmbH. `www.semiconductors.bosch.de/pdf/can2spec.pdf`, 1991.

[3] S. Chakraborty, S. Künzli, and L. Thiele. A general framework for analysing system properties in platform-based embedded system designs. In *DATE*, 2003.

[4] J. Csirik, J. B. G. Frenk, M. Labbé, and S. Zhang. Two simple algorithms for bin covering. *Acta Cybernetica*, 14(1):13–25, 1999.

[5] DECOMSYS - Dependable Computer Systems, Hardware und Software Entwicklung GmbH. `www.decomsys.com`.

[6] dSPACE GmbH. `www.dspace.de`.

[7] J. Ferreira, P. Pedreiras, L. Almeida, and J. A. Fonseca. The FTT-CAN protocol for flexibility in safety-critical systems. *IEEE Micro*, 22(4):46–55, 2002.

[8] The FlexRay Communications System Specifications, Ver. 2.1. `www.flexray.com`.

[9] Class B Data Communications Network Interface, SAE J1850 Srandard, Rev. 2, Nov. 1996. `www.interfacebus.com/Automotive_SAE_J1850_Bus.html`.

[10] Local Interconnect Network Specification, Lin Consortium. `www.lin-subbus.org`.

[11] C. A. Lupini, T. J. Haggerty, and T. A. Braun. Class 2: General Motors' version of SAE J1850. In *8th Intl. Conf. on Automotive Electronics*, London, 1991.

[12] N. Navet, Y. Q. Song, F. Simonot-Lion, and C. Wilwert. Trends in automotive communication systems. *Proceedings of the IEEE (special issue on Industrial Communications Systems)*, 96(6):1204–1223, 2005.

[13] P. Pop, P. Eles, and Z. Peng. Schedulability-driven communication synthesis for time-triggered embedded systems. *Real-Time Systems*, 26(3):297–325, 2004.

[14] T. Pop, P. Pop, P. Eles, Z. Peng, and A. Andrei. Timing analysis of the FlexRay communication protocol. In *18th Euromicro Conference on Real-Time Systems (ECRTS)*, 2006.

[15] K. Tindell, A. Burns, and A. Wellings. Calculating Controller Area Network (CAN) message response times. *Control Engineering Practice*, 3(8):1163–1169, 1995.

[16] K. Tindell and J. Clark. Holistic schedulability analysis for distributed hard real-time systems. *Microprocessing and Microprogramming*, 40(2-3):117 – 134, 1994.

[17] K. Tindell, H. Hanssmon, and A. J. Wellings. Analysing real-time communications: Controller Area Network (CAN). In *IEEE Real-Time Systems Symposium (RTSS)*, 1994.

[18] ISO/CD11898-4, Road Vehicles Controller Area Network (CAN) Part 4: Time-Triggered Communication, International Standards Organization, Geneva, 2000.