General Framework for Routing, Scheduling and Formal Timing Analysis in Deterministic Time-Aware Networks

Anaïs Finzi ✉
TTTech Computertechnik AG, Wien, Austria

Ramon Serna Oliver ✉
TTTech Computertechnik AG, Wien, Austria

Abstract

In deterministic time-aware networks, such as TTEthernet (TTE) and Time Sensitive Networking (TSN), time-triggered (TT) communication are often routed and scheduled without taking into account other critical traffic such as Rate-Constrained (RC) traffic. Consequently, the impact of a static transmission schedule for TT traffic can prevent RC traffic from fulfilling their timing constraints.

In this paper, we present a general framework for routing, scheduling and formal timing analysis (FTA) in deterministic time-aware networks (e.g. TSN, TTE). The general framework drives an iterative execution of different modules (i.e. routing, scheduling and FTA) searching for a solution that fulfills an arbitrary number of defined constraints (e.g. maximum end-to-end RC and TT latency) and optimization goals (e.g. minimize reception jitter). The result is an iteratively improved solution including the routing configuration for TT and RC flows, the static TT schedule, a formal analysis for the RC traffic, as well as any additional outputs satisfying user constraints (e.g. maximum RC jitter). We then do a performance evaluation of the general framework, with a proposed implementation of the necessary modules for TTEthernet networks with mixed time-triggered and rate-constrained traffic. The evaluation of our studied realistic use case shows that, using the general framework, the end-to-end latency for RC traffic can be reduced up to 28.3%, and the number of flows not fulfilling their deadlines divided by up to 3 compared to existing methods.

2012 ACM Subject Classification Networks → Network performance evaluation

Keywords and phrases TSN, TTEthernet, AFDX, AVB, Modeling, Routing, Scheduling, Formal timing analysis, Worst-case analysis, Performance evaluation

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2022.8

Funding This paper is part of the ADACORSA project that has received funding within the ECSEL JU in collaboration with the EU H2020 Programme and National Authorities, under grant agreement 876019.

1 Introduction

For many application domains, the temporal behavior of critical communication flows needs to be formally validated. For example, in aerospace the authorities require the proof of correctness as part of the certification process, as it also occurs in emerging industrial automation systems, with respect to critical traffic fulfilling end-to-end latency, jitter and available memory requirements. These proofs have been provided through analysis methods like Network Calculus [10, 8, 4] or the more recent Compositional performance evaluation [24], for technologies like Avionics Full DupleX (AFDX) [1], TTEthernet (TTE) [12, 20] or Time Sensitive Networking (TSN) [11].

Deterministic time-aware networks such as TSN and TTE enhance the event-triggered Rate Constrained traffic class (RC) with a fully synchronous time-triggered (TT) communication paradigm offering stringent guarantees, deterministic real-time temporal behavior, and composability. For the TT traffic class, determinism is ensured via an offline communication schedule enforcing the contention-free and timely delivery of critical frames across switched multi-hop networks within defined latency and jitter bounds.

The schedule synthesis for the TT traffic class is typically done either through heuristic-based approaches [21, 5] or optimal algorithms based on MiP or SMT solvers [19, 7, 3, 18].

The classical method considering routing, scheduling and FTA is to perform each of these steps sequentially. However, this means that manual intervention to correct non-optimal routing is often necessary, which can be very difficult. In particular, when routing is based on methods such as load balancing, where modifying one route can have repercussions on multiple flows. For this reason, approaches in which the steps computing the schedules and routes are coupled have been developed, using methods such as ILP [17, 6]. However, these approaches do not incorporate a formal timing analysis, so no alternative solutions can be easily explored if the RC constraints are not fulfilled.

To our knowledge, two previous works integrate the formal timing analysis of RC traffic when computing routes for both RC and TT traffic flows along with the schedule for TT traffic [9, 27]. However, both methods consider the end-to-end latency as the single user constraint. Additionally, we discuss in Section 2 several flaws leading to long computation times and inefficiencies that we address with our proposed method.

In this paper we provide a general framework for deterministic time-aware networks with mixed time-triggered and rate-constrained traffic classes, computing the static routing and scheduling configuration such as the constraints of both traffic classes are fulfilled. In our method, the routing, scheduling, and formal timing analysis modules can be implemented and customized by the user according to their own requirements. Moreover, additional RC or TT constraints (e.g. end-to-end latency, frame-memory limitations, etc...) can be incorporated to drive the search towards better solutions.

The main contributions of this paper are twofold, we propose: i) a general search for routing and scheduling which considers formal timing analysis of RC in Section 3; and ii) we conduct a performance evaluation with an application of the general framework for TTEthernet networks, including the time-triggered and rate-constrained traffic classes, in a realistic use case in Section 4 for which we developed a set of modules combining heuristic routing and SMT solver based scheduling with a formal RC analysis based on Network Calculus. To complement the analysis we compare our results with those of the current state-of-the-art in Section 4.4.

2 Related Work

Increasing the performance of time-triggered and event-triggered traffic has been pursued in previous works using methods to either improve the routes or the schedule instants of frames, by including event-triggered constraints into the scheduling problem formulation, or by applying a combination of these. To enhance TT traffic, the routing and scheduling can be done jointly using heuristics [25], or Integer Linear Programming (ILP) formulations [17, 6]. However, when using ILP, the complex worst-case timing analysis necessary to assess the RC constraints cannot be integrated into the ILP formulation as they are not linear.

Other approaches develop enhanced heuristics incorporating RC constraints guiding the search for a schedule. In [22, 7], the routing of flows is fixed and the computation of TT offsets is done considering the RC constraints of TTE. Results show that the schedulability of RC traffic can be significantly improved (e.g doubled in [7]) with a heuristic search.

Finally, at the time of writing this section and to the best of our knowledge, there are two published approaches [9, 27] on integrating the formal timing analysis of RC traffic with the routing and scheduling.

In [9], the proposed method is developed specifically for a TSN network with scheduled 802.1*Qbv* (i.e. TT traffic) and AVB flows (i.e. RC traffic). However, we have identified a few limitations impacting the efficiency of the approach: the routing strategy is to re-route only TT flows, without trying to re-route RC flows. Besides limiting the solution space, this re-routing strategy is computationally expensive since a TT flow also requires re-scheduling. Additionally, the scheduling strategy is to re-schedule all TT flows at every new step, which again is a very time consuming operation.

In [27], a method to optimize the routing is proposed for TTE networks, including the TT and RC traffic classes. In their, approach the authors use the RC end-to-end latencies when computing the routes for RC flows. However, the computation of the TT routes is done using load balancing independently from RC traffic properties. Moreover, when searching for a solution for the RC constraints, the TT routes are already fixed, which limits the solution space.

Additionally, we have noted that in both [9, 27], the RC constraints are limited to the end-to-end latency constraint, which does not map to typical industry requirements, usually including jitter and frame-memory restrictions (e.g. backlog).

In this paper, we introduce a generalized search framework that can be used for any deterministic time-aware network. We propose to explore the solution space by re-routing and re-scheduling one flow at a time, to avoid the expensive cost of re-scheduling all the flows after the initial schedule is computed. For our performance evaluation in Section 4, we present an implementation of all three functions for TTEthernet. In particular, our method can re-route any RC or TT flow, as well as re-schedule individual TT-flows, although it reduces as much as possible the re-scheduling operation, as it is the most expensive step. The search includes a formal analysis step including extensive RC constraints for our test-case, namely end-to-end latency, jitter, and memory occupancy.

3 General Framework description

The goal of our proposed general framework is to compute, within a configurable time interval, the best possible network configuration including routing, scheduling, and formal timing analysis (FTA), as well as optional parametrization and user-defined constraints.

The general framework implements a search algorithm leveraging the work of predefined functions for routing, scheduling, and FTA. Each of these three functions is encapsulated in a module, and can be adapted to implement existing or future solutions found in literature (e.g. for scheduling, heuristic solver [21], SMT solver [19, 7]). The interfaces of each module (i.e. set of non-optional input and outputs) are defined in Section 3.3.

We begin by describing the general search algorithm in Section 3.2. Then, the different modules are detailed in Sections 3.3 to 3.8.

3.1 Network and System Models

We define a general model wherein a network $\mathcal{N}$ comprises a set $\mathcal{V}$ of nodes and $\mathcal{E}$ of links, and a set of $\mathcal{F}$ of communication streams, or flows, with one sender (talker) and one or multiple receivers (listeners), and wherein $\mathcal{F}^{TT} \subset \mathcal{F}$ represents the subset of TT flows and $\mathcal{F}^{RC} \subset \mathcal{F}$ represents the subset of RC flows. The set $\mathcal{C}$ represents the set of communication constraints, like maximum end-to-end latency, jitter, or any other optional routing, scheduling, or shaping constraint. For convenience, we also define $\mathcal{P}$ as the complete set of possible network routes between any of the nodes in $\mathcal{N}$ and all of the other.

A flow $f \in \mathcal{F}$ is characterized by the tuple $\langle f.path, f.deadline \rangle$, wherein $path \in \mathcal{P}$ relates to the selected network route between the flow sender and receiver(s), and $deadline \in \mathbb{N}$ corresponds to a maximum end-to-end latency bound requirement. Note that, for the sake of simplicity, we omit to characterize in detail the implicit $1 : N$ relation of flows with its sender node and one, or multiple, receiver nodes. A TT flow $f \in \mathcal{F}^{TT}$ is further characterized by its time-triggered transmission instant[1], $f.offset \in \mathbb{N}$.

We also define $\mathcal{O}$ as the set of solutions in the configuration space, wherein a solution $o \in \mathcal{O}$ represents a possible output of the general framework. The subsets $\mathcal{O}^{RC}, \mathcal{O}^{TT} \subset \mathcal{O}$ represent, respectively, the RC and TT solution space.

We further introduce $\mathcal{F}_s^1 : \langle f_{s_i}, i \in \mathbb{N} \rangle$, and respectively, $\mathcal{F}_s^3 : \langle f_{s_i}, i \in \mathbb{N} \rangle$, as sorted lists, or sequences, with index in the natural numbers, wherein each sequence is equivalent to the respective set, namely $set(\mathcal{F}_s^1) \equiv \mathcal{F}^{TT}$ and $set(\mathcal{F}_s^3) \equiv \mathcal{F}$. Note that the sort operation is described in Section 3.3.

3.2 General search algorithm

The main workflow behind the search algorithm, represented in Algorithm 1, consists of incrementally (re)routing or (re)scheduling one flow at a time following sorted lists maintained by the scheduling and FTA modules. Thanks to three feedback loops, depicted in Figure 1, one flow is identified in each iteration as a candidate to be (re)routed and/or (re)scheduled aiming at iteratively converging towards a better solution. After trying to find an initial solution (line 2), we start the search by ensuring all the TT traffic is routed and scheduled (Loop 1, line 5), as it is a necessary step before doing the formal timing analysis of the RC traffic. Secondly, if the RC traffic does not fulfill its constraints, we attempt to find an acceptable solution by keeping the same routing and only rescheduling TT flows one at the time (Loop 2, line 8). If this fails, then we attempt to reroute a flow (Loop 3, line 11). Hence, with these three feedback loops we explore both routing and scheduling alternatives extensively while limiting time expensive steps such as rescheduling all the TT flows at once.

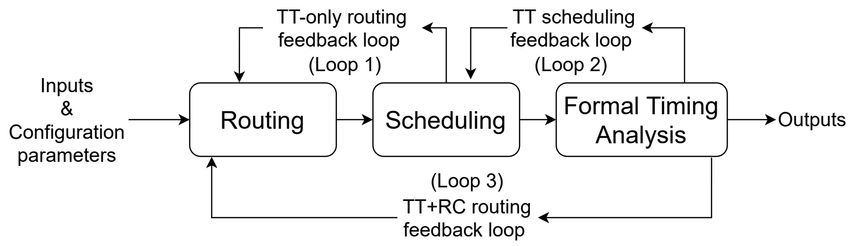

Figure 1 General framework workflow.

[1] Note that we characterize the output of the schedule operation applied to a TT Flow to comprise *offsets*, referring to the transmission instant of said TT Flow on each hop along its route. However, certain time-triggered networks may require additional information, like for example a priority queue assignment in the case of IEEE 802.1Qbv with multiple TT queues (cf. [3]), or alternatively, a mapping to a GCL transmission window (cf. [18]). We claim that accounting for additional elements in the characterization of the schedule output is a trivial generalization and we remain with the simplified notation for the sake of clarity.

Algorithm 1 Main algorithm.

```
Require: $\mathcal{N}, \mathcal{F}, \mathcal{C}$
 1: mem ← memories
 2: initialize($\mathcal{N}, \mathcal{F}$, mem)                                   ▷ See Algorithm 2
 3: repeat
 4:     if $\exists f \in \mathcal{F}^{TT}$: sch.checkConstraints($f$) = $false$ then
 5:         execute($Loop_1$)                                   ▷ Algorithm 3 in Section 3.5
 6:     end if
 7:     if $\exists f \in \mathcal{F}^{RC}$: fta.checkConstraints($f, \mathcal{F}^{RC}, \mathcal{N}, \mathcal{C}$) = $false$ then ▷ Formal timing analysis
 8:         execute($Loop_2$)                                   ▷ Algorithm 4 in Section 3.6
 9:     end if
10:     if $\exists f \in \mathcal{F}^{RC}$: fta.checkConstraints($f, \mathcal{F}^{RC}, \mathcal{N}, \mathcal{C}$) = $false$ then
11:         execute($Loop_3$)                                   ▷ Algorithm 5 in Section 3.7
12:     end if
13: until $\forall f \in \mathcal{F}^{RC}$: fta.checkConstraints($f, \mathcal{F}^{RC}, \mathcal{N}, \mathcal{C}$) = $true$ ∧
14:         $\forall f \in \mathcal{F}^{TT}$: sch.checkConstraints($f$) = $true$
```

▼ **Inputs**: The main algorithm takes the typical inputs when computing network routing and scheduling, i.e. the specification of network topology, $\mathcal{N}$, and flows, $\mathcal{F}$, as well as optional timing and routing constraints, $\mathcal{C}$.

▲ **Outputs**: The output of the algorithm consist of the typical outputs generated by network routing and scheduling algorithms, namely a route for each RC and TT flows as well as a TT flow schedule. Additional shaping or scheduling parameters (e.g. AVB reserved bandwidth, WRR weights if TTE is extended to include these) as well as custom metrics (e.g. minimum end-to-end deadlines, minimum memory configurations) can be optionally added to the output as required. In the particular case of RC shaping and scheduling parameters, these are optionally computed in the FTA module, in Algorithms 1 and 4.

As the solution space can be very extensive, we limit the search algorithm in three dimensions: first, we limit the number of routes that will be tested for each flow. Secondly, we limit the number of iterations performed when computing a new schedule for a particular set of paths. Thirdly, we limit the number of flows that may be selected before re-sorting the list of flows, i.e. rearranging the selection order of those flows. Therefore, we define the following configuration parameters, directing the solution space exploration:

▼ **Configuration parameters**:

- `conf.maxExploredPaths`: maximum number of paths that may be explored for each flow in both routing feedback loops (see Section 3.8.2).
- `conf.maxSchedIterations`: maximum number of iterations in a TT scheduling feedback loop (see Section 3.6);
- `conf.maxExploredFlowReset`: maximum number of flows from the sorted list of flows that can be explored before re-sorting the list (see Section 3.8.1).

These parameters expose different trade-offs enabling the customization of the search to the specifications of given use cases. For instance, limiting the number of routes per flows, `conf.maxExploredPaths`, allows testing more flows within a reasonable amount of time. Limiting the number of schedule search iterations, `conf.maxSchedIterations`, allows testing more routes for each flow. However, the down-side of these limitations is that an optimal solution may be missed or the search may remain within a local optima.

3.3 Module requirements

There are three main functions to be fulfilled for out general framework: routing, scheduling and formal timing analysis. Each function is decoupled from the other two as an independent module, with a defined interface between them. We define the scope of each module with an enumeration of requirement.

The framework allows to leverage existing methods for the implementation of each of its modules. We describe here the requirements for the general case and detail an implementation in Section 4, which is later used in the evaluation in Section 4.2.

- **Routing (`rt`)**, manages functions related to finding network routes for RC and TT flows. The module shall provide:
  - `findRoute(`$f \in \mathcal{F}, \mathcal{N}, \mathcal{C}$`)`: finds an initial path in $\mathcal{N}$ for flow f, subject to constraints $\mathcal{C}$;
  - `allPaths(`$f \in \mathcal{F}, \mathcal{N}, \mathcal{C}$`)` a list of possible paths in $\mathcal{N}$ for flow f, subject to constraints $\mathcal{C}$.
- **Scheduling (`sch`)**, manages functions related to TT traffic. It shall provide:
  - `schedule(`$f, \mathcal{C}$`)` $\rightarrow \mathcal{O}^{TT}$ attempts to schedule TT flow $f \in \mathcal{F}^{TT}$, subject to constraints $\mathcal{C}$;
  - `checkConstraints(`$f \in \mathcal{F}^{TT}$`)` $\rightarrow$ ***boolean*** evaluates whether flow $f \in \mathcal{F}^{TT}$ has been successfully scheduled (i.e. has transmission offset(s));
  - `sortFlows(1,`$\mathcal{F}^{TT}$`)` sort operation over the set $\mathcal{F}^{TT}$ of TT flows, for Loop 1, used to prioritize the flows and guide the search toward a better solution;
  - `sortPaths(`$f \in \mathcal{F}$`)`: sorted list of paths of TT flow f for Loop 1, used to prioritize the paths to guide the search toward a better solution, optionally supported by scheduling information;
  - `costFunction(`$\mathcal{O}^{TT} \rightarrow \mathbb{R}$): cost function to assess a partial solution or save the best solution, in Loop 1.
  - `output` $\in \mathcal{O}^{TT}$: module output, including a TT schedule.
- **Formal timing analysis (`fta`)**, manages network analysis, related to either both RC and TT flows, or only RC flows. It shall provide:
  - `checkConstraints(`$f, \mathcal{F}^{RC}, \mathcal{N}, \mathcal{C}$`)` $\rightarrow$ ***boolean***: evaluates whether the RC flow $f \in \mathcal{F}^{RC}$ fulfills the constraints in $\mathcal{C}$;
  - `impossibilityTest (`$\mathcal{F}^{RC}, \mathcal{N}, \mathcal{C}$`)` $\rightarrow$ ***boolean*** necessary test for constraints in $\mathcal{C}$ being fulfilled by flows $\mathcal{F}^{RC}$ in $\mathcal{N}$ for any flow path and/or TT offset (e.g. the maximum end-to-end latency constraint required is below the minimum possible end-to-end latency on the shortest path);
  - `feasibilityTest(`$\mathcal{F}^{RC}, \mathcal{N}, \mathcal{C}$`)` $\rightarrow$ ***boolean*** evaluates whether the constraints in $\mathcal{C}$ can possibly be fulfilled for the flows $\mathcal{F}^{RC}$ in $\mathcal{N}$ with the current flow paths;
  - `portImpact(`$f \in \mathcal{F}$`)`: evaluates if the path flow f has any port in common with the paths of RC flows which are not yet fulfilling their constraints;
  - `sortFlows(3,`$\mathcal{F}$`)`: sort operation over the set $\mathcal{F}$, for Loop 3;
  - `sortFlows(2, `$\mathcal{F}^{TT}$`)`: sort operation over the set $\mathcal{F}^{TT}$;
  - `sortPaths(`$f \in \mathcal{F}$`)`: sorted list of paths of flow f, for Loop 3;
  - `costFunction(`$\mathcal{O}$) $\rightarrow \mathbb{R}$: cost function to assess a partial solution or save the best solution, in Loop 3.
  - `output` $\in \mathcal{O}^{RC}$: module output, including the parameters for RC shaping/scheduling (e.g. WRR weights, AVB bandwidth reservation) and output requirements (e.g. necessary memory reservation)

3.4 Search initialization

Algorithm 2 Search Initialization algorithm.

```
Require: $\mathcal{N}, \mathcal{F}, \mathcal{C}$, mem
 1: for all $f \in \mathcal{F}$ do                              ▷ Attempts to route all flows
 2:     rt.findRoute($f, \mathcal{N}, \mathcal{C}$)
 3: end for
 4: for all $f \in \mathcal{F}^{TT}$ do                         ▷ Attempts to schedule all TT flows
 5:     sch.schedule($f, \mathcal{N}, \mathcal{C}$)
 6: end for
 7: if $\forall f \in \mathcal{F}^{TT}$ : sch.checkConstraints($f$) $=$ $true$ then
 8:     mem.savedOffsets[$f$] $\leftarrow f.offset, f.path$          ▷ Save current paths and offsets
 9: end if                                                  ▷ Initialize search memories
10: for all $f \in \mathcal{F}$ do
11:     mem.exploredPath_3[$f$] $\leftarrow f.path$
12:     mem.defaultPaths[$f$] $\leftarrow f.path$
13:     mem.exploredPathSets $\leftarrow$ mem.exploredPathSets $\bigcup \langle f, f.path \rangle$
14: end for
15: mem.currentFlow_3 $\leftarrow$ NULL
16: mem.$\mathcal{F}_s^1 \leftarrow \emptyset$
17: mem.$\mathcal{F}_s^3 \leftarrow \emptyset$
18: mem.allTTPathsExplored $=$ $false$
```

In a a first step, the search computes an initial routing and scheduling solution, followed by the initialization of the memories necessary to keep track of the progress, as defined in Algorithm 2. First, all the routes are computed (line 2). Note that there is an implicit failure termination if an initial route cannot be found for each of the flows, meaning that the set of flows is not feasible with the given topology, and hence causing the search to abort with failure. Secondly, the algorithm attempts to schedule TT flows (line 5). Afterward, the memories (`mem`) used in the global search are initialized, namely:

- `mem.savedOffsets`$[f \in \mathcal{F}^{TT}] \leftarrow \{\langle p_1, O_1\rangle..\langle p_n, O_n\rangle : f_i \in \mathcal{F}^{TT}, p_j \in \mathcal{P}, O_i = \{o_i^0..o_i^k\}, o_i^j \in \mathbb{N}\}$: stores the latest successfully scheduled set of TT paths and their offsets. Note that offsets are represented as a set of values corresponding to the transmission offset on each port of a multicast route; `mem.savedOffsets` is used as a restore point to a previous state in which TT Flows were both routed and scheduled, before continuing the search. The use of `mem.savedOffsets` will be detailed in Loops 1 and 2.
- `mem.exploredPathSets` $\leftarrow \{S_0..S_n\} : S_i = \{\langle f_0^i, p_0^i\rangle..\langle f_m^i, p_m^i\rangle : f_j^i \in \mathcal{F}, p_j^i \in \mathcal{P}\}, i \in \mathbb{N}$: storing the set of explored path sets (each flow in a set is associated to one path), shared by the routing feedback loops. This is mainly used to determine whether all solution have been explored;
- `mem.defaultPaths`$[f \in \mathcal{F}] \leftarrow \{p_0..p_n : p_i \in \mathcal{P}\}$: storing the so-called *default paths* previously used by the routing searches. The definition and use of the default paths is detailed in Section 3.5.
- `mem.exploredPaths_3`$[f \in \mathcal{F}] \leftarrow \{p_0..p_n : p_i \in \mathcal{P}\}$: storing the sets of explored paths for each flow (each flow is associated to a list of explored paths) for Loop 3.
- `mem.currentFlow_3` $\leftarrow f \in \mathcal{F}^{TT}$: stores the flows currently being re-routed in Loop 3. It is used to select a new flow to reroute.

- mem.$\mathcal{F}_s^1$: stores a sorted list of the elements of $\mathcal{F}^{TT}$, wherein $\mathcal{F}_{s_k}^{TT}$ is the $k+1^{th}$ element in the list.
- mem.$\mathcal{F}_s^3$: stores a sorted list of the elements of $\mathcal{F}$, wherein $\mathcal{F}_{s_k}$ is the $k+1^{th}$ element in the list.
- `mem.allTTPathsExplored` $\leftarrow$ (*boolean*) flags when all the TT paths have been explored;

Once the initialization is performed the search of a solution begins as described in the following sections.

3.5 Loop 1: TT-only routing feedback loop

Algorithm 3 (Loop 1) tries to re-route and schedule TT flows if not all TT flows were scheduled, either following the initial routing or a re-routing step from Loop 3. The goal is to find a solution in which all TT flows are routed and scheduled, regardless of the RC traffic.

Algorithm 3 Loop 1: TT-only routing feedback loop.

```
Require: $\mathcal{N}$, $\mathcal{F}^{TT}$, $\mathcal{C}$, mem
 1: for all $f_k \in \mathcal{F}^{TT}$ do
 2:     mem.exploredPaths_1[$f_k$] $\leftarrow \{f_k.path\}$          ▷ Initialize with current paths
 3: end for
 4: mem.currentFlow_1 $\leftarrow NULL$
 5: mem.$\mathcal{F}_s^1 \leftarrow$ sch.sortFlows(1,$\mathcal{F}^{TT}$)          ▷ Sort the list of TT flows
 6: while $\exists f_k \in \mathcal{F}^{TT}$ : sch.checkConstraints($f_k$) $= false) \wedge$ mem.allTTPathsExplored
    $= false$ do
 7:     $f' \leftarrow$ selectFlowPath(1,mem.exploredPaths_1)          ▷ See Algorithm 7
 8:     schedule($f$,$\mathcal{N}$,$\mathcal{C}$)
 9:     mem.currentFlow_1 $\leftarrow f'$
10:     if sch.costFunction(output) $<$ sch.costFunction(mem.bestOutput) then
11:         mem.bestOutput $\leftarrow$ current output
12:     end if
13:     if $\forall f_i \in \mathcal{F}^{TT}$ : sch.checkConstraints($f_i$) $= true$ then
14:         for all $f_j \in \mathcal{F}^{TT}$ do          ▷ Save paths and offsets
15:             mem.savedOffsets[$f_j$] $\leftarrow f.offset, f.path$
16:         end for
17:     end if
18:     updateMemories(1, $f'$, mem)          ▷ See Algorithm 6
19: end while
20: if $\exists f_k \in \mathcal{F}^{TT}$ : sch.checkConstraints($f_k$) $= false$ then
21:     if mem.savedOffsets $\neq \emptyset$ then
22:         for all $f_j \in \mathcal{F}^{TT}$ do          ▷ Reset paths and offsets
23:             $f_j.offset \leftarrow$ mem.savedOffsets[$f_j$].offset
24:             $f_j.path \leftarrow$ mem.savedOffsets[$f_j$].path
25:         end for
26:     else
27:         exitPartialSolution(mem.bestOutput)          ▷ Or fail without output
28:     end if
29: end if
```

The search starts from an initial set of paths, called *default path* (stored in `mem.defaultPaths`) and a flow selected to be rerouted. If no schedule is found with any of its different possible paths, the selected flow is set back to the default path, and another flow is chosen. Restoring the path to the already explored default path adds stability to the search process. Algorithm 6 details how new default paths can be selected in order to test the different permutations.

Algorithm 3 (Loop 1) uses three memories:

- `mem.exploredPaths_1`$[f \in \mathcal{F}^{TT}] \leftarrow \{p_0..p_n : p_i \in \mathcal{P}\}$: storing the sets of explored paths for each TT flow (each flow is associated to a list of explored paths) for Loop 1;
- `mem.currentFlow_1` $\leftarrow f \in \mathcal{F}^{TT}$: stores the flows currently being rerouted in Loop 1;
- `mem.bestOutput` $\leftarrow o \in \mathcal{O}$: stores the best solution found so far. This includes all the outputs listed in Section 3.2 which are already available in the current state of the search.

The algorithm begins initializing `exploredPaths_1` with the current paths (lines 1 to 3), and then sorting the set of flows (line 5). Following, a search for a feasible solution is initiated, until either all TT flows are scheduled or all path permutations have been explored (line 6). Within the search, a flow is selected using Algorithm 7, then rerouted, and attempted to be scheduled (lines 7 and 8), following the update of `mem.currentFlow_1` and `mem.bestOutput` (lines 9 to 12).

If a schedule has been found for all TT flows, the paths and offsets are stored in `mem.savedOffsets` (lines 13 to 17), which if needed, can be used to restore a solution with feasible TT traffic offsets (see lines 22 to 25). This is necessary when all available TT path combinations have been tested and Loop 1 finishes, but there remain still untested RC paths that may be explored via Loop 3. Note that the search algorithm may fail and exit in Loop 1, either with a partial or no solution at all, if, directly after the initialization, no valid TT schedule has been found after having explored all paths (line 27).

Finally, after saving the paths and offsets, `mem.exploredPathsSets` and `mem.defaultPaths` are updated as described in Algorithm 6 (line 18).

3.6 Loop 2: TT scheduling feedback loop

Algorithm 4 (Loop 2), identifies, supported by the FTA module, the TT flow with a higher impact on RC traffic (line 17), which is then re-scheduled (line 18) with the aim of finding a solution improving RC traffic performance.

Algorithm 4 uses two memories:

- `mem.exploredOffsets`$[f \in \mathcal{F}^{TT}] \leftarrow \{O_0..O_n : O_i = \{o_i^0..o_i^k\}, o_i^j \in \mathbb{N}\}$: stores the sets of explored offsets for each flow (each flow is associated to a list of explored offsets). Note that offsets are represented as a set of values corresponding to the transmission offset on each port of a multicast route;
- `mem.diversification` $\leftarrow \{f_0..f_n : f_i \in \mathcal{F}\}$: tracks the flows already selected for diversification purposes, allowing to select alternative flows and explore different part of the solution space. Hence, avoiding iterations over stable regions of the solution space by always choosing the same flows.

The search ends when a feasible solution is found or when either the maximum number of iterations, defined in the configuration parameter `conf.maxSchedIterations` (see Section 3.2), is reached, or else when the diversification memory contains all flows (line 16), meaning that no other flow is left to be selected (lines 24 to 28).

Note that Algorithm 4 is a generalization of Algorithm 1 in [7], so we only detail here the main improvements, namely

1. allowing an arbitrary number of constraints $\mathcal{C}$, including the end-to-end latency, as well as the possibility of storing the best solution found so far at any given time (cf. lines 7 and 33);

Algorithm 4 Loop 2: TT scheduling feedback loop.

```
Require: $\mathcal{N}, \mathcal{F}, \mathcal{C}$, mem , conf, currentOutput
 1: it_loop2 $\leftarrow 0$
 2: mem.exploredOffsets, mem.diversification $\leftarrow \emptyset$
 3: if mem.$\mathcal{F}_s^1 = \emptyset$ then
 4:     mem.$\mathcal{F}_s^1 \leftarrow$ fta.sortFlows(2,$\mathcal{F}^{TT}$)
 5: end if
 6: if costFunction(currentOutput) < costFunction(mem.bestOutput) then
 7:     mem.bestOutput $\leftarrow$ currentOutput
 8: end if
 9: fullfilled $\leftarrow \forall f \in \mathcal{F}^{RC}$ : fta.checkConstraints($f, \mathcal{F}, \mathcal{C}$)  ▷ Run the FTA Analysis
10: if fullfilled = $false$ $\wedge$ fta.impossibilityTest($\mathcal{N}, \mathcal{F}^{RC}, \mathcal{C}$) = $true$ then
11:     exit
12: else if fullfilled = $false$ $\wedge$ fta.feasibilityTest($\mathcal{N}, \mathcal{F}^{RC}, \mathcal{C}$) = $true$ then
13:     for all $f_k \in \mathcal{F}^{TT}$ do                                    ▷ Reset offsets
14:         $f_k.offset \leftarrow$ mem.savedOffsets[$f_k$].$offset$
15:     end for
16:     while $\exists f_j \in \mathcal{F}^{RC}$: fta.checkConstraints($f_j, \mathcal{N}, \mathcal{F}^{RC}, \mathcal{C}$)=$false$ $\wedge$
    |mem.diversification|< $|\mathcal{F}^{TT}|\wedge$ it_loop2 < conf.maxSchedIterations do
17:         $f' \leftarrow$ mem.$\mathcal{F}_{s_0}^{TT}$                       ▷ Select flow impacting most RC
18:         sch.schedule($f', \mathcal{N}, \mathcal{C}$)
19:         if $f'.offset \notin$ mem.exploredOffsets[$f'$] then
20:             mem.diversification $\leftarrow \emptyset$
21:         end if
22:         while $f'.offset \in$ mem.exploredOffsets[$f'$] $\wedge$ |mem.diversification| < $|\mathcal{F}^{TT}|$
    do
23:             mem.diversification $\leftarrow$ mem.diversification $\cup f'$
24:             $k \leftarrow 0$
25:             repeat                    ▷ Select first flow not in mem.diversification
26:                 $f' \leftarrow$mem.$\mathcal{F}_{s_k}^{TT}$
27:                 $k \leftarrow k + 1$
28:             until $f' \notin$ mem.diversification
29:             sch.schedule($f', \mathcal{N}, \mathcal{C}$)
30:         end while
31:         $\forall f \in \mathcal{F}^{RC}$ : fta.checkConstraints($f, \mathcal{F}, \mathcal{C}$)
32:         if costFunction(currentOutput) < costFunction(mem.bestOutput) then
33:             mem.bestOutput $\leftarrow$ currentOutput
34:         end if
35:         for all $f_k \in \mathcal{F}^{TT}$ do                   ▷ Update explored offsets
36:             mem.exploredOffsets[$f_k$] $\leftarrow$ mem.exploredOffsets[$f_k$] $\cup f_k.offset$
37:         end for
38:         it_loop2 $\leftarrow$ it_loop2 + 1
39:     end while
40: end if
```

2. the addition of an impossibility check (cf. line 10) as well as a feasibility check (cf. line 12) to avoid searching for solutions when none exists;
3. before a flow is re-scheduled (line 18), all other TT flow offsets are reset to the values stored in `mem.savedOffsets` (lines 13 to 15), already presented in Loop 1. We found that the re-scheduling in Loop 2 can lead to a stable but non-optimal area of the solution space. By restoring the state stored before Loop 2 after re-routing a flow, it is more likely to avoid this area and, hence, find solutions otherwise inaccessible.

3.7 Loop 3: RC+TT routing feedback loop

Finally, the third feedback loop, represented in Algorithm 5, uses the same principle as Loop 1. Namely, if no feasible solution is found after all paths of a specific flow have been tested, the flow is set back to the default path and a new search iteration begins (see line 3 in Algorithm 1). In the case of Loop 3, the selected flow can be either a TT or an RC flow, which enables testing a large array of solutions while trying to prioritize the more likely to succeed first.

Algorithm 5 Loop 3: TT+RC routing feedback loop.

```
Require: $\mathcal{N}, \mathcal{F}, \mathcal{C}$, mem, conf
 1: if mem.$\mathcal{F}_s^3 = \emptyset$ then
 2:     mem.$\mathcal{F}_s^3 \leftarrow$ fta.sortFlows(3,$\mathcal{F}$)
 3: end if
 4: $f'$ = selectFlowPath(3,mem.exploredPaths_3)                ▷ See Algorithm 7
 5: mem.currentFlow_3 $\leftarrow f'$
 6: if $f' \in \mathcal{F}^{TT}$ then
 7:     for all $f_k \in \mathcal{F}^{TT}$ do                         ▷ Reset offsets
 8:         $f_k.offset \leftarrow$ mem.savedOffsets[$f_k$].$offset$
 9:     end for
10:     sch.schedule($f'$,$\mathcal{N}$,$\mathcal{C}$)
11:     if $\forall f_j \in \mathcal{F}^{TT}$ : sch.checkConstraints($f_j$) = $true$ then
12:         mem.savedOffsets[$f_j$] $\leftarrow f_j.offset, f_j.path, \forall f_j \in \mathcal{F}^{TT}$ ▷ Save path and offsets
13:     end if
14: end if
15: updateMemories(3, $f'$, mem)                                  ▷ See Algorithm 6
```

Algorithm 5 (Loop 3) begins sorting the flows if they have not been sorted yet (lines 1 to 3). Then a new flow is selected and rerouted (line 4), followed by the update of `mem.currentFlow_3` in line 5. If the selected flow is TT, it must then be rescheduled. As Loop 3 follows Loop 2, the offsets are restored to the saved values to avoid stable but non-optimal solution space areas (lines 7 to 9), similar to Subsection 3.6. Next the new offsets are computed for the selected flow (line 10) and if successful, they are stored in `mem.savedOffsets` (line 12). If the TT flow cannot be rescheduled, then Loop 1 follow (see line 4 in Algorithm 1).

3.8 Common support algorithms

In this section, we describe two algorithms supporting both Loop 1 and Loop 3. Algorithm 6 updates the memories and manage the default paths, while Algorithm 7 implements the selection of a new flows and paths.

3.8.1 Update of memories

The goal of Algorithm 6 is to update the memories tracking the progress of the search, such as the set of explored paths `mem.exploredPaths_i` and `mem.exploredPathSets`.

Algorithm 6 `updateMemories(i)`.

```
Require: $\mathcal{N}, \mathcal{F}, \mathcal{C}$, mem, conf, mod $\in \{sch, fta\}$, $i \in \{1,3\}$    ▷ Current loop index (1,3)
1: $f_c$ =mem.currentFlow_i    ▷ Current flow in Loop $i, \in \{1,3\}$
2: mem.exploredPaths_i$[f_c] \leftarrow f_c.path$    ▷ Update path selected flow
3: if $(\bigcup\langle f_k, f_k.path\rangle : f_k \in$mem.$\mathcal{F}_s^i) \notin$ mem.exploredPathSets then
4:     mem.exploredPathSets $\leftarrow$ mem.exploredPathSets $\bigcup\langle f_i, f_i.path\rangle : \forall f_i \in \mathcal{F}$
5:     if $|$mem.exploredPaths_i$| \geq$ conf.maxExploredFlowReset $\vee \big[ i = 3 \wedge$ fta.portImpact$(f_c.path) =$ false $\big]$ then
6:         mem.exploredPaths_i$[f_c] \leftarrow \emptyset$
7:         mem.$\mathcal{F}_s^i \leftarrow$ mod.sortFlows$(i, \mathcal{F}^i)$
8:     end if
9: else if $\forall f_p \forall p_q : f_p \in$mem.$\mathcal{F}_s^i, p_q \in$ rt.allPaths$(f_p, \mathcal{N}, \mathcal{C})$, $p_q \in$ mem.exploredPath_i then    ▷ All flow paths tested for current default paths
10:    if $i = 1 \wedge$ savedOffsets $\neq \emptyset \wedge \forall f_p \forall p_q : f_p \in$mem.$\mathcal{F}_s^i, p_q \in$ rt.allPaths$(f_p, \mathcal{N}, \mathcal{C})$, $p_q \in$ mem.defaultPaths then
11:                                                  ▷ All flow paths tested as default paths
12:        mem.allTTPathsExplored $\leftarrow$ $true$
13:        for all $f_l \in \mathcal{F}^{TT}$ do    ▷ Reset paths and offsets
14:            $f_l.path =$ savedOffsets$[f_l].path$
15:            $f_l.offset =$ savedOffsets$[f_l].offset$
16:        end for
17:    else
18:        $f'.path =$ selectFlowPath($i$, mem.usedDefaultPaths)    ▷ Algorithm 7
19:        if $f' \in \mathcal{F}^{TT}$ then
20:            for all $f_l \in \mathcal{F}^{TT}$ do    ▷ Reset offset
21:                $f_l.offset \leftarrow$ mem.savedOffsets$[f_l]$
22:            end for
23:            sch.schedule$(f', \mathcal{C})$
24:            if $\forall f \in \mathcal{F}^{TT}$ : sch.checkConstraints$(f) = true$ then
25:                for all $f_l \in \mathcal{F}^{TT}$ do    ▷ Save path and offsets
26:                    mem.savedOffsets$[f_l] \leftarrow f_l.offset, f_l.path$
27:                end for
28:            end if
29:        end if    ▷ Update default path
30:        mem.defaultPaths$[f']$.path$\leftarrow$ mem.defaultPaths$[f']$.path $\bigcup f'.path$
31:        mem.exploredPaths_i $\leftarrow \emptyset$
32:        mem.$\mathcal{F}_s^i \leftarrow$ mod.sortFlows$(i, \mathcal{F}^i)$
33:    end if
34: else if $f_c =$ mem.$\mathcal{F}_{s_k}^i : k = |$mem.$\mathcal{F}_s^i| - 1$ then ▷ All flows of the current set were tested
35:    mem.exploredPaths_i $\leftarrow \emptyset$
36:    mem.$\mathcal{F}_s^i \leftarrow$ mod.sortFlows$(i$,mem.$\mathcal{F}^i)$
37: end if
```

For Loop 1 and 3 we use the *default path* memory to explore around a stable set of paths, i.e. the default paths, and only after that exploration is concluded the set of default paths is updated and a new exploration around the new stable set of paths begins. With this, we avoid the excessive time it would take to explore all the solutions around the default paths, when it is likely that not all of them lead to feasible solutions.

A first part of our strategy to guide the search toward a better part of the solution space is to sort the flows, as already explained. However, this is not sufficient due to the fact that when a new path is found, or the default paths change, it has a global impact on the totality of flows. This leads to the need to regularly re-sort the flows so as to continue testing those with the highest impact on the flows causing more trouble to the algorithm. It is important to note that always re-sorting is also not a good approach, as it can lead to selecting always the same flows and not making progress.

In Section 3.2 we define `conf.maxExploredFlowReset`, which lets the user configure after how many tested flows the memories `mem.exploredPaths_i` are reset and the flows resorted. This parameter can be tailored based on the characteristics of the network.

The algorithm begins setting the current flow (line 1) and updating `mem.exploredPaths_i` (line 2). Then, if the current set of path is unknown to `mem.exploredPathSets` it is added (line 3). If enough flows have been explored or if, in Loop 3, the path of the current flow has no impact on the flows not fulfilling their constraints, then `mem.exploredPaths_i` is reset and the flows resorted (lines 6 and 7).

However, if the path set has already been explored and all combinations of flow paths have been testes for the current default paths (line 9), it is checked if, in Loop 1, all the TT path combinations have been tested (lines 10). If that is the case and there are saved offsets in `mem.savedOffsets`, then `mem.allTTPathsExplored` is set to *true* and the paths and offsets are resets to the saved values (lines 13 to 16). If that is not the case, then a new default path is selected (line 18). If it consists of a TT flow, its offsets are reset to the saved values, if possible (line 20) and the flow is rescheduled (line 23). If all TT flows are scheduled, then the paths and offsets are saved in `mem.savedOffsets` (line 25). Following, the memory `mem.defaultPaths` is updated with the new default path (line 30), and the flows are resorted to select the more promising flows around the newly chosen default paths. Finally, if the selected flow was the last on the list (line 34), `mem.exploredPaths_i` is updated and the flows again resorted (lines 35 and 36) in preparation for the next iteration.

3.8.2 Selection of a flow and path

Algorithm 7 is used to select a flow and its new path within Loops 1 and 3. The algorithm selects the first untested path of the current flow f_c in the sorted list of flows (line 25).

Additionally, we define `sortPaths`(f_c, `conf.maxExploredPaths`) instantiating the functions `sortPaths(`f_c`)`, respectively from the FTA module in Loop 1, or the Scheduling module in Loop 3, and selecting the first `conf.maxExploredPaths` items of the provided sorted list.

The algorithm tries setting the current flow (line 1) or, if none (line 2), selects the first flow from the sorted list (line 3). If the maximum configured number of flows (i.e. `conf.maxExploredPaths`) has been reached (line 4), the flow path is reset to the latest default path (lines 5). If the flow happens to be TT, the offsets are resets, if possible, (lines 7 to 9) and the flow is reschedule (line 10). Upon success, the paths and offsets are saved (lines 12 to 14).

The next flow in the sorted list is selected as the new current flow (line 17) and the first path not in memory is selected to reroute the flow (lines 19 to 24).

Algorithm 7 `SelectFlowPath(i)`: Flow and Path selection.

```
Require: $\mathcal{N}, \mathcal{F}, \mathcal{C}$, conf, $i \in \{1,3\}$, mem
 1: $f_c$ = mem.currentFlow_i                          ▷ Current flow for Loop i, $i \in \{1,3\}$
 2: if $f_c = NULL$ then
 3:     $f_c \leftarrow$mem.$\mathcal{F}^i_{s_0}$                          ▷ First flow in the sorted list
 4: else if |memory[$f_c$]| = conf.maxExploredPaths then
 5:     $f_c.path \leftarrow$ mem.defaultPaths[$f_c$].path[k]: k=|mem.defaultPaths[$f_c$]|-1
 6:     if $f_c \in \mathcal{F}^{TT}$ then
 7:         for all $f_n \in \mathcal{F}^{TT}$ do
 8:             $f_n.offset \leftarrow$ mem.savedOffsets[$f_n$]                ▷ Reset offset
 9:         end for
10:         sch.schedule($f_c, \mathcal{N}, \mathcal{C}$)
11:         if $\forall f_k \in \mathcal{F}^{TT}$ : sch.checkConstraints($f_k$) = $true$ then
12:             for all $f_n \in \mathcal{F}^{TT}$ do
13:                 mem.savedOffsets[$f_n$] $\leftarrow f_n.offset, f_n.path$    ▷ Save offset and path
14:             end for
15:         end if
16:     end if
17:     $f_c \leftarrow$mem.$\mathcal{F}^i_{s_{k+1}}$ : mem.$\mathcal{F}^i_{s_k} = f_c$               ▷ Next flow in the sorted list
18: end if
19: $P^s \leftarrow$ sortPaths($f_c$, conf.maxExploredPaths)
20: j $\leftarrow$ 0
21: while $P^s_j \in$ memory[$f_c$] do
22:     j $\leftarrow$ j+1
23: end while
24: $f_c.path \leftarrow P^s_j$                          ▷ Set first sorted path not in memory[$f_c$]
25: return $f_c$
```

Note that the Algorithm 7 is instantiated with the inputs $i \in \{1,3\}$ and `memory = mem.exploredPaths_i` (e.g. line 4 in Algorithm 5) to select the *new current flow*, or with the inputs $i \in \{1,3\}$ and `memory = mem.defaultPaths` (see line 18 in Algorithm 6) to select a *new default path*.

4 Performance evaluation

We have so far presented a general search framework for routing, scheduling and formal timing analysis. In this section, we present a performance evaluation with an implementation of the framework for TTEthernet, following a detailed industrial case study supporting our analysis, and a discussion of the evaluation results.

4.1 Implementation of modules for TTEthernet

For the performance evaluation, we consider two user-provided input constraints, namely *maximum end-to-end latency* for RC and TT flows, and *maximum available frame memory* for switches. The output provided by the general framework, in addition to the paths and schedule, are the i) jitter of RC flows at the switch ports, ii) queue memory reservation requirements for critical traffic, allowing to properly dimension the switch memory and maximizing the remaining capacity for best-effort traffic, and iii) minimum end-to-end delay for RC flows (i.e. minimum achievable deadlines) based on the best solution found (i.e. with the minimum cost function).

We detail below the implementation of each module function described in Section 3.3 used for the evaluation of the general framework.

- **Routing (`rt`)** is implemented as follows:
  - `findRoute()`: is an implementation of the method in [2] with additional load balancing;
  - `allPaths()`: leverages the JAVA library *JGraphT* [13, 15] to compute all possible paths using the class `AllDirectedPaths`;
- **Scheduling (`sch`)** consists of an *SMT-based* scheduler as described in [7]:
  - `schedule()`: is computed using the constraints in [7], Section III C;
  - `checkConstraints()`: is a trivial check of the existence of offsets for the flow;
  - `sortFlows()`: detailed in Algorithm 8;
  - `sortPaths()`: detailed in Algorithm 9;
  - `costFunction()`: is directly proportional to the number of non-scheduled TT flows;
- **Formal timing analysis (`fta`)**, implements *Network Calculus* with linear curves[2]:
  - `checkConstraints()`: checks the RC constraints by implementing the TTE model proposed in [26], additionally considering the higher priority *Protocol Control Frames* (PCFs) flows in TTEthernet[3]. Therefore, it subtracts the PCF arrival curve (i.e. the maximum amount of data that can arrive in any time interval) alongside the TT arrival curve (cf. Theorems 3 and 7 in [26]);
  - `impossibilityTest()`: implements a simple check returning *true* when at least one flow exists, for which its maximum end-to-end deadline is less than its minimum possible end-to-end latency on the shortest path (based on the fastest possible transmission delays);
  - `feasibilityTest()`: implements a necessary optimistic analysis, whereby instead of considering the impact of the TT flow offsets as [26] to compute the maximum burst of TT traffic impacting RC, it only considers, in each output port, the maximum frame size among all transmitted TT flows. Therefore, the worst-case output port delays will be greater than or equal to the optimistic bound, and consequently, `feasibilityTest()` returns *false* if there is at least one flow with its maximum end-to-end deadline being less than the optimistic computed value;
  - `portImpact()`: checks if the input flow intersects with a flow not fulfilling its constraints;
  - `sortFlows()`, *Loop 2*: the sorted list of TT flows with highest impact on RC flows is computed by sorting TT flows from highest to lowest cardinality (Card). For details refer to lines 2 to 14 of Algorithm 2 in [7];
  - `sortFlows()`, *Loop 3*: the sorted list of RC+TT flows with highest impact on RC flows not fulfilling their deadlines[4] is computed using Algorithm 8, with `mod=FTA`, wherein `fta.intersections(`$j \in \mathcal{F}$`)` is the number of ports in the current path of flow j which are in common with the flows not fulfilling their constraints;

[2] Network Calculus is a framework allowing to compute upper-bounds for flow delays as well as backlogs, which we use to check the fulfillment of RC constraints, such as end-to-end latency, jitter and memory occupancy (cf. [14].

[3] Protocol Control Frames (PCFs) are Ethernet frames periodically transmitted by Synchronization Master nodes to implement the fault-tolerant clock synchronization in TTEthernet (cf. [12])

[4] Note that Loop 3 is only iterated if all TT flows are scheduled.

- `sortPaths()`: Algorithm 9 implements the sorted list of paths;
- `costFunction()`: is assessed by adding i) the number of non-scheduled TT flows, plus ii) among all flows, the average difference between the flow deadline and its assigned end-to-end delay;

Algorithm 8 Best flow to re-route for `sch.sortFlows`(1,$\mathcal{F}^{TT}$) and `fta.sortFlows`(3,$\mathcal{F}$).

```
Require: $\mathcal{F}, i \in \{1,3\}, f_A, f_B \in \mathcal{F}$ , mod $\in \{sch, fta\}$
1: $sch_A \leftarrow$ mod.checkConstraints($f_A$)
2: $sch_B \leftarrow$ mod.checkConstraints($f_B$)
3: if $f_A \in \mathcal{F}^{RC} \wedge f_B \in \mathcal{F}^{TT}$ then        ▷ Case Loop 3
4:     return $f_A$
5: else if $sch_A = false \wedge sch_B = true$ then
6:     return $f_A$
7: else if $sch_A = false \wedge sch_B = false \wedge f_A.deadline > f_B.deadline$ then
8:     return $f_A$
9: else if $\big[sch_A = true \wedge sch_B = true \wedge \{$ mod.intersections($f_A$) > mod.intersections($f_B$) $\vee$ { mod.intersections($f_A$) = mod.intersections($f_B$) $\wedge (f_A.deadline - f_A.delay) > (f_B.deadline - f_B.delay)\}\}\big]$ then
10:    return $f_A$
11: else
12:    return $f_B$
13: end if
```

In Algorithm 8 we denote $deadline_j$ the deadline, and $delay_j$ the end-end-end latency, of TT flow f_j. When instantiated with $mod = sch$, `sch.intersections`($f_j \in \mathcal{F}^{TT}$) computes the number of ports in the path of TT flow f_j which are in common with the non-scheduled TT flows (note that a port in common with n flows is accounted n times).

RC flows are the best candidates to be rerouted as they do not necessitate rescheduling, which is very time expensive (line 3), therefore they are sorted first. The next best flows are flows not fulfilling their constraints (line 5), followed by flows not fulfilling their constraints with larger deadlines (line 7). The rationale of this strategy is the following: during initialization, in Algorith 2, the selected paths tend to be among the shortest paths available. Therefore, by rerouting flows on potentially longer (and hopefully less loaded) paths, there is a higher chance to both find an acceptable path for the current flow and decrease its impact on other flows having shorter deadlines, which may likely not fulfill their deadlines on longer paths anyway. Finally, for flows fulfilling their constraints, those with the highest impact on flows not fulfilling their constraints are chosen, with the expectation of decreasing this impact by using a new path (lines 9 and 13). If the resulting impact is equivalent, the flow with the largest difference between deadline and their calculated end-to-end latency is selected, for the the reason of having a larger leeway.

In algorithm 9 we denote `sch.totalBandwidth`($p \in \mathcal{P}$) the sum of the bandwidth of TT flows in each output port of path. We denote `fta.totalBandwidth`(path) the sum of the bandwidth of RC+TT flows in each output port of path (the computation is done as the maximum of the sum per receiver, same as for `fta.totalTime`(path_i)). The function `fta.intersections(path)` is equivalent to `fta.intersections(`$j \in \mathcal{F}$`)`, but applied to the input path, instead of the current path of flow j;

To compare different paths of an RC flow (lines 1 to 4), a rough estimation of the RC end-to-end latency is used, using data previously computed via Network Calculus. Therefore `totalTime(path_i, receiver)` is computed for each flow receiver, as the sum

Algorithm 9 Best path to re-route for **sortPaths**($f \in \mathcal{F}$).

```
Require: $f \in \mathcal{F}, p_1, p_2 \in \mathcal{P}$, mod $\in \{sch, fta\}$
1: if $f \in \mathcal{F}^{RC} \wedge$ fta.checkConstraints($f$) $= false$ then
2:     if fta.totalTime($p_1$) $<$ fta.totalTime($p_2$) then
3:         return $p_1$
4:     end if
5: else if [ mod.intersections($p_1$) $<$ mod.intersections($p_2$) ] $\vee$ [ mod.intersections($p_1$) $==$ mod.intersections($p_2$) $\wedge$ ( mod.totalBandwidth($p_1$) $<$ mod.totalBandwidth($p_2$) ) ] then
6:     return $p_1$
7: end if
8: return $p_2$
```

of the delays in each output port of `path_i` from the sender to a receiver. We denote `fta.totalTime`(path_i) as the maximum of `totalTime(path_i, receiver)` over all the receivers of `flow_i` of `path_i`. Hence, the algorithm begins by selecting the path with the strictly smaller estimated total end-to-end-delay using `totalTime(`p_1`)` (line 2). This is more likely to be a valid path for the current flow.

For a TT flow, the preferred path is that with less intersections with flows not fulfilling their constraints, to lessen the impact of this flow on them (line 5). If the number of intersections are identical, then the less loaded path is preferred, since the flow should have better chances of fulfilling its deadline as well as interfering with fewer other flows (line 5).

4.2 Industrial case study: the Orion network

For the performance evaluation we consider the Orion network, illustrated in Figure 2, based on the *Orion Crew Exploration Vehicle* (CEV), 606E baseline as presented in [9] and described in [23, 16]. The network consists of i) 99 TT flows with periods varying from 7.5 ms to 187.5 ms and maximum frame sizes between 87 bytes and 1518 bytes; ii) 87 RC flows with periods from 4 ms to 128 ms and maximum frame sizes from 89 to 1499 bytes. Each TT and RC flow i has a defined deadline constraint, denoted as `deadline_initial_i`.

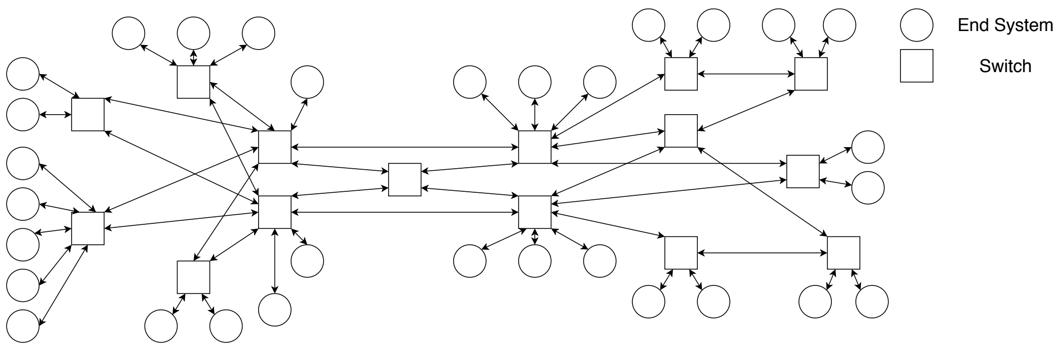

Figure 2 Orion network topology.

We have empirically determined two sets of input parameters listed in Table 1. We have observed that 10 is a good limit for trying to re-schedule TT flows in Loop 2 and, similarly, we have settled to 70% of the total number of RC+TT flows for the parameter `conf.maxExploredPaths`. Both settings show to be a good compromise exploring TT re-routing without exhausting all possibilities, which result in a highly computationally expensive step. We have selected two different values for `conf.maxExploredPaths`, i.e. 2 and 10, to show the large impact this specific parameter has. Nevertheless, we foresee that a further study of the sensitivity of the parameters in a wider range of use cases would be beneficial in future work.

Table 1 Sets of input parameters.

Parameter	**GF: MEP 10**	**GF: MEP 2**
`conf.maxSchedIterations`	10	10
`conf.maxExploredPaths`	10	2
`conf.maxExploredFlowReset`	0.7×187	0.7×187

To assess our proposal, we compare *GF: MEP 2* and *GF: MEP 10* to three other results from literature:

- **GF Shortest path**: we consider that the general framework is not limited by the configuration parameters `conf.maxSchedIterations`, `conf.maxExploredPaths`, and that `conf.maxExploredFlowReset`=0, i.e. the flows are always resorted and the memory `mem.exploredPaths_i` cleared. Moreover, the flows not fulfilling their constraints are sorted as proposed in Algorithm 8, and the flows fulfilling their constraints are sorted from highest to lowest end-to-end latency, while the path are sorted from shortest to longest. The schedule is computed with SMT, using the constraints of Section III C [7];
- **Scheduling loop** [7]: we implement the solution described in [7]
- **Static (no loop)** [19]: we consider that routing and scheduling are set with the initial solutions described in Section 4.1 and [19].

We would have liked to compare our proposal to [27], and [9], but as will be explained in Section 4.4, we lack information about their use cases to be able to do a proper comparison. However, with the three methods selected for comparison, we will be able to assess the impact of both re-routing and re-scheduling (i.e.**Static (no loop)**), the importance of re-routing in addition to re-scheduling (i.e. **Scheduling loop**), and the importance of selecting the best parameters and heuristics in the modules (i.e. **GF Shortest path**).

We have defined two test cases: i) computation of minimum end-to-end deadline constraints, and ii) analysis of deadline reduction. For i), we set all the deadlines to their initial values, except the deadline of the flows for which we want to obtain the minimum possible. Those are initialized to their minimum end-to-end latency based on the fastest possible transmission (i.e. minimum latency without any queuing delay). We set a timeout to conclude the search after 1 hour with the best found solution. For ii), we analyze the execution time (denoted *exec. time*), and cost function (denoted *cost*), when varying the deadlines of all RC flows proportionally to the initial deadline within the range 50%..100%, as shown in Table 2. For this experiment we set the timeout value to 24 hours.

It is important to note that for a small network, limiting `conf.maxExploredPaths` may limit the number of explored flows and paths, due to some part of the solution space not being accessible. Indeed, experiments run on a network with only 4 switches and 10 flows showed *GF: MEP 10* to perform better. However, for larger networks, the solution space becomes so large that exploring all possibilities tends to be intractable. Therefore, guiding the search with the parameter `conf.maxExploredPaths` shows effective. In the proposed Orion Network, both *GF: MEP 10* and *GF: MEP 2* explore the solution space until a solution is found or until a timeout expires. Limiting `conf.maxExploredPaths` does not significantly affect the total number of explored solutions, but instead guides more strongly the search toward regions of the solution space more likely to contain better solutions.

4.3 Evaluation results

In our tests, the switch memory constraints were always fulfilled, so we concentrated our efforts on the end-to-end deadlines. Note that an evaluation similar to the one we provide here could also address the switch jitter and switch memory allocation.The results of test case i) are presented in Figure 3. The minimum deadlines provided by the search are normalized using the minimum deadlines found for the *Static (no loop)* case.

The results in Figure 3 show that *GF: MEP 2* and *GF: MEP 10* find much lower deadlines than either the *Scheduling loop* and the *Static (no loop)* searches. In fact, on average, the deadlines found with *GF: MEP 2* (resp. *GF: MEP 10*) are smaller by 21% (resp. 12%) compared to *Static (no loop)*, with a maximum decrease of 28.3% for *rc_850* (resp. 23.1% for *rc_190*). On the contrary, *Scheduling loop* only reduces the minimum deadlines by 1.8% on average, with results ranging from a slight increase of 0.9% for *rc_620* to a decrease of 4.5% for *rc_560*.

The increase of minimum deadlines compared to *Static (no loop)* is due to the use of an optimization function added in the SMT as described in [7]. While it typically decreases the average delays, due to the nature of the optimization constraint and SMT solvers, in some cases it can also do the opposite.

With respect to *GF: Shortest path*, we see that the minimum deadlines found are on average 1.1% lower than the deadlines found with *Static (no loop)*, from an increase of 1.6% to a decrease of 5.1%. This first set of results confirms that guiding more strongly the search with smaller values of `conf.maxExploredPaths` is effective for large networks.

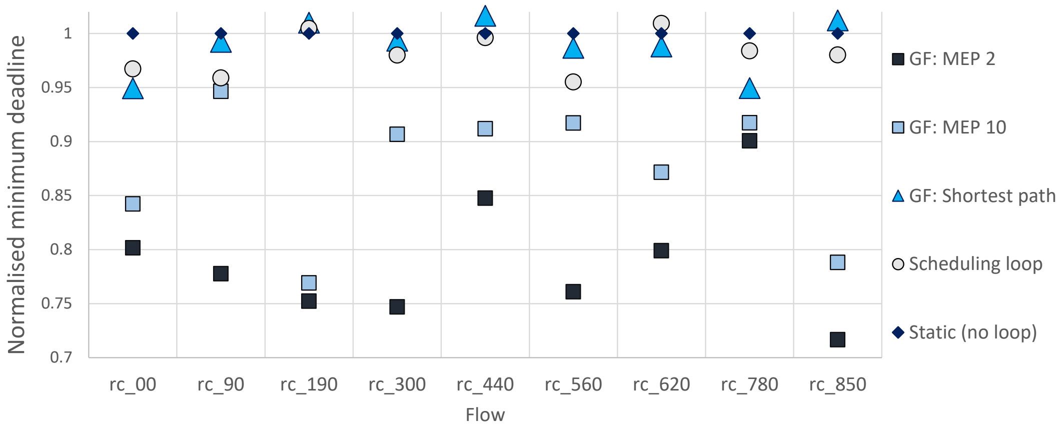

Figure 3 Normalised minimum deadlines for 9 randomly selected flows.

The results of test case ii) are presented in Table 2. In addition, we run an experiment with *GF: MEP 2* with deadline settings to 55%, for which we obtained an acceptable solution (i.e. cost = 0) in 18h 32min. With the exception of *Static (no loop)*, all other results in the range 100% to 80% are equivalent, finding an acceptable solution after the first iteration within 20 min. We can see that without the SMT optimization function, i.e. *Static (no loop)*, the computation time is shorter, i.e. 13 min. However, with *Static (no loop)* the deadlines cannot be reduced under 75%, with a cost function value of 1.000013. With *Scheduling loop* (resp. *GF: Shortest path*) however, the deadlines are reduced to 70%, but at the cost of a large computation time, i.e 56 min (resp. 2h 43min). With *GF: MEP 10*, we are able to reduce the deadlines down to 65%, and with *GF: MEP 2* we find a solution for 55%.

Unsurprisingly, the execution time increases when the deadlines decrease, i.e. when a solution is more difficult to find. Nevertheless, at 70%, *GF: MEP 2* finds a solution twice as fast as *Scheduling loop*, and 6 times faster than *GF: Shortest path*.

Table 2 Results when varying deadlines from 100% to 50% of the initial deadlines.

deadlines	100% & 80%		75%		70%	
metrics	exec. time	cost	exec. time	cost	exec. time	cost
GF: MEP 2	20 min	0	22 min	0	26 min	0
GF: MEP 10	20 min	0	22 min	0	32 min	0
GF: shortest path	20 min	0	28 min	0	2h 43min	0
Scheduling loop [7]	20 min	0	53 min	0	56 min	0
Static (no loop) [19]	13 min	0	13 min	0	13 min	1.000013
deadlines	65%		60%		50%	
metrics	exec. time	cost	exec. time	cost	exec. time	cost
GF:MEP 2	45 min	0	4h 20min	0	24h	2.00094
GF: MEP 10	55 min	0	24h	1.000028	24h	3.00055
GF: shortest path	24h	1.00026	24h	2.00044	24h	6.00100
Scheduling loop [7]	24h	1.00040	24h	2.00041	24h	6.00066
Static (no loop) [19]	13 min	2.00027	13 min	4.00029	13 min	6.00075

Hence, in our test case ii), we have shown that *GF: MEP 2* improves the maximum deadline reduction by at least 26.7%, from 75% to 55% compared to *Static (no loop)*, and finds an acceptable solution quicker than the other searches we compared it to, when the deadlines are constraining (e.g. 70%). When no solution is found within the allocated time, *GF: MEP 2* is the search that finds the solution with the smallest cost (e.g. 50%). The number of flows not fulfilling their constraints is divided by 3 when the deadlines are divided by 2 (i.e. 50%).

The two test cases show that with our proposed general framework, we can largely reduce the deadlines with regard to the compared state-of-the-art, i.e. *Scheduling loop* [7] and *Static (no loop)* [19]. This is because both methods explore a much reduced solution space compared to our proposal and, in particular, due to the fixed routing (and scheduling for *Static(no loop)*), they are unable to find better solutions.

The comparison between *GF: MEP 2* and *GF: MEP 10* shows the importance of selecting good parameters for the search, and the comparison with *GF: shortest path* shows the importance of selecting good parameters and good heuristics to obtain good results. In the case of shortest path, we observe that i) many paths which are longer in terms of number of hops but shorted in terms of delays are disregarded, and ii) the lists are constantly resorted, which is time expensive and can cause a lack of diversity in the selected flows and paths.

We can see that selecting a low value for `conf.maxExploredPaths` works well on large network in which exploring all solution within an acceptable time limit is not reasonable, and so potentially, reducing the solution space does not affect the number of explored solution compared to setting a larger value of `conf.maxExploredPaths`. Indeed, the only difference is which solution are being tested within the time limit. However, for smaller network larger values of `conf.maxExploredPaths` are advisable so as not to limit the number of explored solutions.

4.4 Comparison to related work

To conclude the performance evaluation, we compare our results to those found in four previous works: [19], [7], [27], and [9]. In Section 4.3, we have compared our proposed method against previous literature, namely: *Scheduling loop* [7] and *Static (no loop)* [19]. We have shown that compared to *Static (no loop)* (resp. *Scheduling loop*), our proposed solution can decrease the minimum deadlines by up to 28.3% (resp. 26.9%).

In [27], the authors compare their proposed method to the shortest paths (SPA) in a TTE network. They used SMT to compute the TT schedule as described in [19]. So, *Static (no loop)* is very close to the SPA implemented in [27]. In the performance evaluation of [27], we see that they obtain a maximum reduction of 6.41% of the worst-case delays compared to SPA, which is significantly less than the 28.3% we have obtained with our proposed method (the minimum deadline being equal to the worst-case delay). The execution times are not provided in [27], so we cannot compare the results for this metric.

In [9], the evaluation is done on a TSN network, using the schedulability of the flows to assess the solution. Unfortunately, not enough information about deadlines and traffic load is provided, which prevents a performance comparison in our evaluation.

5 Conclusion

In this paper we have presented a general framework for routing, scheduling and formal timing analysis in deterministic networks (e.g. TSN, TTE). The general framework leverages user-defined modules (i.e. routing, scheduling and Formal Timing analysis) to search for a solution fulfilling arbitrary constraints (e.g end-to-end RC and TT delays) and outputs the best found solution (i.e. TT and RC routing, TT schedule) based on a defined cost-function.

We have provided implementation details for an instantiation of the general framework for TTEthernet, with example module implementations, input and output constraints, and cost functions. With this implementation we have evaluated the performance of our proposed general framework, compared to two state of the art methods.

We have shown that selecting good heuristics and good parameters is of paramount importance to obtain good results, and that the minimum deadlines (i.e. worst-case delays) can be reduced up to 28.3% with our proposed method, compared to the state-of-the-art solution. We have also shown that we are able to divide by up to 3 the number of flows not fulfilling their constraints compared to prior work.

The importance of good parametrization has been highlighted to select each parameter value and obtain the best solutions in the least amount of time. However, future work is necessary to analyze the impact of each parameter on the cost function and execution time, subject to networks and flow characteristics.

References

1 AEEC. *ARINC PROJECT PAPER 664, AIRCRAFT DATA NETWORKS, PART7, AFDX NETWORK (DRAFT)*. AERONAUTIC RADIO, INC., 2551 Riva Road, Annapolis, Maryland 21401-7465, November 2003.

2 Moses Charikar, Chandra Chekuri, To-yat Cheung, Zuo Dai, Ashish Goel, Sudipto Guha, and Ming Li. Approximation algorithms for directed steiner problems. *Journal of Algorithms*, 33(1):73–91, 1999.

3 Silviu S. Craciunas, Ramon Serna Oliver, Martin Chmelik, and Wilfried Steiner. Scheduling real-time communication in IEEE 802.1Qbv Time Sensitive Networks. In *24th International Conference on Real-Time Networks and Systems (RTNS)*. ACM, 2016.

4 J. Diemer, D. Thiele, and R. Ernst. Formal worst-case timing analysis of Ethernet topologies with strict-priority and AVB switching. In *Proc. International Symposium on Industrial Embedded Systems (SIES)*. IEEE Computer Society, 2012.

5 Frank Dürr and Naresh Ganesh Nayak. No-wait packet scheduling for IEEE Time-sensitive Networks (TSN). In *Proc. RTNS*. ACM, 2016.

6 Jonathan Falk, Frank Dürr, and Kurt Rothermel. Exploring practical limitations of joint routing and scheduling for TSN with ILP. In *2018 IEEE 24th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 136–146. IEEE, 2018.

7 Anaïs Finzi and Silviu S. Craciunas. Integration of SMT-based scheduling with RC network calculus analysis in TTEthernet networks. In *2019 24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 192–199. IEEE, 2019.

8 Fabrice Frances, Christian Fraboul, and Jérôme Grieu. Using network calculus to optimize the AFDX network. In *Embeeded Real Time Software and Systems (ERTS)*, 2006.

9 Voica Gavriluţ, Luxi Zhao, Michael L Raagaard, and Paul Pop. AVB-aware routing and scheduling of time-triggered traffic for TSN. *Ieee Access*, 6:75229–75243, 2018.

10 Jérôme Grieu. *Analyse et évaluation de techniques de commutation Ethernet pour l'interconnexion des systèmes avioniques*. PhD thesis, INPT, 2004.

11 Institute of Electrical and Electronics Engineers, Inc. Time-Sensitive Networking Task Group. `http://www.ieee802.org/1/pages/tsn.html`, 2016. retrieved 30-Nov-2020.

12 Issuing Committee: As-2d2 Deterministic Ethernet And Unified Networking. SAE AS6802 Time-Triggered Ethernet. `https://www.sae.org/standards/content/as6802/`, 2011. retrieved 30-Nov-2020.

13 JGraphT team and contributors. JGraphT, September 2016. version: 1.0.0. URL: `https://jgrapht.org/`.

14 J.Y. Le Boudec and P. Thiran. *Network calculus: a theory of deterministic queuing systems for the Internet*, chapter 1, pages 3–81. Springer-Verlag, 2001.

15 Dimitrios Michail, Joris Kinable, Barak Naveh, and John V. Sichi. JGraphT—A Java Library for Graph Data Structures and Algorithms. *ACM Trans. Math. Softw.*, 46(2), May 2020.

16 M Paulitsch, E Schmidt, B Gstottenbauer, C Scherrer, and Kantz H. Time-triggered communication (industrial applications). *Time-Triggered Communication*, pages 121–152, 2011.

17 Eike Schweissguth, Peter Danielis, Dirk Timmermann, Helge Parzyjegla, and Gero Mühl. ILP-based joint routing and scheduling for time-triggered networks. In *Proceedings of the 25th International Conference on Real-Time Networks and Systems*, pages 8–17, 2017.

18 Ramon Serna Oliver, Silviu S. Craciunas, and Wilfried Steiner. IEEE 802.1Qbv gate control list synthesis using array theory encoding. In *Proc. Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2018.

19 Wilfried Steiner. An evaluation of SMT-based schedule synthesis for time-triggered multi-hop networks. In *Proc. RTSS*. IEEE, 2010.

20 Wilfried Steiner, Günther Bauer, Brendan Hall, and Michael Paulitsch. TTEthernet: Time-Triggered Ethernet. In Roman Obermaisser, editor, *Time-Triggered Communication*. CRC Press, August 2011.

21 Domiţian Tămaş-Selicean, Paul Pop, and Wilfried Steiner. Synthesis of communication schedules for TTEthernet-based mixed-criticality systems. In *Proc. CODES+ISSS*. ACM, 2012.

22 Domiţian Tămaş-Selicean, Paul Pop, and Wilfried Steiner. Synthesis of communication schedules for TTEthernet-based mixed-criticality systems. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 473–482, 2012.

23 Domiţian Tămaş-Selicean, Paul Pop, and Wilfried Steiner. Design optimization of TTEthernet-based distributed real-time systems. *Real-Time Systems*, 51(1):1–35, 2015.

24 Daniel Thiele, Philip Axer, and Rolf Ernst. Improving formal timing analysis of switched Ethernet by exploiting FIFO scheduling. In *Proceedings of the 52nd Annual Design Automation Conference*, page 41. ACM, 2015.

25 Qinghan Yu and Ming Gu. Adaptive group routing and scheduling in multicast time-sensitive networks. *IEEE Access*, 8:37855–37865, 2020.

26 Luxi Zhao, Paul Pop, Qiao Li, Junyan Chen, and Huagang Xiong. Timing analysis of rate-constrained traffic in TTEthernet using network calculus. *Real-Time Systems*, 53(2):254–287, 2017.

27 Zhong Zheng, Feng He, and Huagang Xiong. Routing optimization of Time-Triggered Ethernet based on genetic algorithm. In *2020 AIAA/IEEE 39th Digital Avionics Systems Conference (DASC)*, pages 1–8. IEEE, 2020.