Formal Verification of the FTTRS Mechanisms for the Consistent Update of the Traffic Schedule

Daniel Bujosa
Dept. de Matemàtiques i Informàtica
Universitat de les Illes Balears
Palma de Mallorca, Spain
daniel.bujosa@uib.es

Sergi Arguimbau
Dept. de Matemàtiques i Informàtica
Universitat de les Illes Balears
Palma de Mallorca, Spain
sergi.arguimbau@uib.es

Patricia Arguimbau
Dept. de Matemàtiques i Informàtica
Universitat de les Illes Balears
Palma de Mallorca, Spain
patricia.arguimbau@uib.es

Julián Proenza
Dept. de Matemàtiques i Informàtica
Universitat de les Illes Balears
Palma de Mallorca, Spain
julian.proenza@uib.es

Manuel Barranco
Dept. de Matemàtiques i Informàtica
Universitat de les Illes Balears
Palma de Mallorca, Spain
manuel.barranco@uib.es

***Abstract*— Critical Adaptive Distributed Embedded Systems (ADESs) are nowadays the focus of many researchers. ADESs are envisioned to dynamically modify their behavior to support changes of their real-time and dependability requirements at runtime as the conditions of the environment in which they operate vary. To provide ADESs with an adequate communication infrastructure, our research group proposed the *Flexible-Time-Triggered Replicated Star* (FTTRS). FTTRS provides highly reliable communication services on top of Ethernet, while keeping the adaptivity benefits that the *Flexible-Time-Triggered* (FTT) communication paradigm offers from a real-time perspective. This paper formally verifies, by means of model checking, the correctness of the mechanisms FTTRS includes to enforce consistent changes of the communication scheduling at runtime.**

***Keywords*— adaptivity, real-time control network, reliability, replica-determinism, FTTRS, formal verification, UPPAAL**

I. INTRODUCTION

Critical Adaptive Distributed Embedded Systems (ADESs) are real-time and dependable control systems that are expected to play a key role to appropriately interact with physical systems whose operational conditions change at runtime [1,2], even in unpredictable manners.

Changes in the operational conditions e.g. a shift from a given mission phase to another one, a change in the type of terrain an autonomous vehicle has to drive through, an increase of the bit error rate when the environment becomes more harshly, etc., may modify the requirements an ADES has to meet from the point of view of its real-time response and its dependability. For instance, a change in the operational conditions may require to shorten the deadlines of the tasks to be executed and/or the messages to be sent, as well as to increase the probability with which a message is guaranteed to be consistently delivered. Therefore, an ADES has to dynamically adjust its behavior so as to appropriately fulfill the new requirements, e.g. re-schedule the traffic, increase the number of times a message is proactively retransmitted, etc.

In particular, to support this adaptivity, the communication network an ADES relies on has to provide both *real-time and operational flexibility*. Real-time flexibility consists in supporting different types of real-time traffic (hard, soft, periodic and aperiodic), whereas operational flexibility stands for the ability to support changes in the traffic and its real-time requirements without interrupting the communication services. To the authors' best knowledge, the only highly reliable network that supports these two types of flexibility is the *Flexible-Time-Triggered Replicated Star* (FTTRS) [3]. FTTRS is the most reliable implementation of the *Flexible-Time-Triggered* (FTT) paradigm [5] on top of Ethernet.

FTT is a master multi-slave publisher-subscriber paradigm, where the so-called FTT master organizes the communication as a sequence of rounds called *Elementary Cycles* (ECs). The EC starts by the FTT master broadcasting what is called the *Trigger Message* (TM), which allows slaves to synchronize with the master and which polls the periodic messages that (the corresponding) slaves have to transmit in that EC. The time dedicated to transmit the TM is known as the *TM Window* (TMW). The TMW is followed by the *Synchronous Window* (SW), in which the appropriate slaves transmit the polled periodic messages. The last part of the EC is the *Asynchronous Window* (AW), where slaves transmit aperiodic messages.

FTT by itself does not provide high reliability and thus, cannot be used to build systems with that kind of requirements. To overcome this limitation, the Flexible-Time-Triggered Replicated Star (FTTRS) aims at adding fault tolerance mechanisms on top of FTT. The FTTRS architecture is based on the Hard Real-Time Ethernet Switching (HaRTES) [6], a switch-Ethernet implementation of FTT in which all the slaves are interconnected by means of a custom software-implemented switch embedding the FTT master.

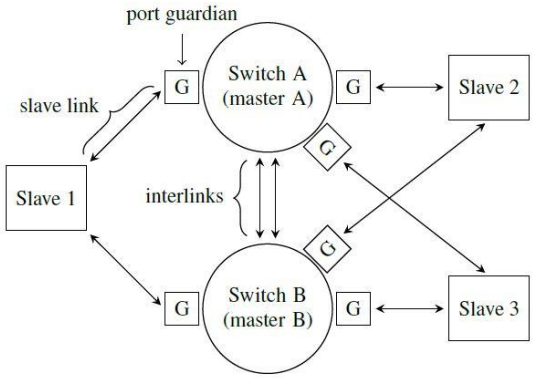

Fig. 1. FTTRS architecture, reprinted from [4]

Figure 1 shows that FTTRS basically consists of a duplicated full-duplex Ethernet star in which each HaRTES switch embeds an FTT master. Each slave connects to each switch by means of a link composed of a separated uplink and downlink. Both switches are interconnected by means of several full duplex interlinks, through which they exchange the TMs and all the traffic they receive from the non-faulty slaves.

FTTRS deals with permanent and temporary *non-malicious operational hardware faults* [7]. First, thanks to its redundant paths, FTTRS tolerates permanent faults affecting the switches and the links (slave links and interlinks). Second, in FTTRS each critical message – periodic or aperiodic – can be proactively retransmitted to timely tolerate temporary faults affecting the links. In particular, each FTTRS master always proactively retransmits several replicas of the TM to guarantee its successful broadcast.

Faults can manifest arbitrarily, however each switch/master is internally duplicated to exhibit crash failure semantics. Also, each switch/master includes a *Port Guardian* (PG) per slave to contain the errors sent by that slave. In this way, faulty slaves are perceived by non-faulty ones as omitting messages or sending messages with an incorrect (application) payload. Finally, thanks to the *Frame Check Sequence* each Ethernet frame includes, faults in the links can only corrupt frames that then are discarded at the receiving Ethernet interfaces.

To appropriately provide fault tolerance, both masters/switches of FTTRS must act as if they were a single one, i.e. they must be *replica determinate* [8]. FTTRS includes several mechanisms to enforce this replica determinism [3]. For instance, from a time point of view, both masters isochronously (with a fixed intertransmission time) broadcast several replicas of the same TM in lockstep (quasi-simultaneously). On the one hand, this allows both masters to be synchronized with each other, since they also exchange the TMs through the interlinks. On the other hand, this allows each slave to reliably synchronize with the beginning of each EC.

Of all the FTTRS's replica determinism mechanisms, in this paper we are interested in the ones that FTTRS includes to consistently update the traffic schedule at runtime. This is so because these later mechanisms are fundamental for FTTRS to provide high reliability while keeping the most distinguishing advantage of FTT, i.e its real-time operational flexibility. In this sense note that [3] argues for the correctness of these FTTRS's consistent schedule update mechanisms. However, since these mechanisms are quite complex, the use of formal methods provides an appropriate way to check their correctness. Thus, in this paper we model and formally verify these complex mechanisms by using a model checker called UPPAAL, which is specially suited for real-time systems.

Next, in Section II, we outline the FTTRS's schedule consistent update mechanisms; followed by Section III, which provides a pseudocode and a set of figures to complement the previous section. Then, in Section IV, we introduce the UPPAAL model checker and its formalism; for later, in Section V, to explain how we modelled these consistency mechanisms with this tool. Once the model is presented, in Section VI we proceed to its verification; and finally, in Section VII, we conclude the paper.

II. SCHEDULE CONSISTENT UPDATE MECHANISMS OF FTTRS

In this Section we summarize the mechanisms that FTTRS provides to consistently update (at runtime) the traffic scheduling. A thorough description of all consistency mechanisms of FTTRS can be found in [4].

First of all let us briefly explain the schedule update mechanisms of FTT. On the one hand, note that in the original FTT there is just one master. The master stores, within the so-called *Systems Requirements Data Base* (SRDB), the real-time parameters (period, deadline, minimum interarrival times, publisher, subscribers, etc.) of every *stream*, i.e. of every message. Similarly, each slave stores, within its own *Node Requirements Data Base* (NRDB), the real-time parameters of just the streams it is the publisher and/or the subscriber of. On the other hand, when a slave wants to request a schedule update, e.g. to modify the period of a stream it is the publisher of, it does so by sending its *update request* to the master within a *Slave Request Message* (SRM). When the master has one or more update requests pending to be processed, it selects one of them to be processed next and subjects it to an *admission control* procedure that decides whether or not the request can be accepted and, if so, how the content of the SRDB and the NRDBs must be modified. Afterwards the master broadcasts a so-called *Master Command Message* (MCM). This message informs the slaves about the decision and, in case the update has been accepted, how to update the NRDBs accordingly. Finally, the master and the slaves update (commit the changes in) their databases and, from then on, the TM the master broadcasts reflects the new schedule.

Note that the above FTT mechanisms allow to consistently update the SRDB and the NRDBs in FTT as long as all slaves receive every MCM. Thus, one of the aspects FTTRS has to guarantee is that all slaves receive every MCM even in the presence of faults. This can be achieved by proactively retransmitting the MCMs, as it will be explained later.

However, to consistently update the schedule in FTTRS, it is further needed to enforce that its two FTT masters both consistently carry out the admission control and consistently update their SRDBs. To achieve this, first it is necessary to make sure that both masters are *internally replica determinated* [8], i.e. that they produce the same outputs (admission control result and SRDB/NRDBs updates) as long as they are provided with the same inputs (update requests). FTTRS fulfills this by implementing the masters using the same internal hardware and software constructs, and by preventing any kind of internal non-determinism. Second, it is necessary to enforce that both masters are *externally replica determinated* [8], i.e. that they are provided with the same inputs (update requests).

Enforcing masters' external replica determinism is specially challenging. Note that in FTTRS a slave that sends an SRM transmits it to each one of the switches. In addition, each switch forwards to the other one said SRM. Thus, in principle, both masters should receive at least one copy of any SRM. Unfortunately, due to different combinations of permanent/transient faults in the links, interlinks and/or slaves, it is possible that one or more SRMs are received by one master only. If this happens both masters will have a different view of which update requests are pending, and this can

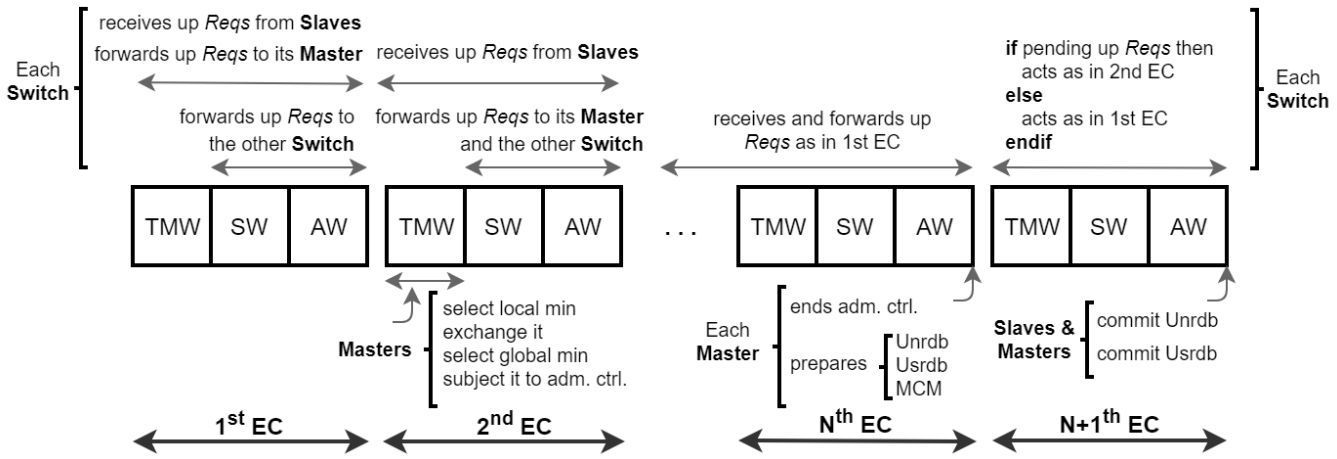

Fig. 2. Timeline of the actions carried out in FTTRS to consistently update the schedule

eventually lead their SRDBs to become inconsistent. For instance, if one master has one pending update request whereas the other one has none, then only one master will carry out the actions needed to process that update request. Similarly, if each master has a different set of pending update requests, each master can select a different request to be processed next.

Next, we sketch how FTTRS masters enforce the necessary masters' external replica determinism and, also, how they interact with the slaves to consistently update the schedule system wide. For the sake of clarity Figure 2 depicts the timeline of the actions FTTRS carries out in this sense.

We refer both FTTRS masters to as *A* and *B*; their SRDBs to as $SRDB_A$ and $SRDB_B$ respectively; and the NRDB of each slave, S_i, to as $NRDB_i$, where $i \in [1, N]$ and N is the total number of slaves. Also, respectively for each master, we refer its list of pending update request to as Q_A and Q_B, and the update request to process next as N_A and N_B.

Let us assume that, initially, no update request is pending nor has to be processed next, i.e. $Q_A = Q_B = N_A = N_B = \{\}$. Slaves can send SRMs whenever they want. Let us consider that each switch receives several SRMs directly through its uplinks during the TMW, SW and AW of the 1st EC. During this same EC the SRMs are forwarded as follows. On the one hand, each switch forwards to the other switch the SRMs it directly receives from its slaves. Note that since during the TMW both switches exchange a set of replicated TMs to appropriately synchronize with each other, each switch shapes the traffic so as to forward these SRMs during the SW or AW. On the other hand, each switch internally forwards to its own master the SRMs it directly receives from its slaves and the ones it receives from the other switch. In this case, since the forwarding is done internally, the switch immediately forwards any SRM to its master independently of which is the current window (TMW, SW or AW).

At this point consider that, due to different faults, both switches have received a different set of SRMs by the end of the 1st EC and, thus, $Q_A \neq Q_B$. For instance, $Q_A = \{1, 2\}$ and $Q_B = \{3, 4\}$. Note that each update request is identified by an integer that specifies its *intrinsic priority total order* with respect to the other update requests. This intrinsic priority total order is needed for the masters to decide which one of any two given requests has a higher priority. This total order can be implemented in several ways, but how to do it is irrelevant for the current discussion.

At the beginning of the 2nd EC each master has to select which update request to process next. However, since $Q_A \neq Q_B$, masters have to reconcile their view of the pending update requests so as to consistently select the same one. For doing so masters proceed as follows. First each master selects its *minimum update request*, i.e. the one with the highest priority, from its own Q list. We refer each one of these two selected update requests to as the *local minimum* of the corresponding master. We denote them as $\min\{Q_A\}$ and $\min\{Q_B\}$. In this example, $\min\{Q_A\} = 1$ and $\min\{Q_B\} = 3$. Then each master piggybacks its local minimum within the replicated TM it generates for triggering the 2nd EC. Since each switch does not only send each TM replica to its slaves, but also to the other switch, this strategy guarantees that each switch successfully receives the local minimum of the other switch. In this sense, by the end of the TMW of the 2nd EC each master updates its own Q list with the other switch's local minimum. Thus, $Q'_A = \{1, 2, 3\}$ and $Q'_B = \{1, 3, 4\}$. Next each master selects the minimum request from its updated list. Since $Q'_A = Q_A \cup \min\{Q_B\}$ and $Q'_B = Q_B \cup \min\{Q_A\}$, then it follows that $\min\{Q'_A\} = \min\{Q'_B\}$. In this example $\min\{1, 2, 3\} = \min\{1, 3, 4\} = 1$. In other words, at the end of the TMW both masters agree on the minimum pending update request to process next, i.e. $N_A = N_B = \{1\}$. We refer this update to as the *global minimum*. Then they consistently and simultaneously subject that global minimum to admission control, and remove it from their Q lists, i.e. $Q''_A = \{2, 3\}$ and $Q''_B = \{3, 4\}$.

It is noteworthy that for masters to agree on the global minimum, it is further necessary that no Q list is updated with any new update request - sent by the slaves - during the TMW in which the masters carry out the just described reconciliation actions, i.e. during the TMW of the 2nd EC in our example. Each master can accomplish this by storing, within an auxiliary list, any new update request its switch internally forwards to it during that TMW. Only when that TMW ends, the master transfers any new request to its Q list.

Also, note that if there are multiple pending update requests, FTTRS processes them in a stepwise manner. In other words, masters do not select nor process any pending update request until the current one has not finished the admission control. In our example this means that masters will not select and exchange a new local minimum until they decide on the current subjected request, i.e. on {1}.

Coming back to the point at issue, note that the admission control can take one or more ECs to finish. Both masters are configured to know what the worst-case execution time of the admission control is, so that both of them conservatively consider it as finished in the same EC. Let us assume they consider the admission control of the update request subjected in the 2nd EC to finish at the Nth EC. Thus, by the end of that EC each master builds up a *Master Command Message* (MCM) to notify (in the next EC) the slaves about the result of the admission control. In addition, if the admission control has accepted the request, each master both includes within the MCM the changes to be applied to the NRDBs and temporarily stores the changes to be applied in its SRDB. We denote these changes to be applied to the NRDBs as *Unrdb*, and the ones to be applied to the SRDB as *Usrdb*. Note that each master has its own copy of these updates, i.e. $Unrdb_A$ and $Usrdb_A$ in A, and $Unrdb_B$ and $Usrdb_B$ in B; where $Unrdb_A = Unrdb_B$ and $Usrdb_A = Usrdb_B$.

Next in the N+1th EC the schedule is consistently updated across the whole system as follows. First, each master piggybacks the MCM (which includes *Unrdb* if needed) within the replicated TMs. In this way the MCM is proactively retransmitted, which guarantees that all non-faulty slaves receive the MCM at the TMW of the N+1th EC and, thus, that slaves are consistently informed about the result of the admission control and the corresponding NRDBs update. Second, in case the result of the admission control was positive, then all slaves and both masters commit the corresponding changes to the NRDBs and SRDBs simultaneously at the end of the N+1th EC. In this way the schedule is consistently updated across the system, so that both masters can trigger the N+2th EC following the new schedule and all slaves can correctly respond accordingly.

Finally, please recall that FTTRS processes update requests in a stepwise manner. Thus, since in our example there are multiple pending update requests, then in the TMW of the N+1th EC the masters will select, exchange, agree on and subject to admission control a new local minimum.

III. Pseudocode of the FTTRS schedule consistent update mechanism

In this Section it is presented a pseudocode and a set of images that complements the previous section. Each of the steps of the mechanism is represented in the set of images (Figure 3).

```
1st EC
   Each slave:
        (1) send update requests to the masters
   Each master:
        (2) forward update requests through
            interlinks to the other master
2nd EC
   Each master during TMW:
        (3) send TM with the local minimum
        (4) select global minimum
        (5) start admission control
From 3th EC to Nth EC
   Each master:
        (6) evaluate admission control
Nth EC
   Each master at the end of the EC:
        (7) end admission control
N+1th EC
   Each slave and each master:
        (8) update NRDB and SRDB, respectively
```

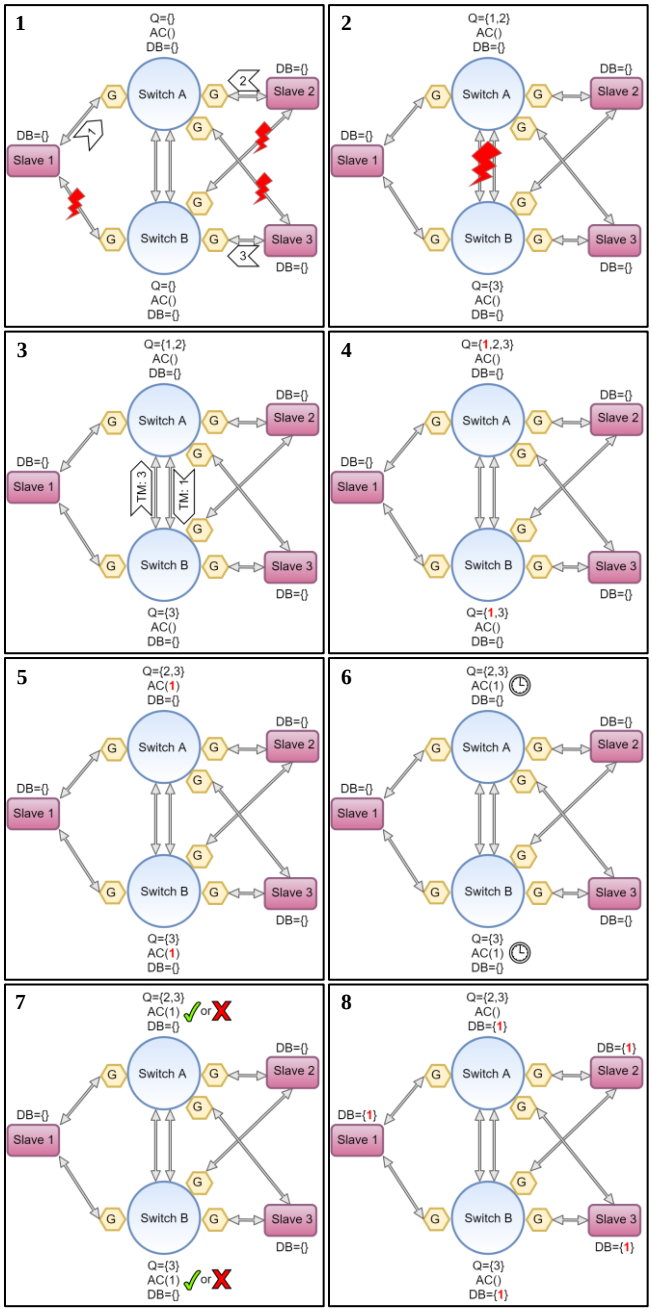

Fig. 3. Interactions between switches/masters and slaves during the FTTRS schedule consistent update mechanism

IV. UPPAAL MODEL CHECKER AND FORMALISM

UPPAAL is a model checker specially designed to formally model real-time systems and exhaustively verify their properties [9]. UPPAAL provides a formalism to model the system as a set of interconnected *timed automata*, i.e. finite-state machines extended with clocks (that progress at the same pace) [9], and a formal query language to express the system's properties to be verified. The model checker accepts the model and the queries as inputs and, then, explores all the possible execution paths of the model to exhaustively check whether or not each property holds. In case a property does not do so, the model checker shows a trace in which that property is violated. Next we summarize the formalism for the automata. Some hints on the query language will be provided in Section VI.

Each one of a model's automata is specified by means of a *template*, which can include parameters to create different instances of the same automaton. Basically, a template is constituted by a set of *locations*, *edges*, local variables and local clocks. The different templates can share global variables and clocks, and they can synchronize by using *binary* or *broadcast channels*.

Each automaton progresses through a set of locations, so that the state of the modeled system is defined as the current location of all automata (and the values of all variables and clocks). An automaton can step from one location to another one by taking an edge. The time a template remains in given location can be upper bounded by means of an expression, called *invariant*, involving one or more clocks. Moreover, a location can be defined as *normal*, *committed* or *urgent*. A template can indefinitely remain in a normal location, unless it is upper bounded by an invariant. Conversely, a template immediately leaves a committed or urgent location, thereby modeling that the time does not elapse in that location. A committed location differs from an urgent one in the sense that the former does not allow interleaving between different automata, whereas the later one does. As regards edges, each one can be enabled or disabled by means of an expression called *guard*, which is defined on variables and clocks. Moreover, an edge can include assignment expressions that are executed when the edge is taken.

Finally, templates can synchronize among them by simultaneously taking edges labeled with the same channel. A channel always has one sender template, but it can have one or more receiver templates. In the first case the channel is binary and the sender and the receiver wait each other to simultaneously take the edge. In the second case the channel is a broadcast one; receivers wait for the sender, but the sender can take the edge even if no receiver is waiting there.

V. MODEL OF THE FTTRS SCHEDULE CONSISTENT UPDATE

We modeled the schedule consistent update mechanisms of FTTRS by means of a *slave template* and a *switch/master template*, which respectively model the relevant actions carried out by a slave and by a switch/master. The slave and switch/master templates are instantiated three and two times respectively. In this way the model is composed by three slaves (justification below), and two switches/masters, since FTTRS presents these two as a way to achieve fault tolerance.

The reason to instantiate three slaves is because this number is the minimum that originates all kinds of scenarios in the queues of the replicated masters. Although it is possible to create inconsistencies in the queues with simply one or two slaves, the presence of three slaves allows having different and common update requests simultaneously in both masters. On the other hand, four or more slaves would create the same kind of scenarios that we obtain with three slaves. By scenarios we mean all possible distribution of update requests in the masters. However, given that there are two masters, those update requests can only be in three different situations (only in master A, only in master B or in both of them). The distribution of the update requests on any of these three situations is what determines the kind of scenario. In this sense, with only three update requests, i.e. with three slaves, it is possible to encompass all possible kind of scenarios. If more than three update requests are present, we would only increase the amount of update requests in one of the three possible situations, without creating new kinds of inconsistency scenarios.

Before starting with the details of each template, it is important to clarify that we made some assumptions in order to keep the complexity of the model at reasonable levels without compromising the accuracy of the model. First, we assumed that the system is always synchronized. That's because there is a set of mechanisms in FTTRS that guarantee this condition. These synchronization mechanisms are the Elementary Cycle Synchronization between Masters [11] and Slave Elementary Cycle Synchronization [12]. Secondly, we assumed that the TM sent from one switch to the other always reaches its destination, thanks in part to the capacity of FTTRS to tolerate faults even in the TMs [13] and to the proper management of the aforementioned replication of the TMs [14].

A. Slave template

Figure 4 depicts the *slave template*. Initially the slave is in the *idle* location (represented as two concentric circles). The slave steps to location *TM_recv* when masters end broadcasting the TMs, i.e. when the TMW ends. Note that in FTTRS masters quasi-simultaneously transmit the TMs during the TMW. However, TM transmissions may suffer from jitter and transient faults, and thus a slave can actually synchronize with either master A or B. To reflect this fact the slave can step to *TM_recv* through two different edges, each labeled with a different broadcast (UPPAAL) channel, namely *endTM_A* and *endTM_B*, whose sender is one of the masters.

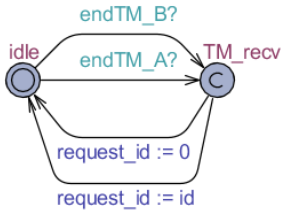

Fig. 4. Slave template in UPPAAL

Once in *TM_recv*, which is a committed location, the slave immediately comes back to *idle*. The slave does so by taking one of two different edges in a non-deterministic manner. The upper edge models the slave not sending any update request,

whereas the bottom one models the opposite. In this sense note that the slave template defines a global integer variable, *request_id*, that represents the update request the slave sends during the current EC. When the template is instantiated, this variable becomes *request_1*, *request_2* or *request_3*, depending on whether it is instantiated to represent what we understand as *slave 1*, *slave 2* or *slave 3* respectively. Keeping this in mind please note that when the slave decides not to send an update request it sets its corresponding *request_id* to *0*. Otherwise, it sets this variable to *1*, *2* or *3* depending on which one of the three slaves it represents.

It is worth recalling that we instantiate only three slaves. Also, each slave is modeled to send a maximum of one update request per EC, and when it does so it always specifies the same one. Thus, at any EC, each master can have a quantity of pending update requests ranging from *0* (no pending update request) to *3*. This strategy allows preventing the state space to explode, while being enough to verify the consistency. On the one hand, what really matters in this sense is to check that the schedule is consistently updated even if the set of update requests received by master A differs from the set received by master B. On the other hand, having three slaves is enough to generate scenarios in which each master has to select its local minimum among two or more update requests, i.e. scenarios in which this selection in both masters is not trivial.

B. Switch/master template

Figure 5 shows the *switch/master template*. Its initial location is *endTMW*, which represents the end of the TMW. From this location the master synchronizes with the other one (and the slaves) by acting either as a sender of the broadcast channel *endTM_id* or as a receiver of the broadcast channel *endTM_other*. Note that if the template represents master A, then *endTM_id* and *endTM_other* are respectively instantiated as *endTM_A* and *endTM_B*; and vice versa when the template represents master B.

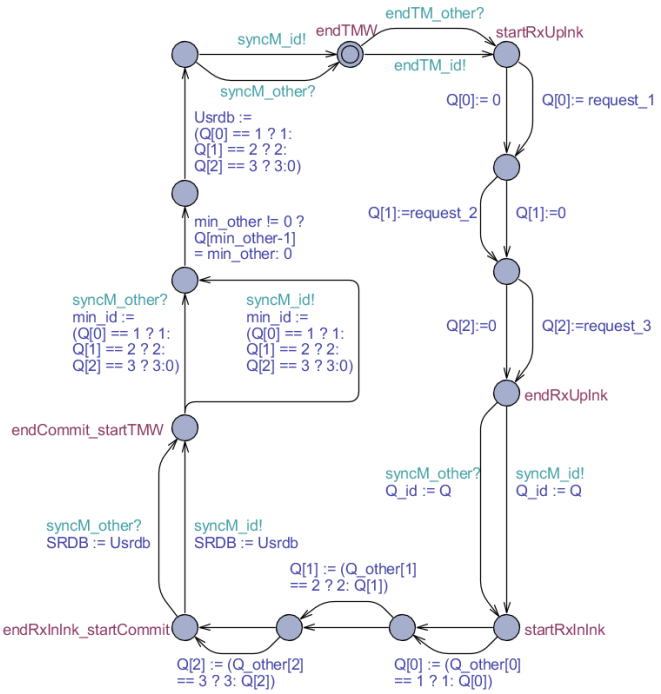

Fig. 5. Switch/master template in UPPAAL

The edges from location *startRxUplnk* to *endRxUpln* model the master successfully/unsuccessfully receiving the update request each slave sends during the current EC through its uplink. For instance, from *startRxUplnk*, the template non-deterministically takes the right or the left-handed edge. The right-handed one represents the master successfully receiving the update request slave 1 sends (if so), whereas the left-handed one models the master failing to receive that slave update request. Note that the master has a global array, instantiated as *Q[]* from its point of view, which represents its local Q list. Each one of the three positions of this array, namely *Q[0]*, *Q[1]* and *Q[2]*, are devoted to storing the update request sent by slave 1, 2 and 3 respectively. Note that in case a slave, *Si*, sends its update request and the master successfully receives it, then this master sets its *Q[i-1]* to value *i*; otherwise the master sets its *Q[i-1]* to *0*. Also note that we abstract away the instants of time, within the EC, in which the master receives the update requests. Whether the master receives an update request during the TMW, SW or AW is irrelevant.

After modeling the unsuccessful/successful reception of the above update requests, the template steps into location *endRxUplnk*. Each one of the two outgoing edges from this location are labeled with a binary channel, i.e. *syncM_id* or *syncM_other*, which are respectively instantiated as *syncM_A and syncM_B* for master A and vice versa for B. They are used to force both instances to wait for each other, similarly to how it is done at location *endTMW*. Afterwards, the edges from *startRxInlnk* to *endRxInlnk_startCommit* model the master unsuccessfully/successfully receiving (in a non-deterministic manner) the update request of each slave the other master forwards through the interlinks. For instance, the upper edge from *startRxInlnk* models the master failing to receive the update request of slave 1 the other master forwards. The bottom edge models the opposite situation. In this later case the position of *Q[]* that corresponds to slave 1, i.e. *Q[0]*, is set to *1* if the value stored in *Q_other[0]* is *1* (otherwise *Q[0]* keeps its own value, i.e. *Q[0] := Q[0]*). *Q_other[]* is the global array that, from the point of view of the master, instantiates the local *Q* list of the other master.

Again, analogously to what (and why) we do with the update requests the master receives through the uplinks, we abstract away the instants of time, within the EC, in which the master receives these forwarded update requests.

Once an instance of the switch/master steps into *endRxInlnk_startCommit* it waits for the other instance by using, again, the binary channels *syncM_id* and *syncM_other*. This is done to model that, at the end of the EC, both masters simultaneously commit a SRDB update if needed. As it will be explained, an update request selected as the global minimum is always modeled as being accepted by the admission control. When so the template assigns that update request's identifier to the local variable *Usrdb*. Thus, when simultaneously exiting from *endRxInlnk_startCommit*, each template sets its local variable *SRDB* to the value of its *Usrdb*. As a result, if a global minimum was subjected to admission control, then the templates sets *SRDB* to 1, 2 or 3 depending on the update request selected as global minimum. Otherwise, SRDB is set to *0*, meaning that no commit needs to be carried out.

Location *endCommit_startTMW* represents the beginning of the TMW of the next EC. From there to *endTMW* each step models a different action to be carried out during the TMW. The first step models each one of the two master instances simultaneously selecting its local minimum. Each instance stores its minimum at the global variable *min_id*, i.e. *min_A* or *min_B* depending on whether it is the instance of master A or master B respectively. Note that if there is a pending update request in its local *Q* list, the instance will select (and assign it to its *min_id*) the one with the lowest identifier, i.e. the one with the highest priority. Otherwise, it assigns *0* to its *min_id*. The second step models masters exchanging their local minimum. Since each master piggybacks its local minimum within the TMs it transmits, and at least one TM always reaches the other switch, each master always successfully receives the local minimum of its counterpart. Specifically, the master consults the other master's local minimum by accessing the global variable *min_other*, which is instantiated as *min_B* or *min_A* for master A and B respectively. When the other master has a pending update request *min_other* will be *1*, *2* or *3* depending on the update request that other master selected as its local minimum. If so the template assigns *min_other* to the position of its *Q* list devoted to allocate that update request identifier, i.e. *Q[0]*, *Q[1]* and *Q[2]* respectively. The third step models the master selecting the global minimum and subjecting it to admission control. As explained before, we assume that the admission control always accepts the subjected global minimum. Note that when a subjected global minimum is not accepted, neither the masters nor the slaves modify the SRDBs and NRDBs. Thus, from the point of view of the consistency among the data bases, not accepting a global minimum is equivalent to not having selected any global minimum; which is modeled as not having any pending update request when the EC starts, i.e. *min_id = min_other = 0*.

Also note that we model neither the preparation of the MCMs nor the commit slaves carry out at their NRDBs. Please recall that each master piggybacks the NRDB updates within the TMs it sends. Thus, since each slave receives at least one TM from at least one master, then all slaves will consistently update their NRDBs as long as the masters consistently update their SRDBs.

Finally, it is worth highlighting that we immediately set *Usrdb* to the global minimum, so that the template will commit *Usrdb* to *SRDB* at the end of the current EC, i.e. we assume the admission control to finish in the same EC in which it begins. Coming back to Figure 2, this means that, from a timeline point of view, we collapse the ECs from the 2nd to the N+1th one (both inclusive) into a single EC. This can be done without losing generality. On the one hand, this is because in this single EC both masters synchronously carry out the same sequence of actions - devoted to select, analyze and commit just one update request - they would synchronously perform throughout the collapsed ECs. On the other hand, the differences between the actions a switch carries out in a collapsed EC and the actions it carries out in another collapsed EC are the instants in which it forwards the update requests to its master; which as explained before can be abstracted away.

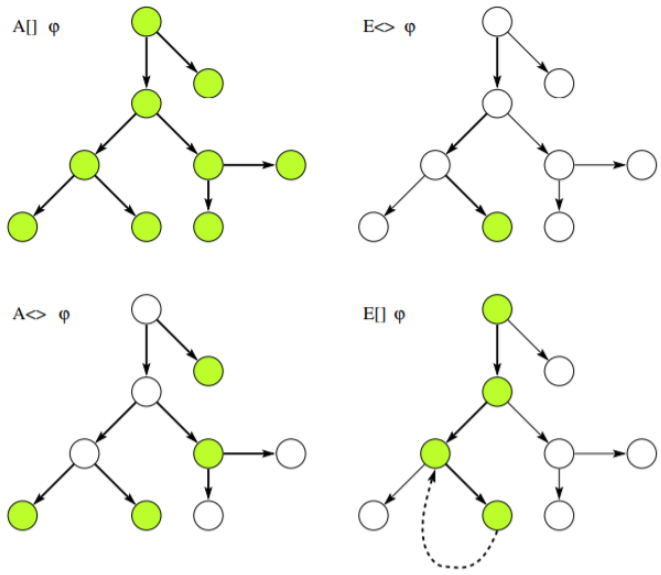

Fig. 6. Path formulae supported on UPPAAL, reprinted from [10]

VI. Model Verification

As said before, UPPAAL provides a formal query language to express the properties to be verified. In UPPAAL there exists three different kinds of properties [10]: *reachability* (uses the expression *E*<>), *safety* (uses the expressions *A[]* and *E[]*) and *liveness* (uses the expression *A*<>). In Figure 6 we can see the paths that these properties check schematically. In the figure, the circles represent states. Each state is, as indicated above, the current location of all automata and the values of all variables and clocks. Furthermore, the green circles represent states in which the state formula (φ) is met. Thus, path formulae check certain distribution of states in which the state formula meets, as we will explain below.

The reachability properties (*E*<>) are the simplest ones since they ensure that the model can reach a certain state eventually. The safety properties check two different kinds of scenarios: the first one (*A[]*) ensures that a certain condition is always fulfilled; the second one (*E[]*) ensures that it always exists a path where certain condition is always fulfilled in all states of the path. Lastly, the liveness properties (*A*<>) ensure that, taking any path it will reach always a certain state, for example, in a communication network transmitting a message will eventually reach its destination whatever happens.

These properties also require a state formula (φ) to be indicated. The state formula specifies certain conditions in the model, for instance, *i == 7* is true when the variable i equals 7. In this regard, depending on the code that we write for the state formula, we can check different properties in our model. For example, if we introduce the query *A[] i <= 7* we are verifying if the variable i is always equal or lower than 7. A special kind of state formula is *deadlock*, which is true when the model is blocked, that is to say, it does not have any chance to change its state.

We verified the correctness of the FTTRS's schedule consistent update mechanisms by checking three properties.

First, we verified the following *safety* property to check that the mechanisms do never lead to a deadlock: ***A[] not deadlock***. This property claims that "in every state, i.e. *[]*, of all reachable paths, i.e. *A*, it always holds that there is no deadlock". Moreover, note that when two edges exit from a given location to model two opposite situations, e.g. to model whether or not a master receives a given message, the model non-deterministically selects one of those edges. In this sense, since the model iteratively progresses in an infinite loop, this property also checks that the model does generate all possible combinations of the different non-deterministic choices.

Second, we checked that both SRDBs are always consistent. For this, we verified that the following *safety* property holds: ***A[] MA.SRDB == MB.SRDB***, where MA and MB respectively represent the switch/master A and B.

Finally, to further check that the just mentioned property is not only fulfilled in trivial cases, i.e. not only when the SRDBs are not updated but also when they are, we used the following *reachability* property: ***E <> MA.SRDB != 0*** This property states that "it exists at least one state, i.e. <>, of at least one reachable path, i.e. *E*, in which the SRDB of master A (and of B due to the previous property) has been updated, i.e. MA.SRDB != 0".

These two conditions verify that the SRDB is updated and its content is always equal in both switches/masters, so they guarantee the correctness of the consistent schedule update mechanism. However, it is important to note that this is achieved under the assumptions of correct synchronization and correct exchange of the TM between the switch/masters.

VII. Conclusions

The *Flexible-Time-Triggered Replicated Star* (FTTRS) represents a step towards developing networks that appropriately support future critical *Adaptive Distributed Embedded Systems* (ADESs), as depicted in Figure 7. Thanks to FTTRS, the FTT communication paradigm leverages on top of Ethernet. Now it is not only possible to take advantage from the real-time and operational flexibility of FTT, but also from the high reliability FTTRS provides. FTTRS extends FTT on Ethernet by means of fault-tolerance capabilities based on different types of redundancy, which in turn require adequate mechanisms to enforce consistency among replicated components.

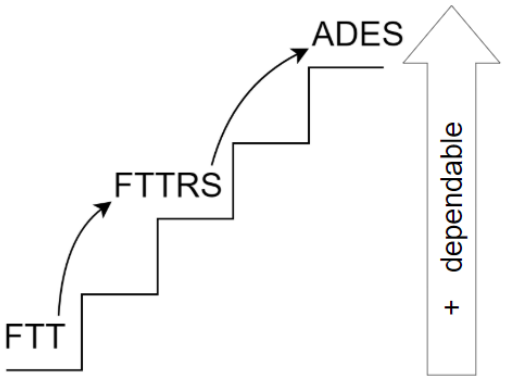

Fig. 7. Evolution step by step towards ADES

In this paper we formally verify the correctness of the most complex FTTRS's consistency mechanism, i.e. the one that guarantees that the traffic schedule is consistently updated at runtime. For this we use UPPAAL, a model checker specially designed for real-time systems. In this sense this paper is the first one of a series of works we plan to carry out to verify the correctness of ADESs based on FTTRS or similar networks.

Acknowledgements

This work is supported in part by the Spanish Agencia Estatal de Investigación (AEI) and in part by FEDER funding through grant TEC2015-70313-R (AEI/FEDER, UE).

References

[1] T. Brade, G. Jaeger, S. Zug, J. Kaiser. "Sensor and Environment Dependent Performance Adaptation for Maintaining Safety Requirements". In: *Computer Safety, Reliability, and Security*. Springer Int. Publishing, pp. 46-54, 2014.

[2] A. Beck, C. Lisbôa, L. Carro. "Adaptable Embedded Systems". Springer New York, 2013.

[3] D. Gessner, J. Proenza, M. Barranco, A. Ballesteros. "A Fault-Tolerant Ethernet for Hard Real-Time Adaptive Systems". *In: IEEE Transactions on Industrial Informatics*, 2019.

[4] D. Gessner, J. Proenza, M. A. Barranco. "Adding Fault Tolerance To a Flexible Real-Time Ethernet Network for Embedded Systems". *PhD Thesis, Universitat de les Illes Balears*, 2017.

[5] P. Pedreiras and L. Almeida, "The flexible time-triggered (FTT) paradigm: an approach to QoS management in distributed real-time systems". *In: Proceedings International Parallel and Distributed Processing Symposium*. IEEE Comput. Soc, 2003.

[6] R. Santos, "Enhanced Ethernet Switching Technology for Adaptive Hard Real-Time Applications". *PhD Thesis, Universidade Aveiro*, 2010.

[7] Algirdas Avižienis et al. "Basic Concepts and Taxonomy of Dependable and Secure Computing". *In: IEEE Transactions on Dependable and Secure Computing*, pp. 11–33, 2004.

[8] S. Poledna. "Fault-Tolerant Real-Time Systems. The Problem of Replica Determinism". *The Springer International Series in Engineering and Computer Science*, Springer US, 1996.

[9] K. Larsen, P. Pettersson, W. Yi. "Uppaal in a nutshell". In: *International Journal on Software Tools for Technology Transfer*, Springer-Verlag, pp. 135-152, 1997.

[10] G. Behrmann, A. David, K.G. Larsen. "A Tutorial on Uppaal". *In: Formal Methods for the Design of Real-Time Systems. Lecture Notes in Computer Science*, vol 3185. Springer, Berlin, Heidelberg, 2004.

[11] A. Ballesteros, J. Proenza, D. Gessner, G. Rodríguez-Navas, T. Sauter. "Achieving Elementary Cycle Synchronization between Masters in the Flexible Time-Triggered Replicated Star for Ethernet". *In: Proceedings of the 19th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2014)*, Barcelona, Spain, 2014.

[12] D. Gessner, I. Álvarez, A. Ballesteros, M. A. Barranco, J. Proenza. "Towards an Experimental Assessment of the Slave Elementary Cycle Synchronization in the Flexible Time-Triggered Replicated Star for Ethernet". *In: Proceedings of the 19th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2014)*, Barcelona, Spain, 2014.

[13] D. Gessner, A. Ballesteros, A. Adrover, J. Proenza. "Experimental Evaluation of Network Component Crashes and Trigger Message Omissions in the Flexible Time-Triggered Replicated Star for Ethernet." *In: Proceedings of the 2015 IEEE World Conference on Factory Communication Systems (WFCS)*, Palma de Mallorca, Spain, 2015.

[14] D. Gessner, J. Proenza, M. A. Barranco. "A Proposal for Managing the Redundancy Provided by the Flexible Time-Triggered Replicated Star for Ethernet." *In: Proceedings of the 10th IEEE International Workshop on Factory Communication Systems (WFCS)*, Toulouse, France, 2014.