Computing Data Streams in Real-Time Networks from Component-Based Software Engineering

Maxime Samson, Thomas Vergnaud, Éric Dujardin, Laurent Ciarletta, Ye-Qiong Song

Computing Data Streams in Real-Time Networks from Component-Based Software Engineering

Maxime Samson*†, Thomas Vergnaud*, Éric Dujardin*, Laurent Ciarletta† and Ye-Qiong Song†
* Thales Research & Technology – Palaiseau, France
{maxime.samson – thomas.vergnaud – eric.dujardin}@thalesgroup.com
† LORIA – Université de Lorraine – Nancy, France
{maxime.samson – laurent.ciarletta – ye-qiong.song}@loria.fr

***Abstract*—Real-time networks are used by distributed embedded systems of increasing complexity. This leads to a need for precise design and configuration techniques for these networks.**

This paper presents an approach for the iterative configuration of a real-time network and the associated distributed real-time control application that is to be deployed on it.

The approach consists in combining network modeling and Component-Based Software Engineering techniques to automatically calculate the model of the data streams that will be exchanged over the network.

The paper explains how to calculate the parameters used in data stream models from the specification of the application architecture and behavior. A single specification is used to generate both the data stream models and the infrastructure code of the application, thus providing guarantee of consistency between the behavior of the actual system and the network configuration. The use of this approach is demonstrated on an example which uses a TSN network.

***Index Terms*—Network Modeling, Model-Based Software Engineering, TSN.**

I. Introduction

Modern distributed embedded systems have growing network communication needs. Many of these systems are used in critical environments such as avionics, industrial automation or in-vehicle networks. In such systems, real-time networks are commonplace and their size and complexity are ever increasing.

In order to manage this increasing system complexity, consistent and efficient design approaches are required. Both consistency and efficiency can be obtained through the use of model-based engineering methods. Such methods use representations of the system components, called models, to formally define the system and enable the automatic generation of a great part of what is needed to deploy the system: application code, network configuration, and simulation and analysis models for network simulators.

Automatic generation ensures consistency and efficiency by easing the use of an iterative design process. Whenever changes are made to the system, they only have to be done in the model and they will take effect in the entirety of what is generated. This saves time compared to having to make all changes manually and greatly reduces the risks of human error.

While it can be relatively easy to create models for most parts of the network, modeling the data streams can prove to be very challenging. The model of the data streams is the most essential part because it contains crucial information that is necessary for configuration synthesis, such as payload size, transmission periods and offsets. Without a formal specification of the application software behavior, this information can be hard to obtain with enough accuracy for the simulation to be realistic in a useful way.

A typical example is the scheduling of data streams using the time-aware shaper of TSN. Usual approaches assume a perfect knowledge of the data stream, and validate a network configuration (e.g. stream scheduling) using network simulators. However, data are produced by their corresponding application tasks that are executed on their host environment (hardware and software system). Any unexpected offset on a task execution would lead to extra delay or to deadline miss. In practice, it is hard to model data streams without a formal specification of the application software behavior as a whole.

One way of solving this problem and ensuring that data streams are accurately specified is to automate this process by leveraging Model-Based Software Engineering (MBSE) methods. An accurate modeling of the software architecture and execution semantics of the applications that use the network enables the automatic generation of the data stream characteristics.

Our work is based on the Unified Component Model for Distributed Real-Time and Embedded Systems (UCM) standard [1] to model software architectures. Our contributions are twofold: 1) We show that the combination of UCM and the sequence diagrams of UML is sufficient to describe the application behavior that allows in turn to extract the execution and communication semantics; 2) This enables the automatic computation of a model of the data streams the system produces at execution time. In addition, we show that the automatically produced data streams are directly used in our configuration generation tool, MoBACT [2], for validating TSN scheduling.

Combining software modeling and network modeling enables our approach to produce accurate simulation models. This reduces the design cost and increases the confidence in its correctness in many different ways: it reduces human error; learning how to create models for different simulation tools is no longer required; the technical code of the applications is generated to ensure consistency between the specification and

the actual execution of the system. Also, managing evolution in the system – either in the applications or the network – is easy and fast because we can consistently regenerate both the simulation models and the technical code that controls the execution of the application algorithms. Instead of assuming a perfect knowledge of the data streams generated by the different nodes of a distributed system, we obtain this knowledge directly from the application architecture.

In this paper, we apply our approach on an example TSN network using Time Aware Shaping [3] to meet the deadlines requirements for the time critical data streams. Using our previous works [2], we can use the automatically computed data stream models to generate simulation models for different tools (e.g. NeSTiNg [4], RTaW-Pegase[1]).

The remainder of the paper is organized as follows. The example system we consider for the paper is described in section II. Related works are presented in section III. Section IV lists the parameters required to characterize data stream models. Section V explains how we model software architectures. The automatic calculation of the data stream models from the software architectures is explained in section VI. Section VII demonstrates the benefits of our approach on an evolution of the system architecture. Section VIII concludes the paper and points out some future works.

II. Description of The Case Study

Described here is the example that we will use in the remainder of this paper, developed as part of the CPS4EU [5] project and based on a TSN network. Yet, our approach of automatic data stream model generation is agnostic of the network technology used in the system.

This example is a case of "Vehicle-to-Infrastructure" communication involving a wireless connection between a roadside unit and a vehicle. The roadside unit is able to send speed limit instructions to passing by vehicles; we assume this communication to be reliable because we are interested in the in-vehicle TSN network. Upon the reception of a speed limit instruction, vehicles must adjust their speed, whether it means slowing down or accelerating.

Fig. 1 shows the topology of the in-vehicle network as well as the roadside unit and the wireless communication. The driving control end-point (DC) computes the acceleration needed to attain the required speed and sends (1) the corresponding instructions to the motor system (MS), which then responds (2) with the current rotation speed of the car wheels. This is the time-critical loop of the system.

Messages from the roadside unit (RSU) are received by a wireless gateway (GW) which, in order to make the example more realistic, is a dedicated end-point as it needs to be close to the antenna in the car. The gateway then transmits (3) the new speed limitations to the DC. The dashboard end-point (DB) receives (4) the current speed of the car to display from the DC. The DB may send (5) speed limitation orders to the DC – this would correspond to driver's commands. Additional

[1]https://www.realtimeatwork.com/rtaw-pegase/

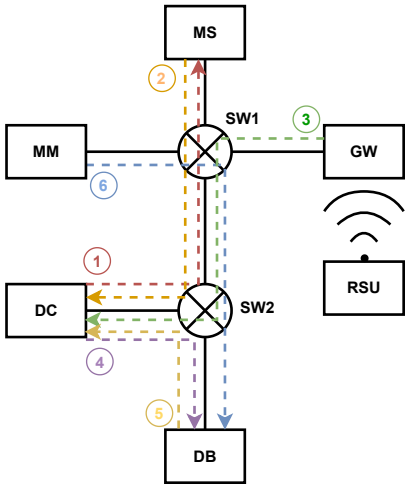

Fig. 1. Topology of a vehicle-to-infrastructure network

data transits (6) from a multimedia end-point (MM) to the DB to illustrate mixed-criticality; it may correspond to the display of GPS navigation instructions.

The interactions between DC and MS consist of data exchanges every 30 ms through streams 1 and 2. These streams are the most critical in the network. DC calculates speed orders every 30 ms; we set both stream deadline requirement to 10 ms. We assume that speed limitation instructions from the roadside unit, represented by stream 3, can be sent every 200 ms at most. Current speed information, represented by data stream 4, is sent from DC to DB every 200 ms. Stream 5 corresponds to the speed limitation orders sent from DB to DC with a period of 200 ms. Stream 6 has no specific deadline; it consists of large amount of data that shall not interfere with the control command streams.

The network must guarantee that the deadline requirements of streams 1 and 2 are always met. In order to reach this goal, these data streams will be scheduled by the Time Aware Shaper [3]. This scheduling mechanism is based on time division and guarantees that the transmission of critical data streams will not be hindered by other transmissions.

This scheduling mechanism requires an important configuration effort. Indeed, solving this scheduling problem is an NP-complete problem [6]. In order to solve it, a characterization of the system data streams is required. This data stream model needs to accurately represent the actual network communications, otherwise the synthesized configuration will not be able to respect the deadline requirements.

The software elements that are deployed on the end-points communicate using unicast UDP messages; data is serialized using CBOR [7]. This is a typical choice for embedded systems.

III. Related Works

Having to make use of approaches based on modeling when designing distributed critical systems, more specifically when designing the communication infrastructure of these systems,

has been identified in multiple works. This need stems from the increase in communication network complexity. Model-based approaches offer ways of managing this complexity by ensuring consistency across the requirements definition phase, the design phase and the deployment of the system.

In [8], a model-based approach to the design of space communication networks is presented. It proposes an integrated process used to model and simulate networks used in space communication applications and puts it into practice on a realistic use case. The paper highlights the necessity of using MBSE when designing such networks and insists on the precision the models must have for the obtained results to be relevant.

In [9], current challenges and potential solutions are explored, in the context of avionics systems. The paper draws conclusions similar to [8] and it also explicitly mentions TSN networks as a promising solution. It does not propose any approach but highlights the need for a model-based approach which ensures consistency.

Works that rely on model-based approaches for the design of embedded systems have been conducted, using various modeling languages such as MARTE [10], AADL [11] and AUTOSAR [12].

In [13], MARTE-compatible models are used to run simulations and schedulability analysis of software on embedded real-time systems. This model-based approach automatically generates system performance models, from the models of its individual components. This work, although it implements an approach which relies on the same principles as ours, is not targeted at designing real-time networks.

AADL is used in [14] to refine models of embedded real-time systems in the domain of avionics. This approach generates the configuration of AFDX networks using worst-case traversal time analysis, based on network calculus. Contrary to our approach, this work does not ensure consistency by also generating the code of the applications.

There are works which use the AUTOSAR standard to assist the design of embedded systems using a model-based approach. In [15], a framework for the design of a steer-by-wire system is presented. This framework uses the model-based approach defined in the AUTOSAR standard but it does not include a formal specification of application behavior. This means that it cannot provide the guarantee that data streams will behave as expected.

In summary, existing works have identified model-based approaches as essential to the design of embedded and distributed real-time systems, as their benefits are important for the trust we can have in the results of analysis and in the produced network configuration. To the best of our knowledge, however, none of these works would generate, from a single model, both the code of the application and the network configuration as a way of guaranteeing consistency. Some are also tied with a specific application domain and related technology. As stated in the introduction, we chose to use UCM for specifying both application behavior and generating application code, and because it is not specific to any domain or network technology.

IV. Data Stream Specification

Data streams are sequential transmissions of data across a network. In this section, we identify the required parameters to specify data streams in the context of real-time networks. Accurately modeling the data streams is especially important when dealing with real-time networks because their parameters are the ones used to produce the network configuration.

We consider two categories of data streams parameters: parameters that are directly linked to the application execution, and parameters that are system requirements. Parameters of the first category can be calculated, while those of the second category are under the responsibility of the system architect.

A. Application-Related Parameters

The following data stream parameters depend on the application; these are the ones we intend to automatically calculate:

- *Payload size*: The size of the data sent on every transmission of a data stream;
- *Traffic Class*: The category of traffic the data stream belongs to, e.g. control, video, audio;
- *Talker*: An end-point port, source of the data stream;
- *Listeners*: End-point ports, recipients of the data stream;
- *Period*: The transmission period of the data stream;
- *Offset*: The transmission offset of the data stream, which is relative to the start of the system.

The payload size may not be a straightforward parameter to obtain, especially if the payload puts together different pieces of data of different data types. Moreover, any change in the application architecture could lead to changes in the value of this parameter and would require the calculation to be redone.

The traffic class is assigned to data streams by the network architect. It can be used by switches to select which frame to transmit when multiple frames are awaiting their transmission and is used by scheduling mechanisms to grant data streams access to network resources in a tighter or fairer way. The talker is the source of the communication and the listeners are the communication destinations. Such information must be specified in the software architecture, as it impacts the way network sockets are configured.

The period and the offset parameters dictate the timing aspect of data streams. The transmission period parameter depends on the execution periods of the software components that produce the network communications. The transmission offset is both difficult and important to compute accurately. This parameter depends on the computation time the software components take before transmitting data. Configuring a network using wrongly computed offsets leads to increased latency because scheduling mechanisms, and especially those based on time division, expect frames to be made available by software components at specific times.

B. System-Related Configuration

The following parameters define the system requirements that apply to the data streams:

- *Paths*: The paths the data stream will follow when being transmitted over the network;

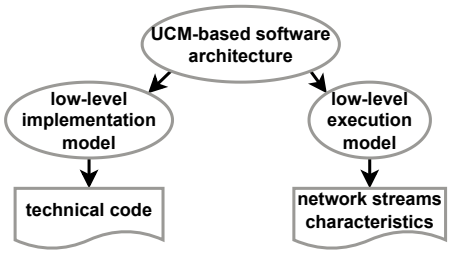

Fig. 2. Generation chain

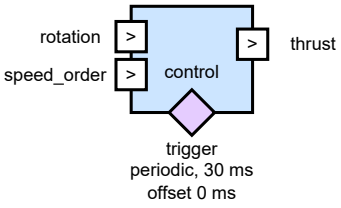

Fig. 3. Declaration of UCM component *control*

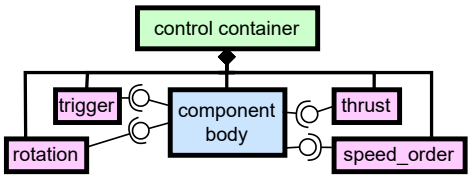

Fig. 4. UCM Container model

- *Deadline*: The maximum end-to-end latency the data stream must never exceed;
- *Maximum jitter*: The maximum amount of jitter the data stream must never exceed;
- *Time critical*: An indication on whether or not the deadline requirement of the data stream is especially critical.

The transmission paths of the data streams cannot be automatically computed using the model of the applications because it is related to the network architecture – not the software architecture. The deadline and maximum jitter are the requirements the network configuration must be able to uphold at all times. If a data stream is especially critical, it can be indicated and a configuration synthesis process for scheduling mechanisms can take it into account. Therefore, the values of these parameters are under the responsibility of the system architect because they do not depend on the application model but on the requirements of the system as a whole.

V. Modeling of the Software Architectures

We aim at extracting the data streams parameters from a model of the software application. We also, at the same time, aim at ensuring these stream parameters are consistent with the way the actual software behaves. It is therefore mandatory to maintain the consistency between the application code and the network stream specifications.

Our approach relies on the automatic generation of the application technical code (which controls execution periods, data transmissions over the network, etc.) and of the generation of the data streams parameters. Both generations use a unique model of the application as a starting point, which ensures the consistency between them. This is illustrated on fig. 2.

A. Software Component Specification with UCM

We use the Unified Component Model for Distributed Real-Time and Embedded Systems (UCM) as a basis for modeling software architectures. UCM [1] is an OMG standard that defines concepts such as *components* and *ports* to describe the functional (i.e. business) parts of a software architecture. The non-functional (i.e. technical) elements are also modeled: *connectors* describe interaction semantics between components; *technical policies* describe the interaction semantics between a component and its execution environment; in particular, task executions are typically managed by technical policies.

UCM components encapsulate algorithms. They define the *APIs* provided to and required from the encapsulated source code to interact with the outside of the component. This includes the *data structures* received and sent by the components.

The algorithmic code encapsulated in a UCM component should not perform task management or network communications. These matters are to be specified by technical policies and connectors in order to achieve a strict separation between functional aspects (i.e. the algorithms inside the components) and the extra-functional aspects (i.e. the execution and communication infrastructure materialized by technical policies and connectors). Fig. 3 illustrates the declaration of the driving control UCM component, with two input ports (rotation and speed_order), one output port (thrust) and one execution technical policy (trigger). The technical policy specifies that the component shall execute periodically every 30 ms, with an offset of 0 ms from the start of the application.

According to the UCM standard, the declaration of a component leads to the code structure illustrated on fig. 4. The component body is a class that contains the algorithmic code. The code for each connector and technical policy is produced by the code generator into separate C++ classes, thus isolating the algorithms from the technical code. A container class is also generated to manage the life cycle of the component body and the technical classes and connect them.

Because algorithmic code is encapsulated within the component body, it can be deployed in various configurations by changing the technical code classes that are connected to it by the container. Connectors and technical policies carry configuration parameters (e.g. task execution periods and offsets, socket configuration). As the technical code is generated from the component specifications, we ensure components executions and communication are consistent with the application specification. In the example, the component body of component *control* shall be executed every 30 ms.

B. Specification of Component Behaviors

UCM does not address the description of component behaviors: UCM components implementations do not carry any

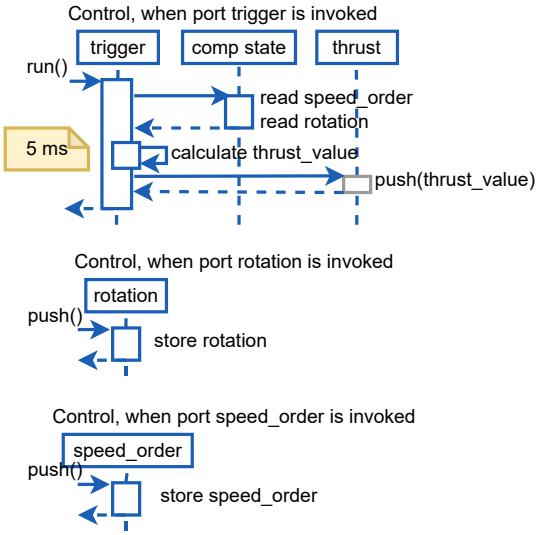

Fig. 5. Behavior specification of component *control*

behavior information. One specifies when a component is executed (triggered by a task configured within a technical policy, or upon the reception of a piece of data from a connector), but we cannot specify what happens then. In particular, we cannot specify when a component outputs data.

In [16], we explained how to complement the description of a UCM component implementation by adding behavior specification. We also explained how to check whether the algorithmic code fits the behavior specification. Such information can be represented by a *subset* of UML sequence diagrams to help software developers.

We specify the behavior of component *control* as illustrated in fig. 5. The first sequence diagram specifies the behavior associated with method `run()` to be called by the *trigger* technical policy. Method `run()` first retrieves the values of speed order and rotation stored in the component, then calculates the thrust value, then sends it by calling method `push()` of port thrust. The second and third sequence diagrams specify the behaviors associated with method `push()` of ports rotation and speed_order. The data is simply stored in the component state. The calculation time is 5 ms while the other steps (e.g. data read and write) are negligible.

We need to model the behaviors for the main processing chains of all the components. That is, the behaviors for the reception of *rotation* and *speed_order* in component *control* may have been left unspecified; the implicit assumed behaviors would be that the invocation of the reception ports does not trigger any calculation or communication.

The behavior of component *control* is very simple on purpose, to be easy to understand. Our approach supports the modeling of more complex behaviors, with loops, sequences and alternatives. The behavior semantics of technical policies and connectors are handled by the generator and are combinations of execution steps, as we explain in the following.

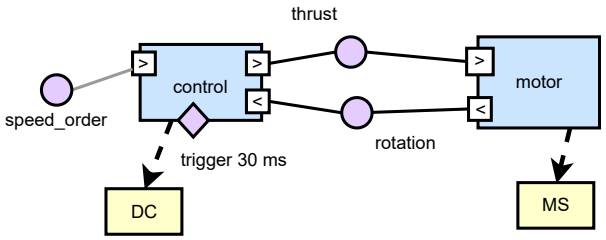

Fig. 6. Component deployment

C. Component Deployment on the Network Topology

In order to create the application architecture, the UCM components are connected through connectors and are deployed on network nodes, as illustrated on fig. 6. The technical code for the connectors is then produced by the generator from the connector semantics and the architecture deployment model (e.g. the data types to transmit through the connector, the component allocations on the network nodes, IP addresses of the network nodes, etc.).

Fig. 6 illustrates the connection of components *control* and *motor*. The two components are deployed on different network nodes, *DC* and *MS*, presented in fig. 1.

From this deployment specification, the code generator produces connector technical code that serializes the data into CBOR messages and sends them using unicast UDP sockets to the destination components. The generator also creates a semantic model of the execution flows that enable the calculation of the network communications (see section V-B).

From the component specification (fig. 3) and its behavior (fig. 5), the generator produces execution flows in the form of sequences of execution steps (fig. 7), which we detail now. Each execution step has the following semantics: it receives incoming execution flows, then performs its execution in a bounded time comprised between a best case execution time (BCET) and a worst case execution time (WCET), then produces outgoing execution flows. If an execution step is associated with a group of execution steps, this group is executed before producing the outgoing flows. It is possible to specify that its execution is triggered upon the reception of all the incoming execution flows, or only one of them; it may produce all its outgoing execution flows at once, or only one of them. Such specification enables the description of various execution semantics.

Technical policy *trigger* is translated by a group that contains a single execution step (*trigger*). This step receives an execution flow produced by a pattern named *periodic exec* handled by the generator. Step *trigger* is associated with the *component* group. Therefore its semantics is the following: upon the reception of the execution flow from the *periodic exec* pattern, step *trigger* executes, then step *run* executes, then *read* (the only successor of *run*), then *calc*, then *push*, then *send* and so on. Step *comm* is specific: it immediately ends the execution of step *send* and executes the two *receive* steps. Then *push* ends, then *trigger* ends. We thus reproduce

the semantics of successive method calls that end up to sending data through a network.

Because component *control* sends thrust data only to component *motor*, the generator configures the *thrust* connector to produce a UDP network datagram from node *DC* to node *MS*, carrying the CBOR serialization of the data. The reception technical code for component *motor* that executes on *MS* deserializes the data from the CBOR buffer, then calls the algorithmic code of *motor*. This means that, for this specific deployment, group *thrust* represented in fig. 7 has only one step *comm* in group *thrust*. The computation time within component *motor* is 3 ms.

The same generator produces both the technical code and the execution steps. This ensures the execution semantics (i.e. the execution steps) is consistent with the actual technical code.

VI. Computation of data stream models

In this section, we explain how we generate the application-related data stream parameters of section IV-A, using the model of the software architecture modeling of section V. Stream parameters that are related to system design cannot be calculated from the software application architecture.

In order to illustrate the automatic computation of data stream parameters, we focus on data streams 1 and 2.

A. Payload Size, Traffic Class, Talker, Listeners

All the data types handled by UCM have a known size (e.g. 16-bit integer, string with a maximum length, etc). In our example, UCM connectors serialize data with CBOR and send them over UDP sockets, adding 2 bytes to identify the connections. Thus the maximum size of the data exchanged between components can be calculated and extracted from the declarations of the data exchanges in the software architecture model. Performing both the connector technical code generation and the data stream computations, as illustrated on Fig. 2, ensures consistency between the code and the stream calculations. Data stream 1 has a payload of 7 bytes and data stream 2 has a payload of 12 bytes[2].

The traffic class parameter is, in general practice, mapped to a specific priority value. As our example system uses a TSN network, the parameter is mapped to a set of 8 different values ranging from 0, the lowest priority, to 7, the highest priority. The notion of priority is included in the design of our UCM connectors, from which we can extract this parameter value of the data streams. Data streams 1 and 2 both belong to the control traffic class which is mapped to the highest available priority. This parameter is called *pcp* (Priority Code Point) in listing 1 because of the TSN terminology.

UCM models contain deployment plans to specify which components are deployed on which network end-point. Components are connected to each other by connectors; the exact network topology is not known at the level of the software architecture. Yet, as connectors model end-to-end communications between components that are deployed on end-points, the talker and the listeners are retrieved from the network interfaces of the end-points on which the components are deployed. The data streams are exchanged back and forth between *DC* and *MS* on their *eth0* port.

[2]With TSN, both are rounded up to reach the minimum Ethernet frame size of 64 bytes. For the sake of genericity and clarity we kept these values.

B. Stream Period

The nature of the streams (periodic, sporadic) depends on the entity that initially induces them.

Fig. 7 represents the sequence of execution steps generated from the specification of the behavior of the software components deployed on end-points *DC* and *MS*. Some of the method calls these execution steps represent are the source of network communications. Models of data streams are generated when such execution steps are encountered in the execution flow.

Periodic data streams models are generated when method calls are controlled by a periodic execution policy. The period of the data stream is the same as the execution policy.

The same logic is applied to sporadic streams by treating the minimum interval between two consecutive executions as the period. We can extract the transmission period of the data streams from the execution flows generated from the behavior specification of the application. Our example data streams are both periodic data streams with a period of 30 ms. According to Fig. 7, stream 1 is produced by the *trigger* technical policy of component *control*. Stream 2 is produced by component *motor* upon the reception of a message produced by component *control*.

C. Stream Offset

The last parameter that can be automatically generated is the offset of each data stream. This parameter has a strong impact on the configuration of the network. When using scheduling mechanisms based on time division, precisely knowing the time of transmission of data streams is critical to configuration synthesis. Using wrong offsets would lead to missed transmission windows and thus increased end-to-end latency, possibly over the deadline. For this reason, we retain the worst case of transmission time, so that the data is always available when the network is expecting it.

In our example, the offset of the second transmission (stream 2) has to take into account the parameters of the first transmission (stream 1) because, according to the software model (section V-B), the emission of stream 2 depends on the reception of stream 1. Since we have to consider the worst case, we have to use the deadline of the first transmission as the base value of the offset because it is the worst possible time at which the second transmission can be triggered. We also have to consider the offset of the first transmission to this value to adjust for the delay between the launch of the system and the time at which the first transmission starts. The sum of these two values gives us the offset associated with the data stream whose transmission is triggered by the reception of another data stream.

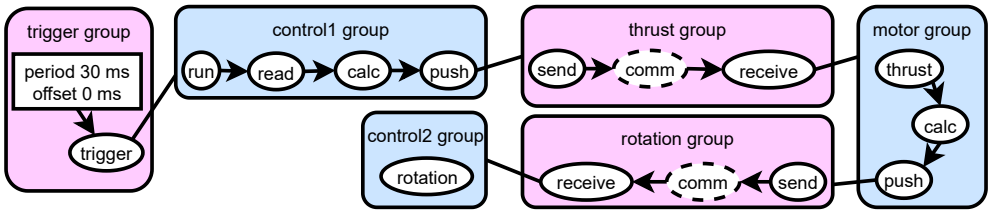

Fig. 7. Generated execution steps corresponding to the behavior of end-points *DC* and *MS* illustrated on fig. 6

Because of the 5 ms execution time of a step in the behavior specification of the source of data stream 1, its offset is 5 ms.

The offset of stream 2 is the addition of offset of stream 1 and the computation time in component *motor*. The computation time of *motor* is 3 ms (as mentioned in section V-C). The sum of the offset of data stream 1 and the computation time of *motor* produces the offset of data stream 2: 8 ms.

Listing 1 presents the different parameters that are automatically computed and their values.

```
[stream1]
  name = "control_thrust_to_motor_thrust"
  type = "Periodic_Stream"
  period.value = 30000
  period.unit = "us"
  offset.value = 5000
  offset.unit = "us"
  payload.value = 7
  payload.unit = "B"
  talker = ["::topology::DC.ep_eth0"]
  listener = ["::topology::MS.ep_eth0"]
  pcp = ["7"]
[stream2]
  name = "motor_rotation_to_control_rotation"
  type = "Periodic_Stream"
  period.value = 30000
  period.unit = "us"
  offset.value = 8000
  offset.unit = "us"
  payload.value = 12
  payload.unit = "B"
  talker = ["::topology::MS.ep_eth0"]
  listener = ["::topology::DC.ep_eth0"]
  pcp = ["7"]
```

Listing 1. Generated parameters for stream 1

Listing 2 illustrates the configuration of the Time Aware Shaper of switch SW1 for the Ethernet port that is connected to GW[3]. The first element defines the index of the time slot, then the bit vector indicates which traffic classes are allowed to transmit during the time slot and finally its duration is given in nanoseconds. Stream 1 and stream 2 are the only time-critical streams in the architecture and they do not reach GW. Therefore the configuration of the Time Aware Shaper for this Ethernet port is straightforward. It corresponds to allowing all traffic classes to always go through.

```
t0  11111111b 1000000000
```

Listing 2. Scheduling configuration on the port going from *SW1* to *GW*

[3]This configuration targets the `tsntool` configuration program from NXP.

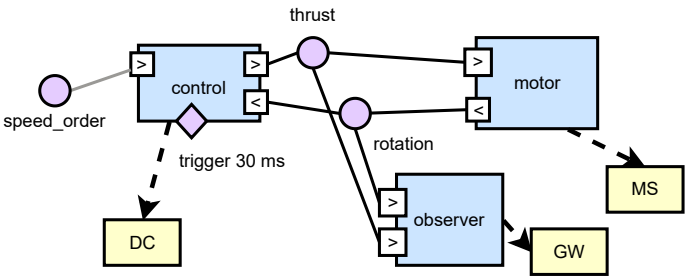

Fig. 8. Components deployment of end-to-end communication and observation

VII. Managing System Evolutions

Designing the whole system represents an important effort that could be made easier by breaking it into multiple iterations. This type of design process pairs well with a model-based approach such as the one proposed in this paper because using models and generation help maintaining consistency when updating the system. To demonstrate this, we will add a new software component to the system for monitoring purpose and show the impact on the stream calculations. The evolution of the architecture is illustrated on Fig. 8.

The role of the observation component is to ensure that the orders received by the vehicle are correctly followed. The presence of the new component may lead to additional network streams.

We choose to deploy component *observer* on end-point *GW*. Because the connectors only handle unicast messages, this creates two additional communications, as illustrated on Fig. 9.

```
[stream1b]
  #[...]
  listener = ["::topology::GW.ep_eth0"]
[stream2b]
  #[...]
  listener = ["::topology::GW.ep_eth0"]
```

Listing 3. Parameters of the newly created stream sent from *DC* to *GW*

The generator therefore produces two new data streams which have the same characteristics as stream 1 and 2 except for the listener, as seen in listing 3.

Instead of having to manually change the code of the applications, calculate the new data stream parameters, create a new network configuration and change simulation models and configuration files, our approach only requires the new component to be added to the application architecture (Fig. 8)

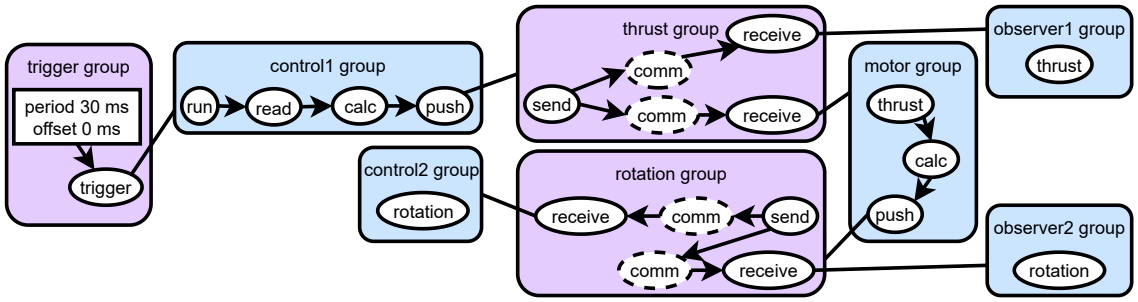

Fig. 9. Execution steps of end-to-end communication and observation that corresponds to Fig. 8

and guarantees the consistency of every generated piece of the system by design.

The new data stream path goes from *DC* to *GW*. No other data stream used this path before so there was no particular scheduling configuration going from *SW1* to *GW*, as seen in listing 2. This updated architecture of the system now needs this configuration, which is generated by MoBACT, the configuration generation tool we presented in [2], using the z3 SMT solver. Listing 4 illustrates the new switch port configuration, which replaces listing 2.

```
t0  01111111b 510000
t1  00000000b 7000
t2  10000000b 500000
t3  01111111b 28983000
```

Listing 4. Scheduling configuration on the port going from *SW1* to *GW* when the new component is deployed

VIII. Conclusion

In this paper, we presented an engineering process to assist the design of real-time networks. Our approach enables the automatic calculation of network data streams for control applications. Since network configuration is based on the characterization of data streams, it is essential to ensure that the model accurately matches reality. To reach this goal, our approach uses a single model of the application from which both the application technical code and the model of the data streams are generated.

Our process relies on a component model, namely UCM, to specify the application structure. We complement UCM with the modeling of the business code behavior, which is nested within software components, using a subset of UML sequence diagrams. The engineering approach assembles components, combined with technical patterns to create a complete software architecture. A generator produces both the technical code and a model of the execution semantics of the application. The model of the data streams can then be extracted from the semantic model and passed, along with a model of the network topology and requirements, to MoBACT [2].

This model-based approach, on top of guaranteeing consistency between software and model, enables the use of an iterative design process in an efficient way. As illustrated by the system we used as an example, adding new components to the system has to be done in a single place; the generators produce the entirety of what is needed to evaluate and deploy the system, i.e. application code, simulation models and network equipment configuration files.

This approach assumes the perfect scheduling of tasks, i.e. applications always have access to computation resources. Future works could be aimed at lifting this assumption by incorporating task scheduling into the approach. Future works will also focus on reducing the pessimism of offset computation. Instead of only considering the WCET of execution steps, the use of a probabilistic approach could be studied.

Acknowledgment

This work was partially supported as part of the CPS4EU project [5] under grant number N°826276.

References

[1] *Unified Component Model for Distributed, Real-Time and Embedded Systems (UCM)*, http://www.omg.org/spec/UCM, OMG Std., 2021.
[2] M. Samson, T. Vergnaud, E. Dujardin, L. Ciarletta, and Y.-Q. Song, "A model-based approach to automatic generation of tsn network simulations," in *WFCS*, 2022.
[3] *IEEE 802.1Q 2018 – Bridges and Bridged Networks*, IEEE Std., 2018.
[4] J. Falk, D. Hellmanns, B. Carabelli, N. Nayak, F. Durr, S. Kehrer, and K. Rothermel, "NeSTiNg: Simulating IEEE time-sensitive networking (TSN) in OMNeT++," in *Int. Conf. on Networked Systems*, 2019.
[5] About CPS4EU. [Online]. Available: https://cps4eu.eu/about/
[6] S. S. Craciunas, R. S. Oliver, M. Chmelík, and W. Steiner, "Scheduling Real-Time Communication in IEEE 802.1Qbv Time Sensitive Networks," in *RTNS*, 2016.
[7] *CBOR*, https://www.rfc-editor.org/rfc/rfc8949.html, IETF Std., 2013.
[8] B. Barritt, W. Eddy, S. Matthews, and K. Bhasin, *Integrated Approach to Architecting, Modeling, and Simulation of Complex Space Communication Networks*. SpaceOps Conference, 2010.
[9] B. Annighoefer, M. Halle, A. Schweiger, M. Reich, C. Watkins, S. H. VanderLeest, S. Harwarth, and P. Deiber, "Challenges and ways forward for avionics platforms and their development in 2019," in *DASC*, 2019.
[10] *UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems*, http://www.omg.org/spec/MARTE, OMG Std., 2019.
[11] *AADL*, SAE International Std., 2022. [Online]. Available: https://www.sae.org/standards/content/as5506d/
[12] *AUTOSAR*, Std. [Online]. Available: https://www.autosar.org/
[13] K. Triantafyllidis, E. Bondarev, and P. With, de, "Performance analysis method for rt systems : promartes for autonomous robot," in *2013 Forum on Specification & Design Languages (FDL)*, 2013, pp. 1–8.
[14] G. Brau, J. Hugues, and N. Navet, "Refinement of AADL models using early-stage analysis methods : an avionics example," in *Proc. of the 4th Analytic Virtual Integration of Cyber-Physical Systems Workshop*, 2013.
[15] K. Chaaban, N. Rizoug, B. Barbedette, and S. Saudrais, "Model-based development of an embedded steering-by-wire system," in *8th Int. Symp. on Mechatronics and its Applications*, 2012, pp. 1–6.
[16] M. Samson and T. Vergnaud, "Automatic Generation of Test Oracles from Component Based Software Architectures," in *IFIP ICTSS*, 2019.