Real-Time Syst (2008) 39: 205–235
DOI 10.1007/s11241-007-9040-3

Timing analysis of the FlexRay communication protocol

Traian Pop · Paul Pop · Petru Eles · Zebo Peng · Alexandru Andrei

Published online: 20 October 2007

Abstract FlexRay is a communication protocol heavily promoted on the market by a large group of car manufacturers and automotive electronics suppliers. However, before it can be successfully used for safety-critical applications that require predictability, timing analysis techniques are necessary for providing bounds for the message communication times. In this paper, we propose techniques for determining the timing properties of messages transmitted in both the static and the dynamic segments of a FlexRay communication cycle. The analysis techniques for messages are integrated in the context of a holistic schedulability analysis that computes the worst-case response times of all the tasks and messages in the system. We have evaluated the proposed analysis techniques using extensive experiments. We also present and evaluate three optimisation algorithms that can be used to improve the schedulability of a system that uses FlexRay.

Keywords Real-time analysis · Distributed embedded systems · FlexRay

1 Introduction

Many safety-critical applications, following physical, modularity or safety constraints, are implemented using distributed architectures composed of several different types of hardware units (called nodes), interconnected in a network. For such systems, the communication between functions implemented on different nodes has an important impact on the overall system properties, such as performance, cost and maintainability.

T. Pop (✉) · P. Pop · P. Eles · Z. Peng · A. Andrei
Linköping University, 58183 Linköping, Sweden
e-mail: trapo@ida.liu.se

There are several communication protocols for real-time networks. Among the protocols that have been proposed for in-vehicle communication, only the Controller Area Network (CAN) (R. Bosch GmbH 1991), the Local Interconnection Network (LIN) (Local Interconnect Network Protocol Specification 2005), and SAE's J1850 (SAE 1994) are currently in use on a large scale (Navet et al. 2005). Moreover, only a few of the proposed protocols are suitable for safety-critical applications where predictability is mandatory (Rushby 2001).

Communication activities can be triggered either dynamically, in response to an event (event-driven), or statically, at predetermined moments in time (time-driven). Therefore, on one hand, there are protocols that schedule the messages statically based on the progression of time, such as the SAFEbus (Hoyme and Driscoll 1992), SPIDER (Miner 2000), TTCAN (International Organization for Standardization 2002), and Time-Triggered Protocol (TTP) (Kopetz and Bauer 2003). The main drawback of such protocols is their lack of flexibility. On the other hand, there are communication protocols where message scheduling is performed dynamically, such as Byteflight (Berwanger et al. 2000) introduced by BMW for automotive applications, CAN (R. Bosch GmbH 1991), LonWorks (Echelon 2005) and Profibus (Profibus International 2005).

A large consortium of automotive manufacturers and suppliers has recently proposed a hybrid type of protocol, namely the FlexRay communication protocol (FlexRay 2005). FlexRay allows the sharing of the bus among event-driven (ET) and time-driven (TT) messages, thus offering the advantages of both worlds. FlexRay will possibly become the de facto standard for in-vehicle communications. However, before it can be successfully deployed in applications that require predictability, timing analysis techniques are necessary to provide bounds for the message communication times (Navet et al. 2005).

FlexRay is composed of static (ST) and dynamic (DYN) segments, which are arranged to form a bus cycle that is repeated periodically. The ST segment is similar to TTP, and employs a generalized time-division multiple-access (GTDMA) scheme. The DYN segment of the FlexRay protocol is similar to Byteflight and uses a flexible TDMA (FTDMA) bus access scheme.

Although researchers have proposed analysis techniques for dynamic protocols such as CAN (Tindell et al. 1995), TDMA (Tindell and Clark 1994), ATM (Ermedahl et al. 1997), Token Ring protocol (Strosnider and Marchok 1989), FDDI protocol (Agrawal et al. 1994) and TTP (Pop et al. 2004), none of these analyses is applicable to the DYN segment in FlexRay. In Ding et al. (2005), the authors consider the case of a hard real-time application implemented on a FlexRay bus. However, in their discussion they restrict themselves exclusively to the static segment, which means that, in fact, only the classical problem of communication scheduling over a TDMA bus (Pop et al. 2004; Hamann and Ernst 2005) is considered. The performance analysis of the Byteflight protocol, which is similar to the DYN segment of FlexRay, is discussed in Cena and Valenzano (2004). The authors, however, assume a very restrictive "quasi-TDMA" transmission scheme for time-critical messages, which basically means that the DYN segment would behave as an ST segment (similar to TDMA) in order to guarantee timeliness.

In this paper we present an approach to timing analysis of applications communicating over a FlexRay bus, taking into consideration the specific aspects of this protocol, including the DYN segment. More exactly, we propose techniques for determining the timing properties of messages transmitted in the static and the dynamic segments of a FlexRay communication cycle. We first briefly present a static cyclic scheduling technique for TT messages transmitted in the ST segment, which extends our previous work on the TTP (Pop et al. 2000). Then, we develop a worst-case response time analysis for ET messages sent using the DYN segment, thus providing predictability for messages transmitted in this segment. The analysis techniques for messages are integrated in the context of a holistic schedulability analysis algorithm that computes the worst-case response times of all the tasks and messages in the system.

Such an analysis, while being able to bound the message transmission times on both the ST and DYN segments, represents the first step towards enabling the use of this protocol in a systematic way for time critical applications. The second step towards an efficient use of FlexRay is taken in Sect. 6 of this paper, where we propose several optimisation techniques that consider the particular features of an application during the process of finding a FlexRay bus configuration that can guarantee that all time constraints are satisfied.

This paper is organized in seven sections. Section 2 presents the system architecture considered, and Sect. 3 introduces the FlexRay media access control. In Sect. 4 we present the application model that we use. The main part of the paper is concentrated in Sect. 5, where we present our timing analysis for distributed real-time systems that use the FlexRay protocol, together with the experimental results we have run in order to determine the efficiency of our approaches. In the same section we extend the analysis to capture the independent usage of the two FlexRay channels. Section 6 shows how system schedulability is improved as a result of careful bus access optimisation. The last section presents our conclusions.

2 System model

We consider architectures consisting of nodes connected by one FlexRay communication channel[1] (see Fig. 1a). Each processing node connected to a FlexRay bus is composed of two main components: a CPU and a communication controller (see Fig. 2a) that are interconnected through a two-way controller-host interface (CHI). The controller runs independently of the node's CPU and implements the FlexRay protocol services.

For the systems we are studying, we have designed a software architecture which runs on the CPU of each node. The main component of the software architecture is a real-time kernel that contains two schedulers, for static cyclic scheduling (SCS) and fixed priority scheduling (FPS), respectively[2] (see Fig. 1b).

When several tasks are ready on a node, the task with the highest priority is activated, and preempts the other tasks. Let us consider the example in Fig. 1b, where we have six tasks sharing the same node. Tasks τ_1 and τ_6 are scheduled using SCS, while

[1] FlexRay is a dual-channel bus, aspect discussed in Sect. 5.3.

[2] EDF can also be added, as presented by us in Pop et al. (2005b).

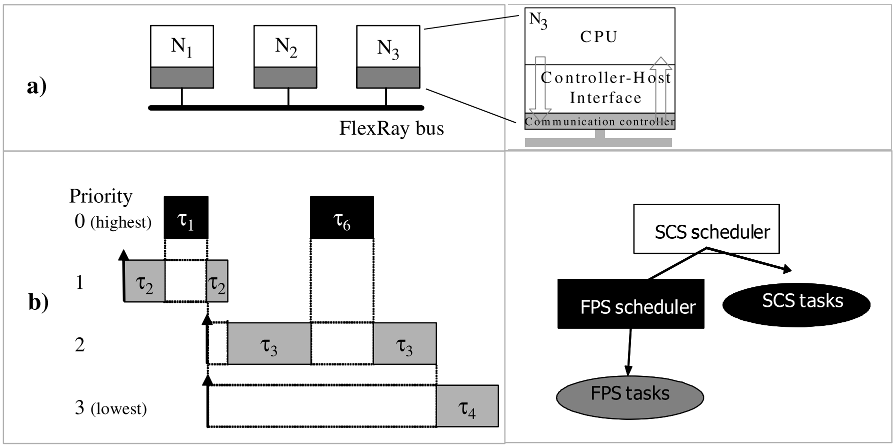


Fig. 1 System architecture example

the rest are scheduled with FPS. The priorities of the FPS tasks are indicated in the figure. The arrival time of a task is depicted with an upwards pointing arrow. Under these assumptions, Fig. 1b presents the worst-case response times of each task. SCS tasks are non preemptable and their start time is off-line fixed in the schedule table (they also have the highest priority, denoted with priority level "0" in the figure). FPS tasks can only be executed in the slack of the SCS schedule table.

FPS tasks are scheduled based on priorities. Thus, a higher priority task such as τ_3 preempts a lower priority task such as τ_4. SCS activities are triggered based on a local clock in each processing node. The synchronization of local clocks throughout the system is provided by the communication protocol (FlexRay 2005).

3 The FlexRay communication protocol

In this section we will describe how messages generated by the CPU reach the communication controller and how they are transmitted on the bus. Let us consider the example in Fig. 2 where we have three nodes, N_1 to N_3 sending messages $m_a, m_b, \ldots, m_h$ using a FlexRay bus.

In FlexRay, the communication takes place in periodic cycles (Fig. 2b depicts two cycles of length T_{bus}). Each cycle contains two time intervals with different bus access policies: an ST segment and a DYN segment.[3] The ST and DYN segment lengths can differ, but are fixed over the cycles. We denote with ST_{bus} and DYN_{bus} the length of these segments. Both the ST and DYN segments are composed of several slots. In the ST segment, the slots number is fixed, and the slots have constant and equal length, regardless of whether ST messages are sent or not over the bus in

[3]The FlexRay bus cycle contains also a *symbol window* and a *network idle time*, but their size does not affect the equations in our analysis. For simplicity, they will be ignored during the examples throughout the paper.

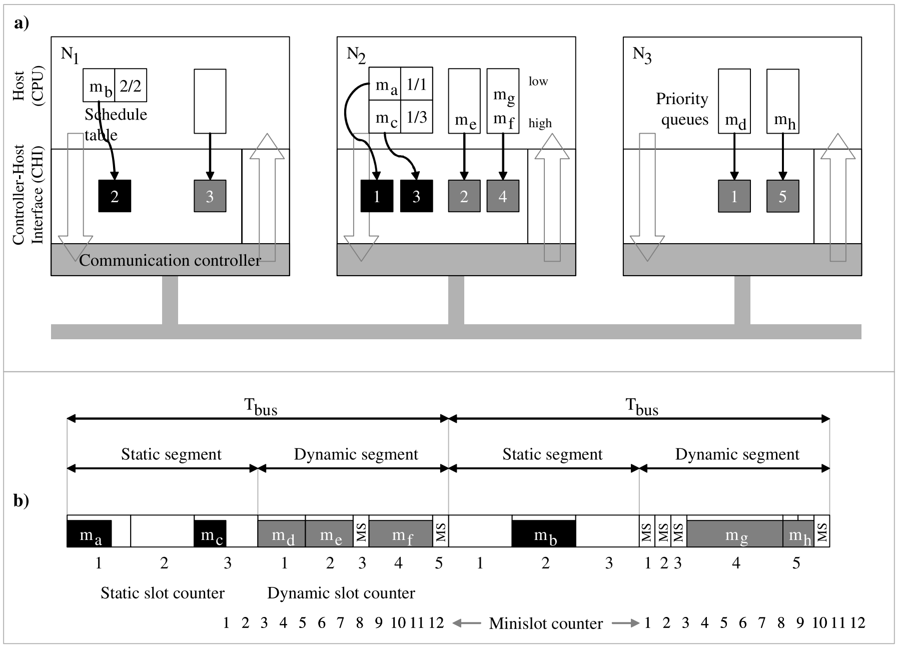


Fig. 2 FlexRay communication cycle example

that cycle. The length of an ST slot is specified by the FlexRay global configuration parameter *gdStaticSlot* (FlexRay 2005). In Fig. 2 there are three static slots for the ST segment.

The length of the DYN segment is specified in number of "minislots", and is equal to *gNumberOfMinislots*. Thus, during the DYN segment, if no message is to be sent during a certain slot, then that slot will have a very small length (equal to the length *gdMinislot* of a so called minislot), otherwise the DYN slot will have a length equal with the number of minislots needed for transmitting the whole message (FlexRay 2005). This can be seen in Fig. 2b, where DYN slot 2 has 3 minislots (4, 5, and 6) in the first bus cycle, when message m_e is transmitted, and one minislot (denoted with "MS" and corresponding to the minislot counter 2) in the second bus cycle when no message is sent.

During any slot (ST or DYN), only one node is allowed to send on the bus, and that is the node which holds the message with the frame identifier (*FrameID*) equal to the current value of the slot counter. There are two slot counters, corresponding to the ST and DYN segments, respectively. The assignment of frame identifiers to nodes is static and decided offline, during the design phase. Each node that sends messages has one or more ST and/or DYN slots associated to it. The bus conflicts are solved by allocating offline one slot to at most one node, thus making it impossible for two nodes to send during the same ST or DYN slot.

In Fig. 2, node N_1 has been allocated ST slot 2 and DYN slot 3, N_2 transmits through ST slots 1 and 3 and DYN slots 2 and 4, while node N_3 has DYN slots 1 and 5. For each of these slots, the CHI reserves a buffer that can be written by the

CPU and read by the communication controller (these buffers are read by the communication controller *at the beginning* of each slot, in order to prepare the transmission of frames). The associated buffers in the CHI are depicted in Fig. 2a. We denote with $DYNSlots_{N_p}$ the number of dynamic slots associated to a node N_p (this means that for N_2 in Fig. 2, $DYNSlots_{N_2}$ has value 2).

We use different approaches for ST and DYN messages to decide which messages are transmitted during the allocated slots. For ST messages, we consider that the CPU in each node holds a schedule table with the transmission times. When the time comes for an ST message to be transmitted, the CPU will place that message in its associated ST buffer of the CHI. For example, ST message m_b sent from node N_1 has an entry "2/2" in the schedule table specifying that it should be sent in the second slot of the second ST cycle.

For the DYN messages, we assume that the designer specifies their *FrameID*. For example, DYN message m_e has the frame identifier "2". While nodes must use distinct *FrameID*s (and consequently distinct DYN slots) in order to avoid bus conflicts, we allow for a node to send different messages using the same DYN *FrameID*.[4] For example, messages m_g and m_f on node N_2 have both *FrameID* 4. If two or more messages with the same frame identifier are ready to be sent in the same bus cycle, a priority scheme is used to decide which message will be sent first. Each DYN message m_i has associated a priority $priority_{m_i}$. Messages with the same *FrameID* will be placed in a local output queue ordered based on their priorities. The message form the head of the priority queue is sent in the current bus cycle. For example, message m_f will be sent before m_g because it has a higher priority.

At the beginning of each communication cycle, the communication controller of a node resets the slot and minislot counters. At the beginning of each communication slot, the controller verifies if there are messages ready for transmission (present in the CHI send buffers) and packs them into frames.[5] In the example in Fig. 2 we assume that all messages are ready for transmission before the first bus cycle.

Messages selected and packed into ST frames will be transmitted during the bus cycle that is about to start according to the schedule table. For example, in Fig. 2, messages m_a and m_c are placed into the associated ST buffers in the CHI in order to be transmitted in the first bus cycle. However, messages selected and packed into DYN frames will be transmitted during the DYN segment of the bus cycle only if there is enough time until the end of the DYN segment. Such a situation is verified by comparing if, in the moment the DYN slot counter reaches the value of the *FrameID* for that message, the value of the minislot counter is smaller than a given value *pLatestTx*. The value *pLatestTx* is fixed for each node during the design phase, depending on the size of the largest DYN frame that node will have to send during run-time. For example, in Fig. 2, message m_h is ready for transmission before the first bus cycle starts, but, after message m_f is transmitted, there is not enough room left in the DYN segment. This will delay the transmission of m_h for the next bus cycle.

[4]This assumption is not part of the FlexRay specification. If messages are not sharing *FrameID*s, this is handled implicitly as a particular case of our analysis.

[5]In this paper we do not address frame-packing (Pop et al. 2005a), and thus assume that one message is sent per frame.

4 Application model

We model an application $\mathcal{A}$ as a set of directed, acyclic, polar graphs $\mathcal{G}_i(\mathcal{V}_i, \mathcal{E}_i) \in \mathcal{A}$. A node $\tau_{ij} \in \mathcal{V}_i$ represents the jth task or message in $\mathcal{G}_i$. An edge $e_{ijk} \in \mathcal{E}_i$ from τ_{ij} to τ_{ik} indicates that the output of τ_{ij} is the input of τ_{ik}. A task becomes ready after all its inputs have arrived and it issues its outputs when it terminates. A message will become ready after its sender task has finished, and becomes available for the receiver task after its transmission has ended. The communication time between tasks mapped on the same processor is considered to be part of the task worst-case execution time and is not modeled explicitly. Communication between tasks mapped to different processors is performed by message passing over the bus. Such message passing is modeled as a communication task inserted on the arc connecting the sender and the receiver task.

We consider that the scheduling policy for each task is known (either *SCS* or *FPS*), and we also know which messages are ST and which are DYN. For a task $\tau_{ij} \in \mathcal{V}_i$, $Node_{\tau_{ij}}$ is the node to which τ_{ij} is assigned for execution. When executed on $Node_{\tau_{ij}}$, a task τ_{ij} has a known worst-case execution time $C_{\tau_{ij}}$. We also consider that the size of each message m is given, which can be directly converted into communication time C_m on the particular bus, knowing the speed of the bus and the size of the frame that stores the message:

$$C_m = Frame\_size(m)/bus\_speed. \tag{1}$$

Tasks and messages activated based on events also have a priority, $priority_{\tau_{ij}}$. All tasks and messages belonging to a task graph G_i have the same period $T_{\tau_{ij}} = T_{\mathcal{G}_i}$ which is the period of the process graph. A deadline $D_{\mathcal{G}_i}$ is imposed on each task graph $\mathcal{G}_i$. In addition, tasks can have associated individual release times and deadlines.

5 Timing analysis

Given a distributed system based on FlexRay, as described in the previous two sections, the tasks and messages have to be scheduled. For the SCS tasks and ST messages, this means building the schedule tables, while for the FPS tasks and DYN messages we have to determine their worst-case response times.

The problem of finding a schedulable system has to consider two aspects:

1. When performing the schedulability analysis for the FPS tasks and DYN messages, one has to take into consideration the interference from the SCS activities.
2. Among the possible correct schedules for SCS activities, it is important to build one which favours as much as possible the schedulability of FPS activities.

Figure 3 presents the global scheduling and analysis algorithm, in which the main loop consists of a list-scheduling based algorithm (Coffman and Graham 1972) that iteratively builds the static schedule table with start times for SCS tasks and ST messages.

```
GlobalSchedulingAlgorithm()
  1  while TT_ready_list is not empty
  2    select $\tau_{ij}$ from TT_ready_list
  3    if $\tau_{ij}$ is a SCS task then
  4      schedule_TT_task($\tau_{ij}$, $Node_{\tau_{ij}}$)
  5    else // $\tau_{ij}$ is a ST message
  6      schedule_ST_msg($\tau_{ij}$, $Node_{\tau_{ij}}$)
  7    end if
  8    update TT_ready_list
  9  end while
end StaticScheduling
schedule_TT_task($\tau_{ij}$, $Node_{\tau_{ij}}$)
  10   find first available time moment $t_s$ after $ASAP_{\tau_{ij}}$
       on $Node_{\tau_{ij}}$
  11   schedule $\tau_{ij}$ after $t_s$ on $Node_{\tau_{ij}}$, so that holistic analysis
       produces minimal worst-case response times
       for FPS tasks and DYN messages
  12   update ASAP for all $\tau_{ij}$ successors
end schedule_TT_task
schedule_ST_msg($\tau_{ij}$, $Node_{\tau_{ij}}$)
  13   find first ST slot($Node_{\tau_{ij}}$) available after $ASAP_{\tau_{ij}}$
  14   schedule $\tau_{ij}$ in that ST slot
  15   update ASAP for all $\tau_{ij}$ successors
end schedule_ST_msg
```

Fig. 3 Global scheduling algorithm

A ready list (*TT_ready_list*) contains all SCS tasks and ST messages which are ready to be scheduled (they have no predecessors or all their predecessors have already been scheduled). From the ready list, tasks and messages are extracted one by one (Fig. 3, line 2) to be scheduled on the processor they are mapped to (line 4), or into a static bus-slot associated to that processor on which the sender of the message is executed (line 6), respectively. The priority function which is used to select among ready tasks and messages is a critical path metric, modified by us for the particular goal of scheduling tasks mapped on distributed systems (Pop et al. 2000). Let us consider a particular task τ_{ij} selected from the ready list to be scheduled. We consider that $ASAP_{\tau_{ij}}$ is the earliest time moment which satisfies the condition that all preceding activities (tasks or messages) of τ_{ij} are finished (line 10). With only the SCS tasks in the system, the straightforward solution would be to schedule τ_{ij} at the first time moment after $ASAP_{\tau_{ij}}$ when $Node_{\tau_{ij}}$ is free. Similarly, an ST message will be scheduled in the first available ST slot associated with the node that runs the sender task for that message.

As presented by us in Pop et al. (2003), when scheduling SCS tasks, one has to take into account the interference they produce on FPS tasks. The function *schedule_TT_task* in Fig. 3 places a SCS task in the static schedule in such a way that the increase of worst-case response times for FPS tasks is minimized. Such an increase is determined by comparing the worst-case response times of FPS tasks obtained with our holistic schedulability analysis before and after inserting the new SCS task in the schedule (Pop et al. 2003).

The next subsection presents our solution for computing the worst case response times of DYN messages, while in Sect. 5.2 we will integrate this solution into a holistic schedulability analysis that determines the timing properties of both FPS tasks and DYN messages (which is called in line 11, of *schedule_TT_task* presented in Fig. 3).

5.1 Schedulability analysis of DYN messages

The worst case response time R_m of a DYN message m is given by the following equation:

$$R_m(t) = \sigma_m + w_m(t) + C_m, \tag{2}$$

where C_m is the message communication time (see Sect. 4), σ_m is the longest delay suffered during one bus cycle if the message is generated by its sender task after its slot has passed, and w_m is the worst case delay caused by the transmission of ST frames and higher priority DYN messages during a given time interval t. For example, in Fig. 4, we consider that a message m is supposed to be transmitted in the 3rd DYN slot of the bus cycle. The figure presents the case when message m appears during the first bus cycle after the 3rd DYN slot has passed, therefore the message has to wait σ_m until the next bus cycle starts. In the second bus cycle, the message has to wait for the ST segment and for the first two DYN slots to finish, delay denoted with w_m (that also contains the transmission of a message m' that uses the second DYN slot).

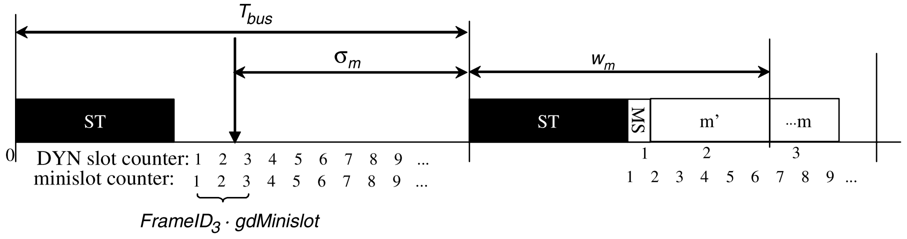
The communication controller decides what message is to be sent on the bus in a certain communication slot *at the beginning* of that slot. As a consequence, in the worst case, a DYN message m is generated by its sender task immediately after the slot with the $\mathit{FrameID}_m$ has started, forcing message m to wait until the next bus cycle starts in order to really start competing for the bus. In conclusion, in the worst case, the delay σ_m has the value:

$$\sigma_m = T_{bus} - (ST_{bus} + (\mathit{FrameID}_m - 1) \times \mathit{gdMinislot}), \tag{3}$$

where ST_{bus} is the length of the ST segment.

What is now left to be determined is the value w_m corresponding to the maximum amount of delay on the bus that can be produced by interference from ST frames and DYN messages. We start from the observations that the transmission of a ready DYN message m during the DYN slot $\mathit{FrameID}_m$ can be delayed because of the following causes:

- Local messages with higher priority, that use the same frame identifier as m. We will denote this set of *higher priority local messages* with $hp(m)$. For example, in Fig. 2a, messages m_g and m_f share *FrameID* 4, thus $hp(m_g) = \{m_f\}$.



T_{bus}
σ_m
w_m
ST
ST
MS
m'
...m
0
DYN slot counter: 1 2 3 4 5 6 7 8 9 ...
minislot counter: 1 2 3 4 5 6 7 8 9 ...
1 2 3
1 2 3 4 5 6 7 8 9 ...
$\mathit{FrameID}_3 \cdot \mathit{gdMinislot}$

Fig. 4 Response time of a DYN message

- Any messages in the system that can use DYN slots with lower frame identifiers than the one used by m. We will denote this set of messages having *lower frame identifiers* with $lf(m)$. In Fig. 2a, $lf(m_g) = \{m_d, m_e\}$.
- Unused DYN slots with frame identifiers lower than the one used for sending m (though such slots are unused, each of them still delays the transmission of m for an interval of time equal with the length *gdMinislot* of one minislot); we will denote the set of such minislots with $ms(m)$. Thus, in the example in Fig. 2b, $ms(m_g) = \{1, 2, 3\}$, and $ms(m_f) = \{3\}$.

Determining the interference of DYN messages in FlexRay is complicated by several factors. Let us consider the example in Fig. 5, where we have two nodes, N_1 (with *FrameIDs* 1 and 3) and N_2 (with *FrameID* 2), and three messages m_1 to m_3. N_1 sends m_1 and m_3, and N_2 sends message m_2. Messages m_1 and m_2 have *FrameIDs* 1 and 2, respectively. We consider two situations: Fig. 5a, where m_3 has a separate *FrameID* 3, and Fig. 5b, where m_3 shares the same *FrameID* 1 with m_1. The values of *pLatestTx* for each node are depicted in the figure.[6]

In Fig. 5a, message m_2, that has a lower FrameID than m_3, cannot be sent immediately after message m_1, because the value of the minislot counter has exceeded the value $pLatestTx_{m_2}$ when the value of the DYN slot counter becomes equal to 2 (hence, m_2 does not fit in this DYN cycle). As a consequence, the transmission of m_2 will be delayed for the next bus cycle. However, since in the moment when the DYN slot counter becomes 3 the minislot counter does not exceed the value $pLatestTx_{m_3}$, message m_3 will fit in the first bus cycle. Thus, a message (m_3 in our case) can be sent before another message with a lower $FrameID(m_2)$. Such situations must be accounted for when building the worst-case scenario.

In Fig. 5b, message m_3 shares the same *FrameID* 1 with m_1 but we consider that it has a lower priority, thus $hp(m_3) = \{m_1\}$. In this case, m_3 is sent in the first DYN slot of the second bus cycle (the first slot of the first cycle is occupied with m_1) and thus will delay the transmission of m_2. In this scenario, we notice that assigning a lower frame identifier to a message does not necessarily reduce the worst-case response time of that message (compare to the situation in Fig. 5a, where m_3 has $FrameID = 3$).

We next focus on determining the delay $w_m(t)$ in (2). The delay produced by all the elements in $hp(m)$, $lf(m)$ and $ms(m)$ can extend to one or more bus cycles:

$$w_m(t) = BusCycles_m(t) \times T_{bus} + w'_m(t), \tag{4}$$

where $BusCycles_m(t)$ is the number of bus periods for which the transmission of m is not possible because transmission of messages from $hp(m)$ and $lf(m)$ and because of minislots in $ms(m)$. The delay $w'_m(t)$ denotes now the time that passes, in the last bus cycle, until m is sent, and is measured from the beginning of the bus cycle in which message m is sent until the actual transmission of m starts. For example, in Fig. 5b, $BusCycles_{m_2} = 1$ and $w'_{m_2}(t) = ST_{bus} + C_{m_3}$. Note that both these terms are functions of time, computed over an analyzed interval t. This means that when computing them we have to take into consideration all the elements in $hp(m)$, $lp(m)$ and $ms(m)$ that

[6] We use $pLatestTx_m$ to denote $pLatestTx_N$ of the node N sending message m.

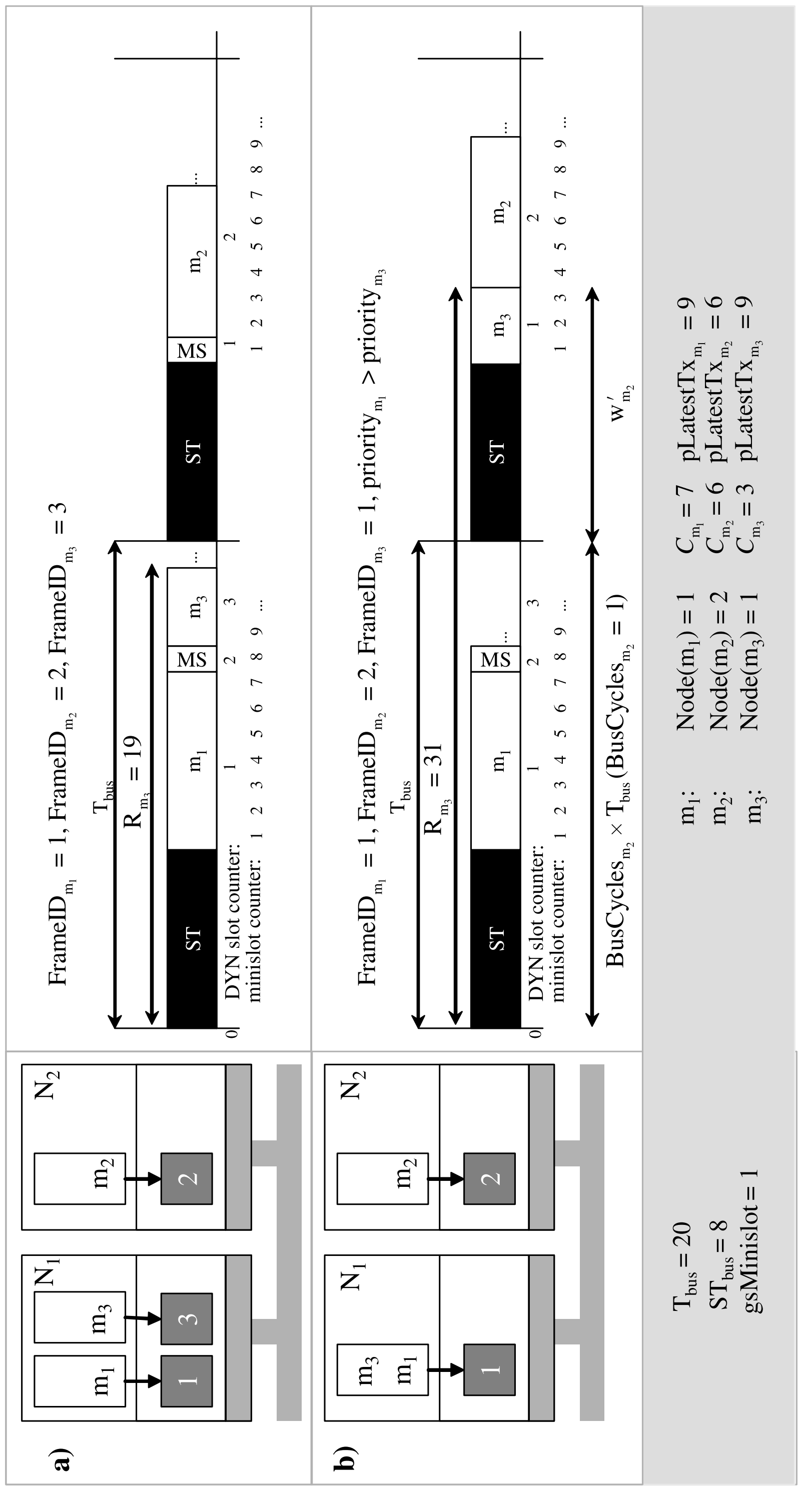

a)
b)
N1
N2
m1
m2
m3
1
2
3
FrameID m1 = 1, FrameID m2 = 2, FrameID m3 = 3
FrameID m1 = 1, FrameID m2 = 2, FrameID m3 = 1, priority m1 > priority m3
Tbus
R m3 = 19
R m3 = 31
ST
MS
DYN slot counter:
minislot counter:
BusCycles m2 × Tbus (BusCycles m2 = 1)
w′ m2
Tbus = 20
STbus = 8
gsMinislot = 1
m1: Node(m1) = 1 C m1 = 7 pLatestTx m1 = 9
m2: Node(m2) = 2 C m2 = 6 pLatestTx m2 = 6
m3: Node(m3) = 1 C m3 = 3 pLatestTx m3 = 9

Fig. 5 Transmission scenarios for DYN messages

can appear during such a given time interval t. Thus, we will consider the multiset $hp(m,t)$ containing all the occurrences over time interval t of elements in $hp(m)$. The number of such occurrences for a message $l \in hp(m)$ is equal to: $\lceil (J_l + t)/T_l \rceil$, where T_l is the period of the message l and J_l is its worst-case jitter (such a jitter is computed as the difference between the worst-case and best-case response times of its sender task s: $J_l = R_s - R_s^b$ (Palencia and Gonzaléz Harbour 1998)). Similarly, $lf(m,t)$ and $ms(m,t)$ consider all the occurrences over t of elements in $lf(m)$ and $ms(m)$ respectively.

The next two sections (5.1.1 and 5.1.2) present the optimal (i.e., exact) solutions for determining the values for $BusCycles_m(t)$ and $w'_m(t)$, respectively. These, however, can be intractable for larger problem sizes. Hence, in Sects. 5.1.3 and 5.1.4 we propose heuristics that quickly compute upper bounds (i.e., pessimistic) values for these terms. Once for any given time interval t we know how to obtain the values $BusCycles(t)$ and $w'_m(t)$, determining the worst case response time for a message m becomes an iterative process that computes $R_m^k(R_m^{k-1})$, starting from $R_m^0 = C_m$ and finishing when $R_m^k = R_m^{k-1}$.

5.1.1 Optimal solution for $BusCycles_m$

We start with the observation that a message m with $FrameID_m$ cannot be sent by a node N_p during a bus cycle b if at least one of the following conditions is fulfilled:

1. There is too much interference from elements in $lf(m)$ and $ms(m)$, so that the minislot counter exceeds the value $pLatestTx_{N_p}$, making it impossible for N_p to start the transmission of m during b. For example in Fig. 5a, message m_2 cannot be sent during the first bus cycle because the transmission of a higher priority message m_1 pushes the minislot counter over the value $pLatestTx_{N_2}$.
2. The DYN slot $FrameID_m$ in b is used by another local higher priority message from $hp(m)$. For example, in Fig. 5b, messages m_1 and m_3 share the same frame identifier and $hp(m_3) = \{m_1\}$. Therefore, the transmission of m_3 in the first bus cycle is not possible.

Whenever a bus cycle satisfies at least one of these two conditions, it will be called "filled", since it is unusable for the transmission of the message m under analysis. In the worst case, the value $BusCycles_m(t)$ is then the maximum number of bus cycles that can be filled using elements from $hp(m)$, $lf(m)$ and $ms(m)$.

We start with the observation that each of the two conditions above, when true, can prevent the message m for being transmitted during the current bus cycle. For example, a bus cycle in which a message from $hp(m)$ is ready for transmission will be completely unusable for the transmission of m, regardless if there are any messages from $lf(m)$. Similarly, if the messages from $lf(m)$ are long enough so that the DYN slot counter reaches $FrameID_m$ after the minislot counter exceeds $pLatestTx_m$, then that bus cycle will be unusable for the transmission of m, regardless of the fact that there are messages from $hp(m)$. Since messages in $hp(m,t)$ and $lf(m,t)$ can become ready at any point during the analyzed interval t, this means that in the worst case, each filled bus cycle will contain either only messages from $lf(m,t)$, or only one message from $hp(m,t)$. For example, considering the same setup presented in Fig. 2,

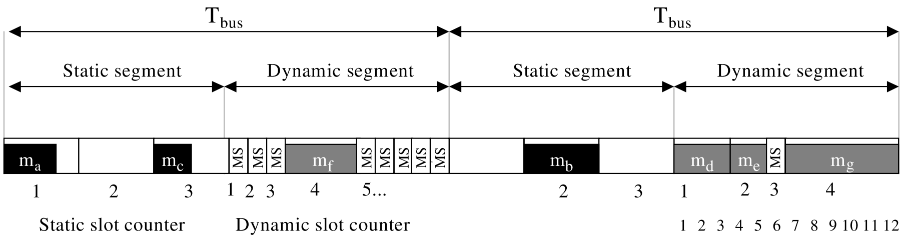


Fig. 6 Worst case scenario for DYN message m_g

the worst-case scenario for message m_g is when message m_f from $hp(m_g)$ is ready at the beginning of the first bus cycle and messages m_d and m_e from $lf(m_g)$ become ready just before the start of their slots in the second bus cycle (see Fig. 6 for the worst-case scenario of m_g).

This means that, in the worst case, the delay produced by elements in $lf(m,t)$ and $ms(m,t)$ adds up to that produced by messages in $hp(m,t)$:

$$BusCycles_m(t) = BusCycles_m(hp(m,t)) + BusCycles_m(lf(m,t), ms(m,t)), \tag{5}$$

where we denote with $BusCycles_m(hp(m,t))$ the number of bus cycles in which the delay of the message m under analysis is produced by messages in $hp(m,t)$ (corresponding to the second case presented above); similarly, $BusCycles_m(lf(m,t), ms(m,t))$ is the number of "filled" bus cycles in which the transmission of message m is delayed by elements in $lf(m,t)$ and $ms(m,t)$ (corresponding to the first condition presented above).

Since each message in $hp(m,t)$ delays the transmission of m with one bus cycle, the occurrences over time interval t of messages in $hp(m)$ will produce a delay equal to the total number of elements in $hp(m,t)$:

$$BusCycles_m(hp(m,t)) = |hp(m,t)|. \tag{6}$$

The problem that remains to be solved is to determine how many bus cycles can be "filled" according to the first condition presented above using only elements in $lf(m,t)$ and $ms(m,t)$. As we will discuss later, a simplified version of this problem is equivalent to bin covering, which belongs to the family of NP-hard problems (Labbe et al. 1995). To obtain the optimal solution, we have modelled the problem of computing $BusCycles_m(lf(m,t), ms(m,t))$ as an integer linear program (ILP). The model starts from the observation that, considering we have n elements in $lf(m,t)$, there are at most n bus cycles that can be filled. For each such bus cycle we create a binary variable $y_{i=1\ldots n}$ that is set to 1 when the ith bus cycle is filled with elements from $lf(m,t)$ and $ms(m,t)$, and to 0 if it is not filled (i.e., it can allow the transmission of message m under analysis).

The goal of the ILP problem is to maximize the number of filled bus cycles (i.e., to calculate the worst-case):

$$BusCycles_m(lf(m,t), ms(m,t)) = \sum_{i=1\ldots n} y_i, \tag{7}$$

subject to a set of conditions that set the variables y_i to 1 or 0. Bellow we describe these conditions, which capture how messages in $lf(m,t)$ and the minislots in $ms(m,t)$ are sent by FlexRay in these bus cycles.

We allocate a binary variable x_{ijk} that is set to 1 if a message $m_k \in lf(m,t)$ ($k = 1 \dots n$) is sent during the ith bus cycle, using the $FrameID\ j = 1 \dots FrameID_m$. The load transmitted in each bus cycle can be expressed as:

$$Load_i = \sum_{\substack{m_k \in lf(m,t) \\ j=1 \dots FrameID_m}} (x_{ijk} \times C_k + (1 - x_{ijk}) \times gdMinislot), \tag{8}$$

where C_k are the communication times (1) of the messages $m_k \in lf(m,t)$. Each term of the sum in (8) captures the particularities of FlexRay DYN frames: if a message k is transmitted in cycle i with frame identifier j, then $x_{ijk} = 1$ and the length of the frame being transmitted is equal with the length of the message k (thus the term $x_{ijk} \times C_k$); if x_{ijk} is 0 for all j and k, then there is no actual transmission on the bus in that DYN slot, but there is still some delay due to the empty minislot of length *gdMinislot* that has to pass in order to increase the value of the DYN slot counter (thus the second term).

The condition that sets each variable y_i to 1 whenever possible is:

$$Load_i > pLatestTx_{N_p} \times gdMinislot \times y_i, \tag{9}$$

where $pLatestTx_{N_p}$ is the last minislot which allows the start of transmission from node N_p which generates the message m under analysis. Such a condition enforces that a variable y_i cannot be set to 1 unless the total amount of interference from $lf(m,t)$ and $ms(m,t)$ in cycle i exceeds $pLatestTx_{N_p}$ minislots (only then message m is not allowed to be transmitted and, thus, bus cycle i is "filled").

In addition to this condition we have to make sure that

- Each message $m_k \in lf(m,t)$ is sent in only one cycle i:

$$\sum_{\substack{i=1 \dots n \\ j=1 \dots FrameID_m}} x_{ijk} \le 1, \quad \forall m_k \in lf(m,t); \tag{10}$$

- Each frame identifier is used only once in a bus cycle:

$$\sum_{k=1 \dots n} x_{ijk} \le 1, \quad \forall i, j; \tag{11}$$

- Each message $m_k \in lf(m,t)$ is transmitted using its frame identifier:

$$x_{ijk} \le Frame_{jk}, \quad \forall i, j, k, \tag{12}$$

  where $Frame_{jk}$ is a binary constant with value 1 if message $m_k \in lf(m,t)$ has a frame identifier $FrameID_{m_k} = j$ (otherwise, $Frame_{jk}$ is 0).

Finally, we have to enforce that in every cycle i no message m_k will start transmission after its associated $pLatestTx_{m_k}$. If we have $x_{ijk} = 1$, then we have to add the condition that the total amount of transmission that takes place before DYN slot j has to finish no later than $pLatestTx_k$:

$$\sum_{\substack{m_q \in lf(m,t) \\ p=1\ldots j-1}} (x_{ipq} \times C_q + (1 - x_{ipq}) \times gdMinislot) \le pLatestTx_k \times gdMinislot. \quad (13)$$

The conditions (8–13) together with the maximization goal expressed in (7) define the ILP program that will determine the maximum worst-case number of bus cycles that can be filled with elements in $lf(m,t)$ and $ms(m,t)$. By adding this result to the value determined in (6), we obtain the total number $BusCycles_m(t)$ (5).

5.1.2 Optimal solution for w'_m

In the worst case, the elements in $lf(m,t)$ and $ms(m,t)$ will delay the message under analysis for $BusCycles_m(lf(m,t), ms(m,t))$ bus periods. In addition, they will delay the actual transmission of m during the DYN segment of the bus period $BusCycles_m + 1$, by an amount w'_m.

The problem of determining the value for w'_m is defined as follows: given the multisets $lf(m,t)$ and $ms(m,t)$ and the maximum number $BusCycles_m(lf(m,t), ms(m,t))$ that they can fill, what is the maximum possible load (8) in the first unfilled bus cycle (i.e. the bus cycle that does not satisfy the condition in (9)).

In order to determine the exact value of w'_m in the worst case, one can use the same ILP system defined in the previous section for computing $BusCycles_m(lf(m,t),$ $ms(m,t))$, with the following modifications:

- Since we know the value $BusCycles_m$ (which is determined solving the ILP formulation presented in the previous section), we add conditions that force the values $y_i = 1$ for all $i = 1\ldots BusCycles_m$, and $y_i = 0$ for all $i = BusCycles_m + 1\ldots n$; in this way, the messages will be packed so that the bus cycles from 1 to $BusCycles_m$ will be filled (i.e. they satisfy condition (9)), while the remaining bus cycles will be unfilled.
- Using the same set of conditions (8–13) for filling the first $BusCycles_m$ cycles, the goal described in (7) is replaced with the following one, expressing that the load of the cycle number $BusCycles_m + 1$ has to be maximized ($Load_L$ is expressed as in (8)):

$$maximize\ Load_L, \text{ for } L = BusCycles_m + 1. \quad (14)$$

5.1.3 Heuristic solution for $BusCycles_m$

In the previous subsections, we have presented solutions for determining the optimal values for $BusCycles_m$ and w'_m. As we will see later in Sect. 5.4, such solutions are unacceptable in practice due to their long computation times inherent to such high complexity algorithms. For this reason, we propose heuristic solutions, with lower complexity, that need extremely short computation times, while at the same time producing results close to the ones offered by the optimal implementations.

We start by presenting a heuristic solution for computing the value $BusCycles_m$. We first make the observation that in a bus cycle where a message m is sent by a node N_p during DYN slot $FrameID_m$, in the worst case there will be at most $FrameID_m - 1$ unused minislots before m is transmitted (in Fig. 5a, the transmission of m_2 can be preceded by at most one unused minislot).

Instead of considering the multiset $ms(m, t)$ to calculate the actual number of unused minislots before message m, as we did for the exact solution, we will consider the worst-case number of minislots. The delay produced by the minislots will be considered as part of the message communication time as follows (see also (1)):

$$C'_m = (FrameID_m - 1) \times gdMinislot + C_m. \tag{15}$$

Since the duration of one minislot (*gdMinislot*) is an order of magnitude smaller compared to the length of a cycle, this approximation will not introduce any significant pessimism.

The problem left to solve now is how many bus cycles can be filled with the elements from a multiset $lf'(m, t)$, that consists of all the messages in $lf(m, t)$ for which we consider the communication times computed using (15).

If we ignore the conditions expressed in (11–13), then determining $BusCycles_m(lf'(m, t))$ becomes a *bin covering* problem (Labbe et al. 1995). Bin covering tries to maximize the number of bins that can be filled to a fixed minimum capacity using a given set of items with specified weights. In our scenario, the messages in $lf'(m, t)$ are the items, the dynamic segments of the bus cycles are bins, and $pLatestT_{N_p} \times gdMinislot$ is the minimum capacity required to fill a bin. The bin-covering problem is NP-hard in the strong sense (Labbe et al. 1995), and our solution is to determine an upper bound, using the approach presented in Labbe et al. (1995), on the number of maximum bins that can be covered. The upper bounds proposed in Labbe et al. (1995) are of polynomial complexity and lead to very good quality results (see Appendix).

Note that, ignoring the conditions from (11–13) and determining an upper bound for bin-covering can only lead to an increase in the number of bus cycles compared to the exact solution. Experiments will show the impact of the heuristic on the pessimism of the analysis.

5.1.4 Heuristic solution for w'_m

A straightforward heuristic to the computation of w'_m stems from the observation that, in a hypothetical worst-case scenario, message m could be sent in the last possible moment of the current bus cycle, which means that

$$w'_m = ST_{bus} + pLatestTx_{N_p} \times gdMinislot, \tag{16}$$

where ST_{bus} is the length of the ST segment of a bus cycle.

5.2 Holistic schedulability analysis of FPS tasks and DYN messages

As mentioned in Sect. 2, the worst-case response times of FPS tasks are influenced on one hand by higher priority FPS tasks, and on the other hand by SCS tasks. The worst-case response time R_{ij} of a FPS task τ_{ij} is determined as presented in Palencia and Gonzaléz Harbour (1998), and in Pop et al. (2003) we have shown how to take into consideration the interference on R_{ij} produced by an existing static schedule. What is important to mention is that R_{ij} depends on jitters of the higher priority tasks and predecessors of τ_{ij}. This means that for all such activities we have to compute the jitter. In the rest of this section we will only concentrate on the situation when the jitter of a task depends on the arrival time of a message.

According to the analysis of multiprocessor and distributed systems presented in Palencia and Gonzaléz Harbour (1998), the jitter for a task τ_r that starts execution only after it receives a message m depends on the values of the best-case and worst-case transmission times of that message:

$$J_{\tau_r} = R_m - R_m^b. \tag{17}$$

The calculation of the worst-case transmission time R_m of a DYN message m was presented in Sect. 5.1. For computing R_m^b we have to identify the best-case scenario of transmitting message m. Such a situation appears when the message becomes ready immediately before the DYN slot with $FrameID_m$ starts, and it is sent during that bus cycle without experiencing any delay from higher priority messages. Thus, the equation for the best-case transmission time of a message is:

$$R_m^b = C_m, \tag{18}$$

where C_m is the time needed to send the message m.

We notice from (17) that the jitters for activities in the system depend on the values of the worst case response times, which in turn depend on the values of the jitters (Pop et al. 2005b). Such a recursive system is solved using a fixed point iteration algorithm in which the initial values for jitters are 0.

Let us make a final remark. According to (Palencia and Gonzaléz Harbour 1998), the worst-case response time calculation of FPS tasks is of exponential complexity and the approach proposed in Palencia and Gonzaléz Harbour (1998) and also used in Pop et al. (2003) is a heuristic with a certain degree of pessimism. The pessimism of the response times calculated by our holistic analysis will, of course, also depend on the quality of the solution for the delay induced by the DYN messages transmitted over FlexRay. The calculation of this delay is our main concern in this paper. Therefore, when we speak about optimal and heuristic solutions in this paper we refer to the approach used for calculating the $BusCycles_m$ and w'_m (used in the worst-case response times calculation for DYN messages) and not the holistic response time analysis which is based on the heuristics in Palencia and Gonzaléz Harbour (1998), Pop et al. (2003).

5.3 Analysis for dual-channel FlexRay bus

The specification of the FlexRay protocol mentions that the bus has two communication channels (FlexRay 2005). The analysis presented above is appropriate for systems where the two channels of the FlexRay bus are used in a redundant manner, transporting the same information simultaneously in order to support fault-tolerance.

In order to increase the bandwidth of the bus, one can use the two channels independently, so that different sets of messages are sent over each of the channels during a bus cycle. In this section we extend our previous analysis in order to compute the worst case response times for messages transmitted in such systems.

First, we extend our system model (Fig. 1a) and consider that all nodes in the system have access to a dual-channel FlexRay bus. As a consequence, in the application model each message m is associated a pair $\langle FrameID_m, Channel_m\rangle$, with the meaning that message m is sent during $FrameID_m$ on $Channel_m$ (where $Channel_m = \{A, B\}$).

Second, we notice that the transmission of a message can be delayed only by messages that are transmitted on the same channel. As a consequence, the only modification in the analysis presented in Sect. 5 is the definition of the sets $lf(m)$ and $hp(m)$, which contain only those messages that are transmitted on $Channel_m$:

- $hp(m)$ becomes now the set of local messages with higher priority, that use the same frame identifier *and* the same channel as m.
- $lf(m)$ contains any messages in the system that can use $Channel_m$ and DYN slots with lower frame identifiers than the one used by m.

5.4 Evaluation of analysis algorithms

We were interested to determine the quality of the proposed analysis approaches, and how well they scale with the number of FlexRay messages that have to be analyzed. All the experiments were run on P4 machines using 2 GB RAM. The ILP-based solutions have been implemented using the CPLEX 9.1.2 ILP solver.[7]

We have generated synthetic applications of 20, 30, 40 and 50 tasks mapped on architectures consisting of 2, 3, 4, and 5 nodes, respectively. Fifteen applications were generated for each of these four cases. The number of time-critical FlexRay messages were 30, 60, 90, and 120 for each case, respectively. Out of these, 10, 20, 30, and 40 messages were time-critical DYN messages that were analyzed using the approaches presented in Sect. 5. We have considered a bus period of $1/100$ of the largest application period in the system, with 15% bandwidth allocated to the DYN segment. We have randomly assigned transmission times C_m to DYN messages, so that the time needed to send such a message was between $1/10$ and $1/3$ of the DYN segment length. We have also randomly allocated DYN *FrameID*s to nodes and messages. Each such application has been analyzed using four holistic analysis approaches, depending on the approach used for the calculation of the components $BusCycles_m$ and w'_m of the worst-case response time R_m for a DYN message:

[7] http://www.ilog.com/products/cplex

Holistic analysis	$BusCycles_m$	w'_m
OO	**O**ptimal solution (5.1.1)	**O**ptimal solution (5.1.2)
OO$^-$	**O**ptimal solution (5.1.1)	ILP from 5.1.2 with 1 min time-out (**O**$^-$)
OH	**O**ptimal solution (5.1.1)	**H**euristic solution (5.1.4)
HH	**H**euristic solution (5.1.3)	**H**euristic solution (5.1.4)

Among the solutions we proposed, OO will always provide the tightest worst-case response times. However, it is only able to produce results for up to 20 DYN messages in a reasonable time. We have noticed that the bottleneck for OO is the exact calculation of w'_m (which is a value smaller than a bus cycle), and that running the ILP from Sect. 5.1.2 using a time-out of one minute we are able to obtain near-optimal results for w'_m. We have denoted with OO$^-$ such an analysis. Since the near-optimal result for w'_m is a lower bound, OO$^-$ can lead to an incorrect (optimistic) result (i.e., the system is reported as schedulable, but in reality it might not be). Although OO$^-$ is, thus, of no practical use, it is very useful in determining, by comparison, the quality of our proposed FlexRay analysis heuristics, OH and HH.

In order to evaluate the approaches for FlexRay analysis, we have determined for an analysis approach A the average ratio:

$$\text{ratio} = \frac{1}{n} \times \sum_{m \in DYN} \frac{R_m^A}{R_m^{\text{OO}^-}}, \tag{19}$$

where A is one of the OO, OH or HH approaches and n is the number of messages in the analysed application.

This ratio captures the degree of pessimism of A compared to OO$^-$; the smaller the ratio, the less pessimistic the analysis. The results obtained with OO, OH and HH are presented in Table 1. For each application dimension, Table 1 presents the average ratio and the average execution times of the complete analysis (including all tasks and messages) in seconds. It is important to notice that, while the execution time is for the whole analysis, including all tasks and messages, the ratio is calculated only for the DYN messages, since their response time calculation is directly affected by the degree of pessimism of the various approaches proposed in the paper. The ratio calculated over all tasks and messages in the system is smaller than the ones shown in Table 1.

Table 1 Comparison of FlexRay analysis approaches

No of msgs.	30 (10 DYN)		60 (20 DYN)		90 (30 DYN)		120 (40 DYN)	
	Ratio	Exec. (s)	Ratio	Exec. (s)	Ratio	Exec. (s)	Ratio	Exec. (s)
OO	1.009	3.1	1.009	42.3	–	–	–	–
OH	1.013	1.29	1.012	14.42	1.005	57.32	1.005	367.87
HH	1.016	0.012	1.018	0.019	1.012	0.036	1.012	0.04

We can see that OO is very close to OO^-, which means that OO^- is a good comparison baseline (it is only slightly optimistic). Due to the very large execution times, we were not able to run OO for more than 20 DYN messages.

Table 1 shows that OH produces very good quality results, in a reasonable time. For example, for 40 DYN messages, the analysis has finished in 367.87 seconds on average, and the average ratio is only 1.005.

Another result from Table 1 concerns the HH heuristic. Although HH is slightly more pessimistic than OH (for example, the DYN response times determined with HH were 1.012 times larger, on average, than those of OO^- for applications with 30 messages, compared to 1.005 for OH), it is also significantly faster. We have successfully analyzed with HH large applications, with over 100 DYN messages in 0.16 seconds on average. Thus, HH is also suitable for design space exploration, where a potentially huge number of design alternatives have to be analyzed in a very short time.

We have run a set of experiments with 15 applications of 40 tasks and 25 dynamic messages mapped on an architecture consisting of two nodes, and varied the number of frame identifiers per processor. Figure 7 presents the ratio for HH calculated according to (19) as we vary the number of frame identifiers per processor from 2 to 6. We can see that the quality of the heuristic improves as the number of frame IDs increases (and, consequently, the number of messages sharing the same *FrameID* decreases). The more messages are sharing a *FrameID*, the more important conditions (11–13) are to the quality of the result, because they restrict the way bins can be covered (e.g., messages sharing the same *FrameID* should not be packed in the same bin). However, even for a small number of frame IDs HH produces good quality results (e.g., for two frame IDs, HH's ratio is 1.1226).

We also considered a real-life example implementing a vehicle cruise controller that consists of 54 tasks mapped over 5 nodes, resulting in 26 DYN messages. We considered that 10 percent of the FlexRay communication cycle is allocated to the DYN segment communication. Scheduling the system using the OO approach took 0.19 seconds. Using the OH approach took 0.08 s, while the HH alternative was the fastest, finishing the analysis in 0.002 s. The average ratio of OH relative to OO is 1.003, while the average ratio of HH relative to OO is 1.004, which means that the heuristics obtained results almost identical to the optimal approach OO.

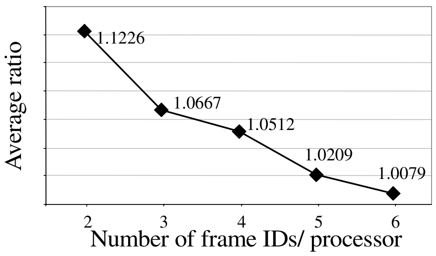


Fig. 7 Quality of HH

6 Bus access optimisation

The design of a FlexRay bus configuration for a given system consists of a collection of solutions for the following subproblems: (1) determine the length of an ST slot, (2) the number of ST slots, and (3) their assignment to nodes; (4) determine the length of the DYN segment, (5) assign DYN slots to nodes, and (6) *FrameID*s to DYN messages.

The choice of a particular bus configuration is extremely important when designing a specific system, since its characteristics heavily influence the global timing properties of the application.

For example, notice in Fig. 8 how the structure of the ST segment influences the response time of message m_3 (for this example we ignored the DYN segment). The figure considers a system with two nodes, N_1 that sends message m_1 and N_2 that sends messages m_2 and m_3. The message sizes are depicted in the figure. In the first scenario, the ST segment consists of two slots, $slot_1$ used by N_1 and $slot_2$ used by N_2. In this situation, message m_3 can be scheduled only during the second bus cycle, with a response time of 16. If the ST segment consists of 3 slots (Fig. 8b), with N_2 being allocated $slot_2$ and $slot_3$, then N_2 is able to send both its messages during the first bus cycle. The configuration in Fig. 8c consists of only two slots, like in Fig. 8a. However, in this case the slots are longer, such that several messages can be transmitted during the same frame, producing a faster response time for m_3 (one should notice, however, that by extending the size of the ST slots we delay the reception of message m_1 and m_2).

Similar optimisations can be performed with regard to the DYN segment. Let us consider the example in Fig. 9, where we have two nodes N_1 and N_2. Node N_1 is transmitting messages m_1 and m_3, while N_2 sends m_2. Figure 9 depicts three configuration scenarios, a–c. Table A depicts the frame identifiers for the scenario in Fig. 9a, while Table B corresponds to Fig. 9b–c. The length of the ST slot has been set to 8. In Fig. 9a, the length of the DYN segment is not able to accommodate both m_1 and m_2, thus m_2 will be sent during the second bus cycle, after the transmission of m_3 ends. Figure 9b and Fig. 9c depict the same system but with a different allocation

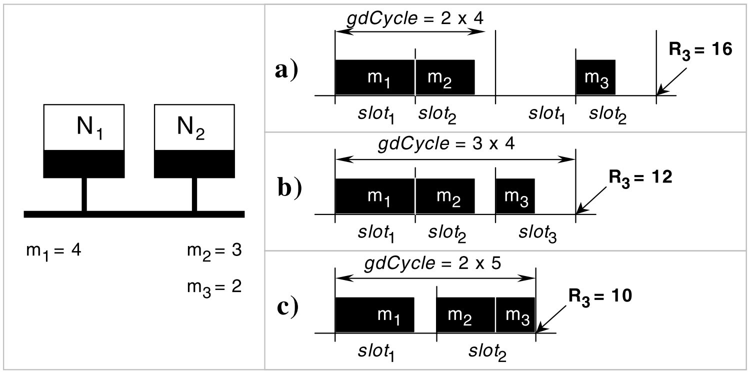


Fig. 8 Optimisation of the ST segment

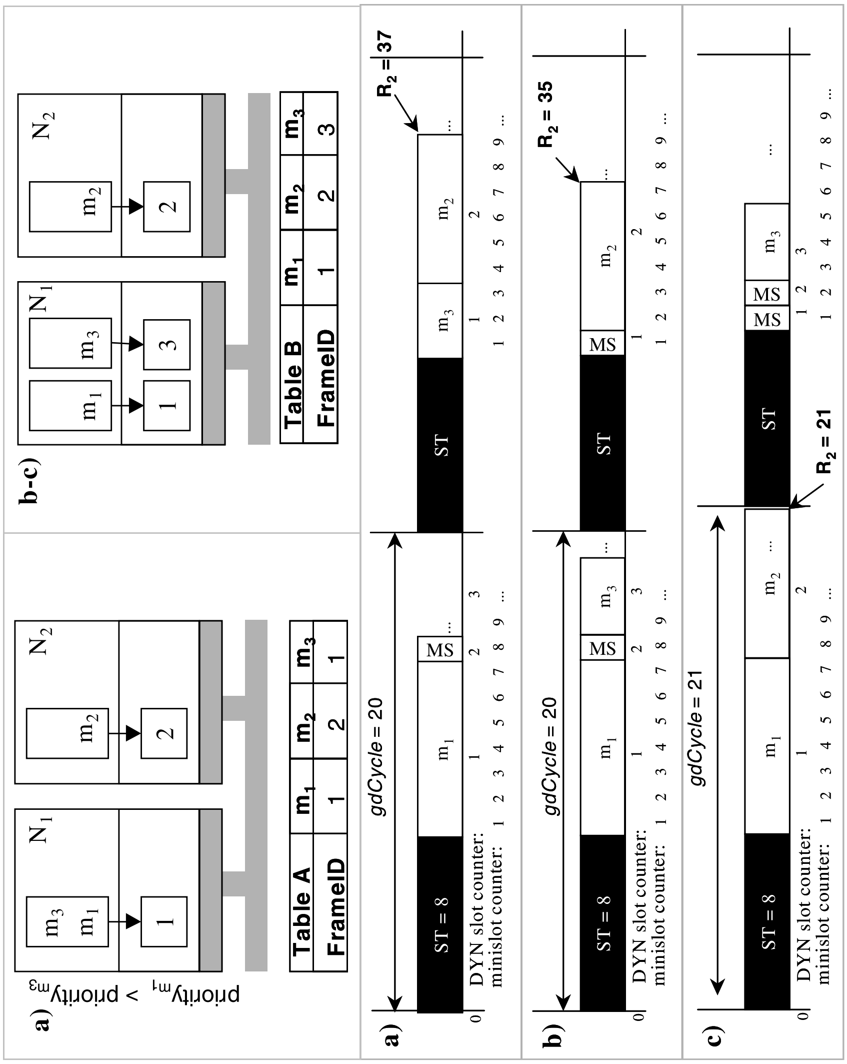


Fig. 9 Optimisation of the DYN segment

of DYN slots to messages (Table B). In Fig. 9b we notice that m_3, which now does not share the same frame identifier with m_1, can be sent during the first bus cycle, thus m_2 will be transmitted earlier during the second cycle. Moreover, if we enlarge the size of the DYN segment as in Fig. 9c, then the worst-case response time of m_2 will considerably decrease since it will be sent during the first bus cycle (notice that in this case m_3, having a greater frame identifier than that of m_2, will be sent only during the second cycle).

In order to illustrate the importance of choosing the right bus configuration, we present three approaches for optimising the bus access such that the schedulability of the system is improved. The first approach builds a relatively straightforward, basic, bus configuration. The other two approaches perform optimization over the basic configuration.

6.1 The basic bus configuration

In this section we construct a basic bus configuration (BBC) which is based on analysing the minimal bandwidth requirements imposed by the application.

The BBC algorithm is presented in Fig. 10 and it starts by setting the number of ST slots in a bus cycle. The length T_{bus} of the bus cycle is captured by the *gdCycle* protocol parameter. Since each node in the system that generates ST messages needs at least one ST slot, the minimum number of ST slots is $nodes_{ST}$, the number of nodes that send ST messages (line 1). The protocol specification also imposes a minimum limit on the number of ST slots, therefore even if there are no nodes in the system that are using the ST segment, there should be at least two ST slots during a bus cycle. Next, the size of an ST slot is set so that it can accommodate the largest ST message in the system (line 2). In line 4, the configuration of the ST segment is completed by assigning in a round robin fashion one ST slot to each node that requires one (i.e. in a system with four nodes, where each node is sending in the static segment, the ST segment of the bus cycle will contain four slots; node 1 will use slot 1, node 2 will use ST slot 2, etc.).

When it comes to determining the size of the DYN segment, one has to take into consideration the fact that the period of the bus cycle (*gdCycle*) has to be an integer

```
1  $gdNumberOfStaticSlots = \max(2, nodes_{ST})$
2  $gdStaticSlot = \max(C_m)$, $m$ is an ST message
3  $ST_{bus} = gdNumberOfStaticSlots * gdStaticSlot$
4  assign one ST slot to each node (round robin)
5  for $n = 1$ to 64 do
6    $gdCycle = T_{SS}/n$
7    if $gdCycle < 16000$ µs then
8      $DYN_{bus} = gdCycle - ST_{bus}$
9      Assign FrameIDs to DYN messages
10     GlobalSchedulingAlgorithm()
11     Compute cost function $Cost$
12     if $Cost < BestCost$ then save current solution
13   end if
14 end for
```

Fig. 10 Basic bus configuration

divisor[8] of the period of the global static schedule (T_{ss}). In addition, the FlexRay protocol specifies that each node implementing a cyclic schedule maintains in the communication controller a counter *vCycleCounter* that has values in the interval 0..63. This means that during a period of the static schedule there can be at most 64 bus cycles, which leads us to the conclusion that the value of *gdCycle* can be determined by iterating over all possible values for *vCycleCounter* (lines 5–14) and choosing the most favourable solution in terms of system schedulability (line 11). Line 7 introduces a restriction imposed by the FlexRay specification, which limits the maximum bus cycle length to 16 ms. Once the length of the bus cycle is set (line 5), knowing the length ST_{bus} of the ST segment (line 3), we can determine the length DYN_{bus} of the DYN segment (line 8).

At this point, in order to finish the design of the bus configuration, a *FrameID* has to be assigned to each of the DYN messages (and implicitly DYN slots are assigned to the nodes that generate the message). This assignment (line 9) is performed under the following guidelines:

- Each DYN message receives an unique *FrameID*; this is recommended in order to avoid large delays introduced by $hp(m)$ in (5). For example, in Fig. 9, we notice that message m_3 has to wait for an entire *gdCycle* when it shares a frame identifier with the higher priority message m_1(Fig. 9a), which is not the case when it has its own *FrameID* (Fig. 9b).
- DYN messages with a higher criticality receive smaller *FrameID*s.; this is required in order to reduce, for a given message, the delay produced by $lf(m)$ and $ms(m)$ in (5). We capture the criticality of a message m as:

  $$CP_m = D_m - LP_m, \tag{20}$$

  where D_m is the deadline of the message and LP_m is the longest path in the task graph from the root to the node representing the communication of message m. A small value of CP_m (higher criticality) indicates that the message should be assigned a smaller *FrameID*.

Once we have defined the structure of the bus cycle, we can analyse the entire system (line 9) by performing the global static scheduling and analysis described in Sect. 5. The resulted system is then evaluated using a cost function that captures the schedulability degree of the system (line 10):

$$Cost = \begin{cases} f_1 = \sum_{\tau_{ij}} \max(R_{ij} - D_{ij}, 0), & \textit{if } f_1 > 0, \\ f_2 = \sum_{\tau_{ij}} (R_{ij} - D_{ij}), & \textit{if } f_1 = 0, \end{cases} \tag{21}$$

where R_{ij} and D_{ij} are the worst case response times and respectively the deadlines for all the activities τ_{ij} in the system. This function is strict positive if at least one task or message in the system misses its deadline, and negative if the whole system is schedulable. Its value is used in line 11 when deciding whether the current configuration is the best one encountered so far.

[8] We consider that the T_{SS} parameter is slightly adjusted, if necessary.

6.2 Greedy heuristic

The Basic Bus Configuration (BBC) generated as in the previous section can result in an unschedulable system (the cost function in (21) is positive). In this case, additional points in the solution space have to be explored. In Fig. 11 we present a greedy heuristic that further explores the design space in order to find a schedulable solution.

While for the BBC the number and size of ST slots has been set to the minimum ($gdNumberOfStaticSlots_{min} = \max(2, nodes)$, $gdStaticSlot_{min} = \max(C_m)$), the heuristic explores different alternative values between these minimal values and the maxima imposed by the protocol specification (FlexRay 2005). Thus, during a bus cycle there can be at most $gdNumberOfStaticSlots_{max} = 1023$ ST slots, while the size of a ST slot can take at most $gdStaticSlot_{max} = 661$ macroticks. In addition, the payload for a FlexRay frame can increase only in 2-byte increments, which according to the FlexRay specification translates into 20 *gdBit*, where *gdBit* is the time needed for transmitting one bit over the bus (line 2).

The assignment of ST slots (line 3) to nodes is performed, like for the BBC, in a round robin fashion, with the difference that each node can have not only one but a quota of ST slots determined by the ratio of ST messages that it transmits (i.e. a node that sends more ST messages will be allocated more ST slots).

The sizes of the bus cycle and of the DYN segment are assigned in lines 4–16 in a similar way to the BBC algorithm.

However, while for the BBC the allocation of *FrameID*s to DYN messages is based on the estimated criticality (20), here we explore several *FrameID* assignment alternatives inside the loop in lines 8–14. We start from an initial assignment as in the BBC after which a global scheduling is performed (line 10). Using the resulted response times, in the next iteration we assign smaller *FrameID*s with priority to those DYN messages m that have a smaller value for $D_m - R_m$, where D_m is the deadline and R_m is the worst case response time computed by the global scheduling.

```
1  for $gdNumberOfStaticSlots = gdNumberOfStaticSlots_{min}$ to
   $gdNumberOfStaticSlots_{max}$ do
2    for $gdStaticSlot = gdStaticSlot_{min}$ to $gdStaticSlot_{max}$ step 20 *
     gdBit do
3      Assign ST slots to nodes
4      for $n = 1$ to 64 do
5        $gdCycle = T_{ss}/n$
6        if $gdCycle < 16000$ μs then
7          $DYN_{bus} = gdCycle - ST_{bus}$
8          do
9            Assign FrameIDs to DYN messages
10           GlobalSchedulingAlgorithm()
11           For all DYN messages, compute $CP_i$
12           Compute cost function $Cost$
13           if $Cost < BestCost$ then save current solution
14         while($BestCost$ unchanged for $max\_iterations$);
15       end if
16     end for
17   end for
18 end for
```

Fig. 11 Greedy heuristic

6.3 Simulated annealing based approach

We have implemented a more exhaustive design space exploration than the one in Sect. 6.2, using a simulated annealing (Kirkpatrick et al. 1983) approach. While relatively time consuming, this heuristic can be applied if both the BBC and the configuration produced by the greedy approach are unschedulable. Starting from the solution produced by the greedy optimisation, the SA based heuristic explores the design space performing the following set of moves:

- *gdNumberOfStaticSlots* is incremented or decremented, inside the allowed limits (when an ST slot is added, it is allocated randomly to a node);
- *gdStaticSlot* is increased or decreased with $20 \times gdBit$, inside the allowed limits;
- The assignment of ST slots to nodes is changed by re-assigning a randomly selected ST slot from a node N_1 to another node N_2. We also use in this context a similar transformation that switches the allocation of two ST slots, $FrameID_1$ and $FrameID_2$, used by two nodes N_1 and N_2 respectively;
- The assignment of DYN slots to messages is modified by switching the slots used by two DYN messages.

In Sect. 6.4 we used extensive, time consuming runs with the Simulated Annealing approach, in order to produce a reference point for the evaluation of our greedy heuristic.

6.4 Evaluation of bus optimisation heuristics

In order to evaluate our optimisation algorithms we generated seven sets of 25 applications representing systems of 2 to 7 nodes respectively. We considered 10 tasks mapped on each node, leading to applications with a number of 20 to 70 tasks. Depending on the mapping of tasks, each such system had up to 60 additional nodes in the application task graph due to the communication tasks. The tasks were grouped in task graphs of 5 tasks each. Half of the tasks in each system were time triggered and half were event triggered. The execution times were generated in such a way that the utilisation on each node was between 30% and 60% (similarly, the message transmission times were generated so that the bus utilisation was between 10% and 70%). All experiments were run on an AMD Athlon 2400+ PC.

Figure 12 shows the results obtained after running our three algorithms proposed in Sect. 6 (BBC—Basic Bus Configuration, GH—Greedy Heuristic, and SA—Simulated Annealing). In Fig. 12a we show the percentage of schedulable applications, while in Fig. 12b we present the computation times required by each algorithm. One can notice that the BBC approach runs in almost zero time, but it fails to find any schedulable configurations for systems with more than 4 processors. On the other hand, the other two approaches continue to find schedulable solutions even for larger systems. Moreover, the percentage of schedulable solutions found by the greedy algorithm is comparable with the one obtained with the simulated annealing. Moreover, the computation time required by the greedy heuristic is several orders of magnitude smaller than the one needed for the extensive runs of simulated annealing.[9]

[9]Due to the extensive runs with SA, we can assume that the actual percentage of schedulable applications is close to that found by SA.

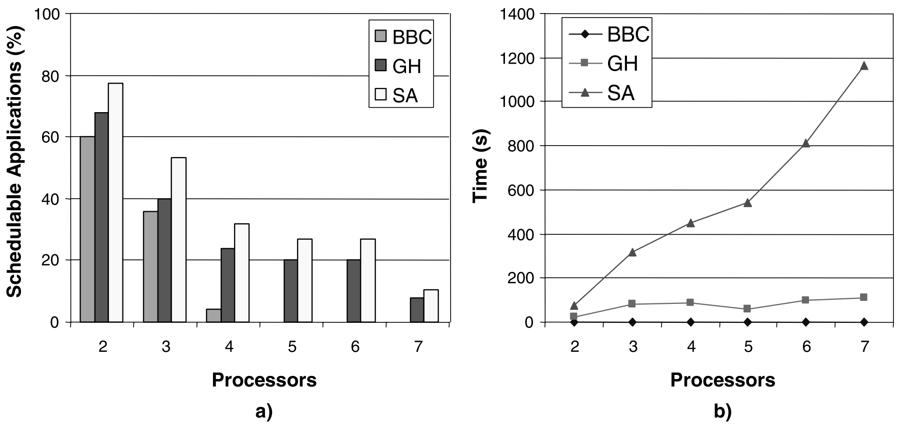


Fig. 12 Evaluation of bus optimisation algorithms

Finally, we considered the same real-life example implementing a vehicle cruise controller as in Sect. 5.4. It consists of 54 tasks and 26 messages grouped in 4 task graphs that are mapped over 5 nodes. Two of the task graphs were time triggered and the other two were event triggered. Configuring the system using the BBC approach took less than 0.001 seconds but resulted in a unschedulable system. Using the greedy heuristic approach took 0.02 seconds, while the simulated annealing was allowed to run for more than one hour; the cost function obtained by the latter was 4% smaller (meaning that the response times were also smaller) than in the solution obtained with the greedy heuristic, but in both cases the selected bus configuration resulted in a schedulable system.

7 Conclusions

In this paper, we have presented a schedulability analysis for the FlexRay communication protocol. For ST messages we have built a static cyclic schedule, while for DYN messages we have, for the first time, developed a worst-case response time analysis. This analysis has been integrated in the context of a holistic schedulability analysis that determines the timing properties for all the tasks and messages in the system. Since FlexRay is rapidly becoming one of the preferred protocols for automotive applications, the development of such an analysis is of huge importance.

We have proposed three approaches for the derivation of worst-case response times of DYN messages. OO uses an ILP formulation to derive the optimal solution for the communication delay. HH uses heuristic-based upper-bounds for a bin-covering problem in order to quickly determine good quality response times. OH is able to further reduce the pessimism of HH by using an ILP formulation for one part of the solution. Our experiments have shown that the HH approach is efficiently producing high quality results.

Finally, we have shown the importance of finding a bus configuration that is dedicated to the particular needs of the application, and have also proposed heuristics that are able to generate such a configuration. Experiments have shown that the proposed heuristics are able to find bus access parameters that are well adapted to the specific requirements of the applications, and thus rendering a potentially unschedulable solution schedulable.

Appendix

In this appendix we briefly present the bin covering heuristics that we used in our timing analysis. These heuristics are presented in more detail in Labbe et al. (1995).

The bin covering problem considers a set of n items of weights $w_1 \dots w_n$ and an unlimited number of bins of infinite capacity. The target is to fill as many bins as possible with a minimum capacity C_{min}, using the n given items.

The heuristics presented below determine upper bounds for a given instance of the bin covering problem. All the following operations assume that the items are sorted in decreasing order of their weights: $w_1 \geq w_2 \geq \cdots \geq w_n$.

Before computing the upper bounds, the list of items is first processed based on the following reduction criteria:

1. Any item with $w_j > C_{min}$ can be assigned alone to a bin. We denote with R_1 the number of such items and we eliminate them from the list of items.
2. If two items k and l satisfy the condition $w_k + w_l = C_{min}$ then there exists an optimal solution in which a bin contains only items k and l. We denote with R_2 the number of bins that can be filled in this way and we remove from the list the items that satisfy this reduction criterion.
3. Let k be the maximum index such that $w_1 + \sum_{j=k}^{n} w_j \geq C_{min}$. If $w_1 + w_k \geq C_{min}$ then there exists an optimal solution in which a bin contains only items 1 and k. The number of bins that can be filled in this way is denoted with R_3 and the items that fill these bins according to the 3rd criterion are removed from the list.

In a second step, we use the n' remaining items to fill bins with at least C_{min}. The heuristics presented bellow will determine upper bounds for the maximum value we are looking for.

1. Since no two remaining items can fill a bin up to C_{min}, then $U_0 = \lfloor \frac{n'}{2} \rfloor$.
2. The continuous relaxation of the problem gives another bound: $U_1 = \lfloor \sum_{j=1}^{n'} \frac{w_j}{C_{min}} \rfloor$.
3. Let $t = \min\{s\colon \sum_{h=s}^{n'} w_h < C_{min}\}$ and define $p(j) = \min\{p\colon \sum_{h=j}^{j+p} w_h \geq C_{min}\}$ for $j = 1, \dots, \tau = \min(\lfloor \frac{n'}{2} \rfloor, t-1)$. Then, for $k = 0, 1, \dots, \tau, U_2(k) = k + \lfloor \sum_{j=k+1}^{n'-\alpha(k)} \frac{w_j}{C_{min}} \rfloor$, where $\alpha(0)$ and $\alpha(k) = \sum_{j=1}^{k} p(j)$ for $k > 0$, is a valid upper bound for the bin covering problem that considers the reduced set of items. We denote with U_2 the minimum of all these values: $U_2 = \min(U_2(k)), k = 1, \dots, \tau$.
4. Let $\beta(j)$ be the smallest index k such that $w_j + \sum_{k=1,k\neq j}^{\beta(j)} w_k \geq C_{min}$ and define $q(j) = \beta(j)$ if $\beta(j) > j$, $q(j) = \beta(j) + 1$ otherwise. Then $U_3 = \lfloor \sum_{j=1}^{n'} \frac{1}{q(j)} \rfloor$ is a valid upper bound for the bin covering problem using the reduced set of items.

The upper bound we are looking for is determined as the minimum of the four values U_0, U_1, U_2, U_3: $U = \min(U_0, U_1, U_2, U_3)$.

This result can be further improved as follows: given an upper bound value U, if $U > \lfloor \frac{\sum_{j=1}^{n'} w_j - (3U - n')}{C_{min}} \rfloor$ then $U - 1$ is a valid upper bound value.

Considering now the initial set of items, before the reductions in the first step, the upper bound *UB* of the maximum number of bins that can be filled with C_{min} is: $UB = R_1 + R_2 + R_3 + U$.

References

Agrawal G, Chen B, Zhao W, Davari S (1994) Guaranteeing synchronous message deadlines with the token medium access control protocol. IEEE Trans Comput 43(3):327–339

Berwanger J, Peller M, Griessbach R (2000) A new high performance data bus system for safety-related applications. http://www.byteflight.de

R Bosch GmbH (1991) CAN Specification Version 2.0

Cena G, Valenzano A (2004) Performance analysis of Byteflight networks. In: Proceedings of the IEEE International Workshop on Factory Communication Systems, pp 157–166

Coffman EG Jr, Graham RL (1972) Optimal scheduling for two processor systems. Acta Inform 1:200–203

Ding S, Murakami N, Tomiyama H, Takada H (2005) A GA-based scheduling method for FlexRay systems. In: Proceedings of EMSOFT

Labbe M, Laporte G, Martello S (1995) An exact algorithm for the dual bin packing problem. Oper Res Lett 17:9–18

Echelon (2005) LonWorks: The LonTalk protocol specification. http://www.echelon.com

Ermedahl H, Hansson H, Sjödin M (1997) Response-time guarantees in ATM networks. In: Proceedings of the IEEE Real-Time Systems Symposium, pp 274–284

FlexRay homepage (2005) http://www.flexray-group.com

Hamann A, Ernst R (2005) TDMA time slot and turn optimization with evolutionary search techniques. In: Proceedings of the Design, Automation and Test in Europe Conference, vol 1, pp 312–317

Hoyme K, Driscoll K (1992) SAFEbus. IEEE Aerosp Electron Syst Mag 8(3):34–39

International Organization for Standardization (2002) Road vehicles-Controller Area Network (CAN)—Part 4: Time-triggered communication. ISO/DIS 11898–4

Kirkpatrick S, Gelatt CD, Vecchi MP (1983) Optimisation by simulated annealing. Science 220:671–680

Kopetz H, Bauer G (2003) The time-triggered architecture. Proc IEEE 91(1):112–126

Local Interconnect Network Protocol Specification (2005). http://www.lin-subbus.org

Miner PS (2000) Analysis of the SPIDER fault-tolerance protocols. In: Proceedings of the 5th NASA Langley Formal Methods Workshop

Navet N, Song Y, Simont-Lion F, Wilwert C (2005) Trends in automotive communication systems. Proc IEEE 93(6):1204–1223

Palencia JC, Gonzaléz Harbour M (1998) Schedulability analysis for tasks with static and dynamic offsets. In: Proceedings of the Real-Time Systems Symposium, pp 26–38

Pop P, Eles P, Peng Z, Doboli A (2000) Scheduling with bus access optimization for distributed embedded systems. IEEE Trans VLSI Syst 8(5):472–491

Pop T, Eles P, Peng Z (2003) Schedulability analysis for distributed heterogeneous time event-triggered real-time systems. In: Proceedings of the 15th Euromicro Conference on Real-Time Systems (ECRTS 2003), pp 257–266

Pop P, Eles P, Peng Z (2004) Schedulability-driven communication synthesis for time-triggered embedded systems. Real-Time Syst J 24:297–325

Pop P, Eles P, Peng Z (2005a) Schedulability-driven frame packing for multi-cluster distributed embedded systems. ACM Trans Embed Comput Syst 4(1):112–140

Pop T, Pop P, Eles P, Peng Z (2005b) Optimization of hierarchically scheduled heterogeneous embedded systems. In: Proceedings of 11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, pp 67–71

Profibus International (2005) PROFIBUS DP Specification. http://www.profibus.com

Rushby J (2001) Bus architectures for safety-critical embedded systems. Lecture notes in computer science, vol 2211. Springer, Berlin, pp 306–323

SAE Vehicle Network for Multiplexing and Data Communications Standards Committee (1994) SAE J1850 Standard

Strosnider JK, Marchok TE (1989) Responsive, deterministic IEEE 802 5 token ring scheduling. J Real-Time Syst 1(2):133–158

Tindell K, Clark J (1994) Holistic schedulability analysis for distributed hard real-time systems, Microprocess. Microprogram. 50(2–3)

Tindell K, Burns A, Wellings A (1995) Calculating CAN message response times. Control Eng Pract 3(8):1163–1169

Traian Pop has received his B.Sc. degree in Computer Science and Engineering from "Politehnica" University of Timisoara, Romania, in 1999, and the Licentiate of Engineering and Ph.D. degrees in Computer Science from Linköping University, Sweden, in 2003 and 2007, respectively.

His research interests are in the area of timing analysis and design optimisation of embedded systems.

Traian Pop was co-recipient of the best presentation award at the IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS 2003), and of the best paper award at the Real-Time in Sweden Conference (RTiS 2007). He was also nominated for the best paper award at the Design Automation Conference (DAC 2001).

Currently, he is a guest researcher at Linköping University, Sweden.

Paul Pop is an associate professor at the Informatics and Mathematical Modelling Dept., Technical University of Denmark. He has received his Ph.D. in Computer Systems from Linköping University, Sweden, in 2003.

He is active in the area of analysis and design of real-time embedded systems, where he has published extensively and co-authored several book chapters and one book.

Paul Pop received the best paper award at the Design, Automation and Test in Europe Conference (DATE 2005) and at the Real-Time in Sweden Conference (RTiS 2007) and was nominated for the best paper award at the Design Automation Conference (DAC 2001).

He is currently involved in the ARTIST2 (Advanced Real-Time Systems Information Society Technologies) Network of Excellence on embedded systems design.

Petru Eles received the Ph.D. degree in computer science from the Politehnica University of Bucharest, Romania, in 1993. He is currently a professor with the Department of Computer and Information Science at Linköping University, Sweden. His research interests include embedded systems design, hardware–software codesign, real-time systems, system specification and testing, and CAD for digital systems. He has published extensively in these areas and coauthored several books, such as "System Synthesis with VHDL" (Kluwer Academic, 1997), "System-Level Design Techniques for Energy-Efficient Embedded Systems" (Kluwer Academic, 2003), "Analysis and Synthesis of Distributed Real-Time Embedded Systems" (Kluwer Academic, 2004), and "Real-Time Applications with Stochastic Task Execution Times: Analysis and Optimisation" (Springer, 2006). He was a corecipient of the Best Paper Awards at the European Design Automation Conference in 1992 and 1994, and at the Design Automation and Test in Europe Conference in 2005, and of the Best Presentation Award at the 2003 IEEE/ACM/IFIP International Conference on Hardware/ Software Codesign and System Synthesis.

Petru Eles is an Associate Editor of the IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, and of the IEE Proceedings—Computers and Digital Techniques. He has served as a Program Committee member for numerous international conferences in the areas of Design Automation,

Embedded Systems, and Real-Time Systems, and as a TPC chair and General chair of the IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis.
Petru Eles has served as an IEEE CAS Distinguished Lecturer for 2004 and 2005. He is a member of the IEEE and of the ACM.

Zebo Peng received the B.Sc. degree in Computer Engineering from the South China Institute of Technology, China, in 1982, and the Licentiate of Engineering and Ph.D. degrees in Computer Science from Linköping University, Sweden, in 1985 and 1987, respectively. He is Full Professor of Computer Systems, Director of the Embedded Systems Laboratory, and Chairman of the Division for Software and Systems in the Department of Computer Science, Linköping University. He is also the Director of the National Graduate School of Computer Science in Sweden. His current research interests include design and test of embedded systems, electronic design automation, SoC testing, design for testability, hardware/software co-design, and real-time systems. He has published more than 200 technical papers, and co-authored the books "System Synthesis with VHDL" (Kluwer Academic, 1997), "Analysis and Synthesis of Distributed Real-Time Embedded Systems" (Kluwer Academic, 2004), "System-level Test and Validation of Hardware/Software Systems" (Springer, 2005), and "Real-Time Applications with Stochastic Task Execution Times" (Springer, 2007).
Prof. Peng was co-recipient of two best paper awards at the European Design Automation Conferences (1992 and 1994), a best paper award at the IEEE Asian Test Symposium (2002), a best paper award at the Design Automation and Test in Europe Conference (2005), and a best presentation award at the IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (2003). He serves currently as Associate Editor of the IEEE Transactions on VLSI Systems, VLSI Design Journal, and the EURASIP Journal on Embedded Systems. He has served as Guest Editor for the special issue on "Emerging Strategies for Resource-Constrained Testing of System Chips" in the IEE Proceedings for Computer and Digital Techniques and the special issue on "Design Methodologies and Tools for Real-Time Embedded Systems" in the Journal on Design Automation for Embedded Systems. He has served on the program committee of a dozen international conferences and workshops, including ATS, ASP-DAC, DATE, DDECS, DFT, ETS, ITSW, MEMOCDE and VLSI-SOC. He was the General Chair of the 6th IEEE European Test Workshop (ETW'01), the Program Chair of the 7th IEEE Design & Diagnostics of Electronic Circuits & Systems Workshop (DDECS'04), and the Test Track Chair of the 2006 Design Automation and Test in Europe Conference (DATE'06). He is the Program Chair of the 12th IEEE European Test Symposium (ETS'07), and the Program Chair of DATE'08. He is the Chair of the IEEE European Test Technology Technical Council (ETTTC), and has been a Golden Core Member of the IEEE Computer Society since 2005.

Alexandru Andrei received the MS degree from Politehnica University Timisoara, Romania, in 2001 and the Ph.D. degree in computer engineering from Linköping University, Sweden in 2007. His research interests include low-power design, real-time systems, and hardware–software codesign. He is currently affiliated with Ericsson AB, Sweden.