Verifying Resource Adequacy of Networked IMA Systems at Concept Level

Rodrigo Saar de Moraes and Simin Nadjm-Tehrani

Saar de Moraes, R., Nadjm-Tehrani, S., (2020), Verifying Resource Adequacy of Networked IMA Systems at Concept Level, *Formal Techniques for Safety-Critical Systems*, (In: Communications in Computer and Information Science vol 1165) , 40-56. https://doi.org/10.1007/978-3-030-46902-3_3

Original publication available at:
https://doi.org/10.1007/978-3-030-46902-3_3

Verifying Resource Adequacy of Networked IMA Systems at Concept Level

Rodrigo Saar de Moraes * and Simin Nadjm-Tehrani

Dept. of Computer and Information Science
Linköping University - Sweden
{rodrigo.moraes,simin.nadjm-tehrani}@liu.se

Abstract. Complex cyber-physical systems can be difficult to analyze for resource adequacy at the concept development stage since relevant models are hard to create. During this period, details about the functions to be executed or the platforms in the architecture are partially unknown. This is especially true for Integrated Modular Avionics (IMA) Systems, for which life-cycles span over several decades, with potential changes to functionality in the future. To support the engineers evaluating conceptual designs there is a need for tools that model resources of interest in an abstract manner and allow analyses of changing architectures in a modular and scalable way. This work presents a generic timed automata-based model of a networked IMA system abstracting complex networking and computational elements of an architecture, but representing the communication needs of each application function using UPPAAL templates. The proposed model is flexible and can be modified/extended to represent different types of network topologies and communication patterns. More specifically, the different components of the IMA network, Core Processing Modules, Network End-Systems, and Switches, are represented by different templates. The templates are then instantiated to represent a conceptual design, and fed into a model checker to verify that a given platform instance supports the desired system functions in terms of network bandwidth and buffer size adequacy – in particular, whether messages can reach their final destination on time. The work identifies the limits of the tool used for this evaluation, but the conceptual model can be carried over to other tools for further studies.

Keywords: Timed Automata, UPPAAL, IMA System, Conceptual Analysis, Network Resource Adequacy

1 Introduction

Modeling complex cyber-physical system (CPSs) [6] can be a challenging task, particularly since, during the initial concept phase, architectures have to be defined or reflected upon without specific knowledge or fine-grained models of the functions to be executed or the software to be run on these platforms. Usually,

* corresponding author

details of the software, algorithms, and functions that are relevant to the development of conceptual platforms are not known beforehand. These elements, however, still have to be considered during the conceptualization of platform models so that enough processing and network resources are allocated to the system from the start. The challenges of modeling CPSs are even more pronounced when those are Integrated Modular Avionics (IMA) Systems [11]. Typically, aircraft implementing IMA-based systems have life-cycles that span across several decades, making it very difficult to consider or plan for future functionality extensions, making it imperative to consider for such phenomena in the initial concept of these architectures.

Given this motivation, the work described here presents a generic IMA-based network model to be used during the conceptual definition of candidate IMA platforms. The goal is to evaluate a candidate IMA architecture in terms of the applications and functions that it must support, abstracting complex network and computational system models. More specifically, the wish is to verify whether the resources of a candidate platform are sufficient to support an Avionics Application Model (AAM) that defines the resource requirements of the aircraft's platform. Also, the model permits the evaluation of alternative platform architectures, helping with the assessment of different candidate platform architectures that could potentially implement the AAM.

This work presents a model to evaluate the performance of IMA-oriented computer networks, focusing on a flexible model that can be later extended to represent different types of network architectures with different topologies and characteristics. The initial model focuses only on the network part of the resource adequacy problem. Other aspects such as processing capacity and schedulability are also important for the problem, but are not considered here.

The paper is structured as follows. Section 2 provides a theoretical background to the problem. Section 3 describes the methodology and the reasoning behind the development of the model, including the process to instantiate particular architectures. Section 4 describes the specification of high-level requirements for the system, as well as how to query the model to obtain relevant results. The results obtained by querying an experimental instance of an IMA architecture are presented in Section 5. Finally, the conclusion is drawn in Section 6.

2 Background

A recent survey performed by Wang and Niu [10] studies and discusses the characteristics of Distributed Integrated Modular Avionics Systems (DIMA) as well as the main technologies, scheduling algorithms, and methods used in the concept and design of contemporary DIMA system. In their discussion, they address the common problems and challenges encountered by engineers and designers during the development of these systems and highlight three key technologies that can help in the process: mixed critical task scheduling; real-time fault-tolerant scheduling; and real-time communication network delay analysis. The first two are concerned with how to schedule tasks to meet timeliness and dependability.

The delay analysis of the real-time communication network, on the other hand, is presented as a way to ensure the real-time performance of the distributed system.

In order to ensure that all tasks, which run on different processors, can meet the time constraints imposed by the application, the communication delay between two processing nodes must be strictly bounded. The problem, however, is that computing the exact worst-case delay for such networks is most of the time impossible since realistic IMA platforms are composed of dozens of communication models and hundreds of message flows. Therefore, approaches such as network calculus (NC) [3, 4] have been proposed. These approaches compute an exact, but often pessimistic upper bound for the delay of each message flow on the network. This pessimistic behavior usually leads to an over-dimensioning of the network architecture, which can quickly become expensive.

The NC technique is based on the idea of over-approximating message flows by arrival curves and under-approximating network elements by service curves. The worst-case delays are obtained by applying convolution and deconvolution operators on these curves. A recent work by Li et al. [8] uses NC to try to provide timing performance guarantees for heterogeneous multicore systems. Their work adds a virtual channel concept to each CPU core and provides a delay analysis for a typical switched network structure. The same NC approach is used by Soni et al. [9] who try to quantify the pessimism of the computed upper bounds of the NC technique when applied to an Avionics Full-Duplex Switched Ethernet (AFDX) network. In their report, the authors compare the delays calculated using network calculus with exact worst-case delays calculated using model checking. Their results show that the NC approach can introduce up to 12% percent overhead on the delay estimation due to its pessimistic tendencies.

Recent work by Xu and Yang [12] couples the concepts of Grouping Strategy and network calculus to take into account the serialization of the messages being transmitted through the same physical link in AFDX networks. They analyze the existing pessimism in network calculus and then propose a rate-constrained grouping strategy to improve the analysis of system performance. Addressing the phenomena of burst enlargement, they present a new strategy to cope with the pessimistic behavior of network calculus. Their approach, however, tends to obtain optimistic estimates for the end-to-end delay that can induce some risks to the utilization of this method in some corner cases.

Robati et al. [1], on the other hand, move away from NC and extend the Architecture Analysis and Design Language (AADL) modeling language to model Time-Triggered Ethernet (TTEthernet) based distributed systems. Their approach proceeds to define model transformations to enable the verification of the AADL models using Discrete Event System Specification (DEVS) based simulations. They present successful results for the verification of small IMA systems, but highlight that the automation of the refinement step of the model transformation is challenging and still requires some significant manual input from the user.

Finally, Zhang et al. [13] present a model for verification of the real-time constraints of IMA systems. They propose a finite-state machine mechanism to represent the behavior model of the application and the platform. The proposed model is based on specific requirements from the ARINC653 and ARINC664 (AFDX) standards. Their approach aims to address the claim that, while significant work has been made in terms of communication delay, RTOS service performance, and scheduling algorithms, these factors do not affect the system independently and the sum of their effects need to be taken into consideration in early development phases. Their approach, however, is tested with a small autopilot use case and is very likely to have scalability problems as the system grows to represent the whole aircraft.

In this work we explore the conceptual modelling of communication requirements and their verification using model checking with timed automata.

3 Methodology

The current model is structured in the form of a Network of Timed Automata (NTA) which can be instantiated according to the characteristics of the architecture and applications the user wants to investigate. This approach lets the behavior of each different component of the network model to be represented as a Timed Automaton (TA)[2] which communicates with other TAs via broadcast channels and shared variables to generate Networks of Timed Automata that can be fed into a Model Checker (MC) for simulation and analysis.

The usage of NTAs allows for a flexible and modular system that can be easily modified to accommodate new components and behaviors or be extended through the modification of the existing TAs or the addition of some new ones. This approach limits the modifications to the TA that implements the component to be changed or extended, not requiring the whole system or the interactions between the other components to be modified. NTAs also allow for flexibility in terms of the instantiation of different candidate architectures, since the TAs behaviors are independent of each other, only exchanging information through the communication channels or shared variables, different architectures can be easily implemented by instantiating different TAs, with different behaviors, for the different components of the system as long as the interface between the components is maintained. One can, for example, instantiate an TA representing a given network scheduling algorithm, i.e. round-robin, to analyse a candidate architecture and, when desired, de-instantiate this TA and switch it for another TA representing another, i.e. priority-based, scheduling algorithm, without having to re-model the whole system and the interaction between the components.

The current work uses the UPPAAL toolbox [7] as a resource for the design, simulation, and verification of the NTA model. The tool provides support for the representation of real-time systems as networks of timed automata, extending the automata representation with integer variables and structured data types, and providing channel synchronization mechanisms to support the communication between the automata.

The instantiation of an NTA model requires two different types of descriptive documents: a Global Declaration File, in which the specifics of the system, in this case of the avionic applications and of the IMA architecture, are described and declared; and a Component Instantiation File that lists which components of a library or set of TA templates will be instantiated and how these templates relate to the information provided on the System Declaration Document. With the information provided by these two documents, the toolbox is able to compile an NTA instance of the IMA architecture that was described. This model is then fed into a Model Checker which will verify if the model satisfies certain desired properties, or, in this case, whether the instantiated architecture meets the resource adequacy and timeliness requirements defined for the IMA system. Finally, the SMC provides the user with results of the verification, providing both the final status of the verification for each of the requested requirements, as well as a trace that represents the state of the system upon non-compliance. More details on each of these documents and the TA templates will be given in the subsequent subsections of the document.

3.1 Overall Network Architecture

Figure 1 illustrates an IMA network system as modeled in this work. The diagram represents a system composed of m processes, labeled T_1 to T_m, allocated to n Core Processing Modules (CPM), labeled CPM_1 to CPM_n. The CPMs, in turn, are associated to n Network End-Systems (ES), labeled ES_1 to ES_n, that are connected to each other through a network, represented by the dotted box on the lower part of the diagram. The arrows in Figure 1 represent the flow of information, or in this case exchange of messages, between the components.

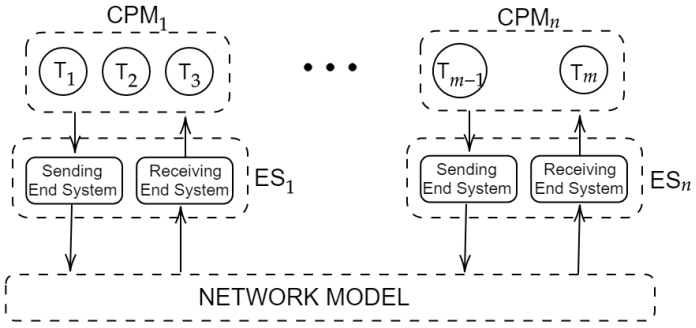

Fig. 1. Diagram of a Generic network

Each Network End-System is composed of two different components, a Sending End-System, responsible for forwarding the messages it receives from processes onwards into the network, and a Receiving End-System, responsible for

delivering the messages it receives from the network to the processes. It is important to highlight that each of these two components is associated, in the NTA model, to a different TA template. On the other hand, the ES itself, which encompassess both components, is not mapped to a TA, being merely a conceptual entity in our model.

Similarly, each process is mapped to a TA model that represents its behavior. CPMs are also just conceptual entities within the model and are not mapped to TAs. This representation choice is due to the fact that modeling the behavior of the CPMs themselves is not really relevant to the analysis of the network adequacy in this work since for the current analysis only the rate in which processes generate messages matter.

Finally, the Network Model represents the network architecture used to connect different CPMs. This component is, again, merely a conceptual entity composed of multiple and different TA instances depending on the type of network or architecture being analyzed.

Figure 2 illustrates how a switched network, where n CPMs are connected through an n-port-switch, can be instantiated. In this example, the switch is represented by two types of TA templates: Sending Interface TAs, which are responsible for forwarding messages to the receiving end-systems; and Switch Core TAs, responsible for the routing and switching of the messages received from the sending end-systems, assigning each message to the corresponding Sending Interface.

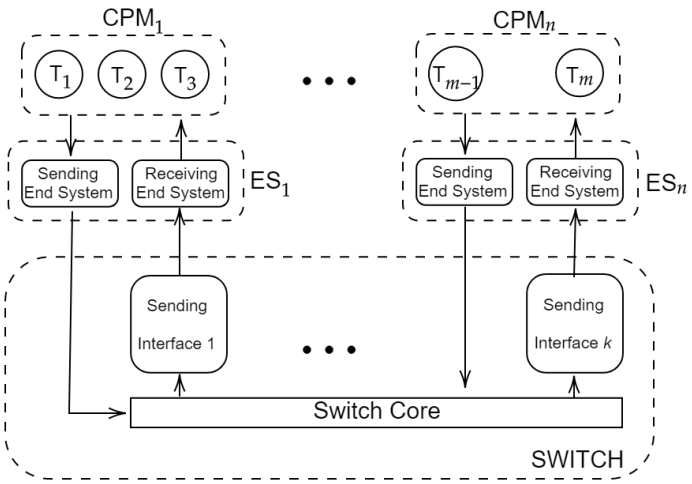

Fig. 2. General Diagram of a Switched Network

In this model, following the interfaces provided by UPPAAL, the communication between the different Timed Automata representing the components of the network is made using shared variables. These shared variables model

buffers and represent the internal storage structures that exist in most of the real physical components. This approach allows each automaton that represents a network model to forward messages to the next node in the network by writing the message directly on the other node's input buffer, modeling the delivery of a message in the receiving node. More on this behavior is discussed when the automata for the components of the system are presented in section 3.3.

3.2 System Global Declarations

The System Global Declaration serves the purpose of describing the resource-related part of the IMA platform and AAM being analyzed. Here, the specific aspects of the system, such as the characterization of the end-to-end communication, the number, and the timing characteristics of the processes and of the underlying network are set. Moreover, it is also where the declaration and initialization of the communication channels, shared variables, system constants, and common functions take place.

Listing 1.1 shows an excerpt of our configuration file, showing the specific part of the file where the general variables used to describe a specific IMA architecture are located, as well as a description of their meaning.The characteristics described by these variable are specific to each architecture, detailing specific aspects of said architecture such as the number of processes, end-systems, and messages, as well as platform aspects such as the size of the network buffers and the bandwidth of the network.

```
const int N_ES =2;              // The number of end systems.
const int N_PROC = 6;           // The number of processes.
const int N_MESS = 11;          // The number of different
types of messages in the system.
const int SIZE_M = 16000;       // The maximum size of the
messages in bytes.
const int BUFFER_SIZE = 16;     // The maximum size of
network buffers in kbytes.
const int NETWORK_BD = 100;     // network bandwidth in mbps/s
```

Listing 1.1. General System Description Variables

Listing 1.2 exemplifies the declaration of a simple process. A process is described by a *Process* data structure that carries information about the worst case execution time of the process, the period in which it should be run, the end-system it is associated with, and the number and list of messages the process is supposed to read and write from the network. Each *Process* structure also caries a specific process ID, which will be fed to a generic process TA template during instantiation and allows the template instantiated for each process to access the shared data about the process they relate to.

In this case, we can see the instantiation of a process $P1$, characterized by id number $TID\_t = 1$, associated with end-system $ESID\_t = 0$, that takes maximum $7ms$ to run and runs each $16ms$. We also see that process P1 makes

3 writes to network, being writes of message types 1, 2 and 3, and performs the reads of two message types, 4 and 5, from the network.

```
 //A data structure representing a process and its
    charachteristics
typedef struct
{   TID_t id;            //process id
    time_t wcet;      //process WCET
    time_t period;       //period of the process
    ESID_t associatedES;        //an identifier of the
    End-System the process is associated with
    NetworkWrites netWrites;    //a NetworkWrites object that
    lists the messages this process sends
    NetworkReads netReads;      //a NetworkReads object that
    lists the messages this process receives
}Process;

// Definition of a Process P1
const Process P1 =  {1,7000,16000,0,
{3,{1,2,3}}, {2,{4,5,NO_MESSAGE,NO_MESSAGE}}};
```

Listing 1.2. Process Data structure and Definition of a Process

We now go on to exemplify how the messages exchanged between processes are defined in the context of the model. Listing 1.3 demonstrates how messages are defined in terms of a message type id, information about the sender and receiver processes, and the size of the message. Towards the end of the listing, there is an example of how a 3608 bytes long message with type id $MID\_t = 1$, that goes from process 3 to process 2, can be instantiated.

```
//A message element structure
typedef struct {
    TID_t sender;        //the id of the sender process
    TID_t  receiver;    //the id of the receiver process
    MID_t id;            //the id of the message type
    int [0,SIZE_M] size; //the size of the message in bytes
}Message;

//Definition of a message M1
const Message M1 = {3,2,1,3608};
```

Listing 1.3. Message Data structure and Definition of a Message

3.3 Timed Automata Templates

Timed Automata Templates in UPPAAL Each automaton template that composes the final system is instantiated from a parameterized template. The parameters for each template are replaced by arguments at the moment template instantiations are declared. After instantiating the components, these have to be composed into a system, which is made through a system definition.

Conceptual Components as Timed Automata Templates In order to instantiate and define a system similar to the ones depicted in Figures 1 and 2 a series of templates modeling the behavior of the components of the systems have been created. The remainder of this subsection is devoted to the presentation of these templates. The syntax of the diagrams used on the representation of the templates follows that of UPPAAL.

– **Process Model**: The Process Model is an abstraction of the application processes' communication needs in this work, acting as both a sink and a source of messages depending on the location [1] the automaton finds itself in. It has 3 different locations: the *Idle* location, representing the situation in which the process is not realizing network-related activities, neither receiving nor sending messages, being idle from the perspective of the network interface; the *RetrievingMessages* location, that is reached immediately after the process leaves *Idle*, is where the automaton verifies which messages were delivered to that process since the last time it ran; and the *Sending* location, which models the state where the process has received all the messages it needed to run and done its computations, after which it creates and sends its own messages to the network before going back to *Idle*. In case a process verifies it did not receive the messages it was expecting in the *ValidatingInput* location, the process automaton communicates this error to the rest of the system through a special error communication channel and goes back to *Idle*, not going forward into the *Sending* location. Figure 3 depicts what this template looks like.

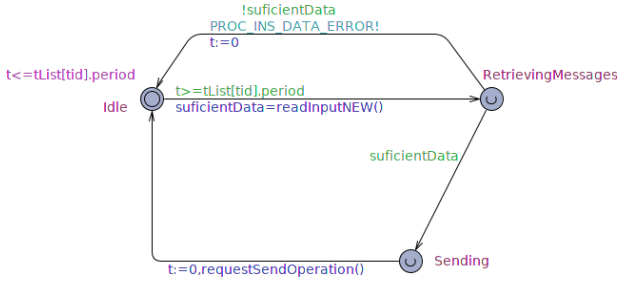

Fig. 3. The Process Model Template

– **Sending End-System Model**: The Sending End-System Automaton is responsible for forwarding the messages generated by one or more processes into the network part of the system. The automaton is composed by an *Idle* location, in which it waits until a request is received from a process; a *Buffering* location, in which the end-system fetches and buffers the messages from the processes upon a request being received; and a *Sending* location, in

[1] UPPAAL term for the state in Automata

which the automaton stays while it is sending messages to other nodes in the network. In case the end system has several messages waiting to be sent, it will bundle the messages together as to use the whole bandwidth available on the network by looping through the *Buffering* and *Sending* locations while is has messages to send. The current implementation of the Sending End-System models a FIFO message scheduling algorithm to arbitrate between the messages of several processes. Given the structure of this template, other scheduling approaches can be implemented if needed by changing the way messages are buffered and sent inside the states of the automaton, which are code that runs on the background and are not reflected on the structure of the model. This approach allows for the extension of the template to support multiple scheduling policies without significant modifications to the structure of the automaton. Figure 4 depicts the Sending End System Automaton.

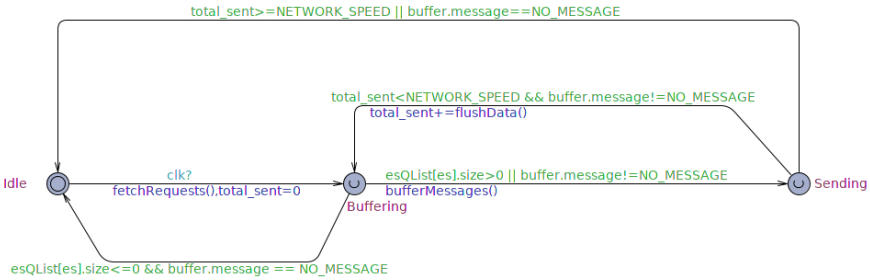

Fig. 4. The Sending End System Template

– **Receiving End-System Model**: The Receiving End-System is perhaps the simplest automaton in the model. Its main role is to deliver the messages that have been written to its internal buffer to the processes. This part is performed by periodically looping through the *Idle* and *Delivering* locations that compose this process. A graphical representation of the Receiving End-System is shown in Figure 5.

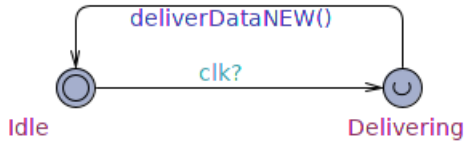

Fig. 5. The Receiving End System Template

- **Switch Sending Interfaces**: The Switch Sending Interfaces model is very similar in behavior to the Sending End-System model, the difference being that the first interface fetches messages from its internal buffer, which is fed by the Router Core, whereas the latter fetches its messages from the processes. Due to the similarity of this automaton with the Sending End-System automaton, a graphical representation of this automaton will be omitted.
- **Switch Core**: This automaton models the behavior of a network switch forwarding engine, forwarding the messages received in its Input Buffer from the Sending End-Systems to the correct Sending Interface associated with the Receiving End-System each message is destined to. This automaton works by periodically leaving the *Idle* location to the *Fetching* location, where it fetches the next message in its input buffer. Having fetched the message the automaton proceeds to the *Routing* location, in which it finds out which Sending Interface to deliver the message to. A cycle of the automaton execution ends on the *Delivering* location, delivering the message to the correct Sending Interface, and returning to the *Idle* location by one of two edges, depending on whether the Sending Interface buffer is full and the MESS_DROP_ERROR error message has to be signaled or not. This behavior can be seen in the automaton representation of Figure 6.

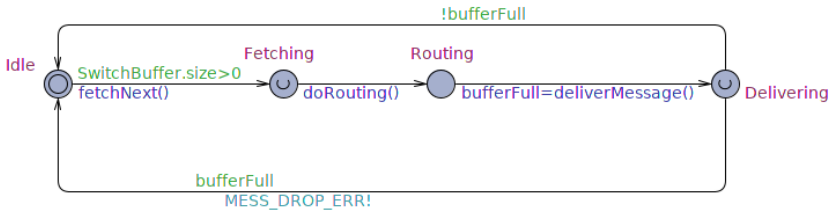

Fig. 6. The Switch Core Template

4 Requirement Specification

We begin by describing the requirements of interest in our case study.

4.1 Requirement Definition

To evaluate a candidate platform within a conceptual architecture, we need to ascertain whether any avionics-related application(process) can ever be starved by the network, meaning that it will not receive the data it needs to run, and also whether any message will be lost due to lack of resources or inadequate sizing of the network. That leads to the specification of two main high-level requirements for the system in terms of resource adequacy and network performance:

1. **No process should ever reach a state in which it needs a data and has not yet received the data it needs** - meaning that whenever a given process needs data from a message this data should be available. The failure to meet this requirement means that, for some reason, that specific IMA platform configuration is not able to respect the communication deadlines imposed by the AAM.
2. **No network node should ever reach a state in which messages are dropped** - this requirement means, in other words, that there should not exist a network node, be it a switch, or an end-system, that continuously receives more data than it can forward or deliver where upon it completely fills its internal buffer. A node for which the buffer is full is very likely to get overloaded in an operational mode.

It is important to note that, while a failure to meet requirement 2 will probably lead to a failure of requirement 1 as well, the opposite is not true. If a given message is dropped somewhere on the network, failing to meet requirement 2, it will never arrive at its final destination, causing a failure to meet requirement 1. This is, however, just a resource adequacy problem. A message not arriving in time at its final destination, on the other hand, can be caused for multiple factors, being a much broader problem related not only to resource adequacy but also to characteristics such as the number of messages being exchanged, the number of switches between two end-systems, and the topology of the network. These requirements are, thus, complementary in some sense, allowing whoever is using the model to get a better insight on where a problem with some platform/architecture might be coming from.

4.2 Verifying Requirements in UPPAAL

In UPPAAL, models can be verified by creating auxiliary observer templates that monitor whenever a requirement is violated (i.e a bad state is reached.) Hence, two observers were created to inspect the status of the platform model during the requirement verification process. Basically, these observers are simple timed automata that listen to the communication channels for error signals sent by processes or network nodes, and change their state, to an error state. Figure 7 shows what an observer listening for processes that signaled a non-compliance to the first requirement looks like. The second observer, which listens to the network nodes waiting for signals that indicate that a full-buffer-state has been reached, was omitted because it looks very similar to the first observer.

Fig. 7. Process Observer Automata

4.3 Expressing Requirements in UPPAAL

The UPPAAL model-checker tool [7] uses a simplified version of a Timed Computational Tree Logic (TCTL) [5] to express requirements over the timed automata models. Like in traditional TCTL, the UPPAAL requirements language supports both path formulae and state formulae. State formulae reason about individual states, whereas path formulae reason over paths or traces in the search space of the model. Since the goal of the IMA network model is to verify whether a given platform is able to serve as a basis for a given AAM with adequate resources, we have a special interest on expressing the requirements of the network model in terms of path formulae in terms of (non) reachability of undesired states (expressed in formal terms as the safety of the model). In short we aim to verify that no undesired or error state can ever be reached.

In the UPPAAL requirements language, given the TCTL logic and a formula φ, the path formula $A\Box\varphi$ express that φ should be true in all reachable states of the model. This type of requirement, usually expresses the so called *safety properties*, that in UPPAAL are formulated positively, e.g., something desirable is **invariantly true**. The two defined requirements are, then, written as:

A □ not ProcessObserver.INS_DATA_ERROR
A □ not NodeObserver.MESS_DROP_ERROR

5 Model Assessment

This section presents the analysis of an abstract networking platform architecture and an application characterised by a mapping to the platform. We then formally verify the the requirements mentioned in Section 4 and discuss the findings of the formal verification. We use an illustrative use case that consists of 6 processes, allocated to 3 different CPMs that communicate with each other by means of a switched network. These 6 processes exchange a total of 11 message types.

While a graphical representation of the architecture is depicted in Figure 8, the message graph of Figure 9 shows the direction of each of the messages exchanged by the processes, depicting the sender and receiver of each message. Figure 9 also outlines the message dependencies between processes, a fundamental piece of information for the verification of Requirement 1. Listing 1.4, in turn, details the declaration of each process and message, characterizing information such as the period of the processes, the end-system each process is associated with, as well as the size of each one of the 11 message types with each other.

The results of the verification of two different platform instantiations for requirement 1 are shown in Table 1. The first instance considers that the network links of the candidate platform have a bandwidth of 1 Gbps; the second, represents the case in which the network bandwidth is just 1 Mbps. Table 2, on the other hand, shows the results of a verification of requirement 2, presenting 4 different instances of the platform with different buffer sizes for the network models.

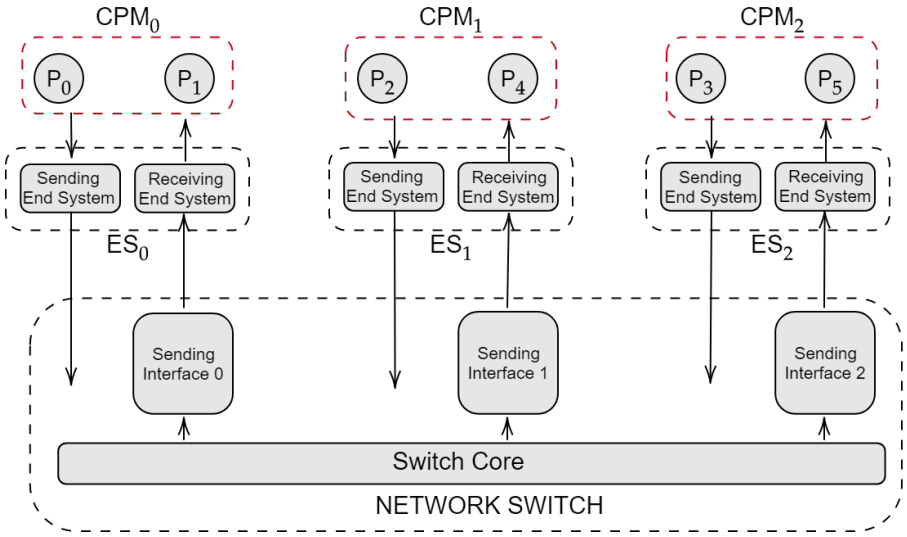

Fig. 8. Test Case Architecture

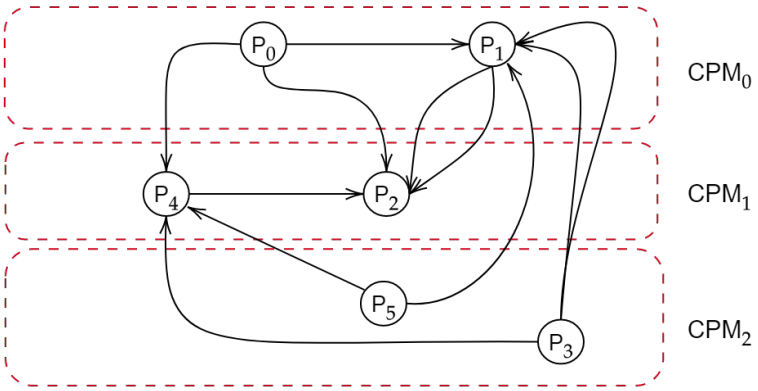

Fig. 9. Test Case Message Graph

Analyzing the results obtained from the verification of requirement 1, it is easy to see that, whereas the instance featuring a fast network (1Gbps bandwidth) was able to respect the communication deadlines imposed by the AAM, the instance featuring a slower network (1Mbps bandwidth) did not meet this requirement. The result of this verification was already expected since this instance was created to illustrate, given the size of the messages, the bandwidth of the network, and the periodicity of the processes, how a bad choice of network bandwidth could lead to a breach of requirement 1.

Query: Req 1: Correct timing for data delivery		
Instance	Verification Time (s)	Verification Result
1Gbps Network	1868.05	SUCCESS
1Mbps Network	34.62	FAILURE

Table 1. Requirement 1 Verification Results

Query: Req 2: No messages dropped		
Instance	Verification Time (s)	Verification Result
8Kb Buffer Size	3.29	FAILURE
16Kb Buffer Size	3.36	FAILURE
32Kb Buffer Size	16.04	FAILURE
64Kb Buffer Size	1857.55	SUCCESS

Table 2. Requirement 2 Verification Results

Turning to the results in Table 2, the verification of requirement 2 leads to the conclusion that the components of the network should have buffers that are somewhere between 32kb and 64kb in size. This behavior can be explained by the periodicity of the processes. When the buffers are smaller than 32kb the periodicity of the processes can lead to bursts of messages that small buffers cannot deal with.

The results also show that the verification approach performs quite well in cases in which the requirements are not met, being able to inform the user about resource inadequacy or network problems within seconds. When the system does not present any problem, however, the verification of the model takes considerably longer. This behaviour was already expected since proving that one of the requirements is not met is an easier task than proving that they are met. To prove that the requirements defined on section 4 are met, the model-checker has to verify the whole state-space of the system to guarantee that no error state is ever reached. On the other hand, proving that the requirements are not met is as simple as finding one branch of the state-space of the system in which one of the error states is reached.

More importantly, the results from this case study show that the proposed approach suffers from a severe scalability problem. Experiments made with more processes and messages, such as 9 nodes and 16 messages, have shown a tendency of the model to quickly get into a state-explosion problem, using up too many computational resources and eventually leading the model-checker to terminate the verification with inconclusive results. Since a common IMA system can be composed of hundreds of processes, tenths of CPMs and end systems, and thousands of message classes, such behavior raises some concerns about the suitability of the system to be used in such cases.

```
// ----- processes
const Process processList[N_PROC]:={
```

```
{tid[0],7000,16000,esid[0], {3,{1,2,3}},   {0,{0,0,0,0}}},
{tid[1],6000,32000,esid[0], {2,{4,5,0}},   {4,{1,11,6,7}}},
{tid[2],3000,64000,esid[1], {0,{0,0,0}},   {4,{3,4,5,9}}},
{tid[3],5000,16000,esid[2], {3,{6,7,8}},   {0,{0,0,0,0}}},
{tid[4],8000,32000,esid[1], {1,{9,0,0}},   {3,{8,10,2,0}}},
{tid[5],3000,16000,esid[2], {2,{10,11,0}}, {0,{0,0,0,0}}}};

// ----- messages
const Message mList[N_MESS]:={
{0,1,1,3608},   {0,4,2,1449},   {0,2,3,8519},   {1,2,4,1519},
{1,2,5,145},    {3,1,6,10585},  {3,1,7,550},    {3,4,8,4956},
{4,2,9,3257},   {5,4,10,5674},  {5,1,11,391}};
```

Listing 1.4. Processes and Messages Declaration

6 Conclusions

This work has detailed the process and methods applied to the development and test of an integrated modular avionics platform performance evaluation model. The developed model was supposed to be a tool to help the professionals involved in the early conceptual phases of IMA architecture definition to evaluate and assess different architectures or platforms for their IMA system.

Through the verification of a candidate architecture, the model is shown to be capable of analyzing and verifying the network requirements of candidate architecture platforms. Such functionality, however, comes with a great cost in computational power and time even for small systems, showing an accentuated scalability problem with the current version of the model, something that can severely influence the usability of the solution. This leads us to the conclusion that, while the conceptual modelling approach developed in this work seems promising, the UPPAAL encoding of it does not seem to scale.

In conclusion, further work is needed to analyse real-life-sized IMA architectures of this nature. Moreover, extensions such as the addition of new message scheduling algorithms, creation of templates for different switches or network modules, and the support for different network standards and topologies could help to enrich the model and improve the value of the developed solution.

Acknowledgements

This work was supported by the Sweden's Innovation Agency - Vinnova, as part of the national projects on aeronautics, NFFP7, project CLASSICS (NFFP7-04890).

References

1. A modeling and verification approach to the design of distributed ima architectures using ttethernet. Procedia Computer Science **83**, 229–236 (2016)

2. Alur, R., Dill, D.: Automata for modeling real-time systems. In: International Colloquium on Automata, Languages, and Programming. pp. 322–335. Springer (1990)
3. Cruz, R.L.: A calculus for network delay. i. network elements in isolation. IEEE Transactions on Information Theory **37**(1), 114–131 (1991)
4. Cruz, R.L.: A calculus for network delay. ii. network analysis. IEEE Transactions on Information Theory **37**(1), 132–141 (1991)
5. Goldblatt, R.: Logics of time and computation, vol. 7. Center for the Study of Language and Information Stanford (1992)
6. Khaitan, S.K., McCalley, J.D.: Design techniques and applications of cyberphysical systems: A survey. IEEE Systems Journal **9**(2), 350–365 (2014)
7. Larsen, K.G., Pettersson, P., Yi, W.: Uppaal in a nutshell. International Journal on Software Tools for Technology Transfer (STTT) pp. 134–152 (1997)
8. Li, M., Zhu, G., Savaria, Y.: Delay bound analysis for heterogeneous multicore systems using network calculus. In: IEEE Conference on Industrial Electronics and Applications (ICIEA). pp. 1825–1830 (2018)
9. Soni, A., Li, X., Scharbarg, J., Fraboul, C.: Work in progress paper: pessimism analysis of network calculus approach on AFDX networks. In: IEEE International Symposium on Industrial Embedded Systems (SIES). pp. 1–4 (2017)
10. Wang, H., Niu, W.: A review on key technologies of the distributed integrated modular avionics system. International Journal of Wireless Information Networks **25**(3), 358–369 (2018)
11. Watkins, C.B.: Integrated modular avionics: managing the allocation of shared intersystem resources. In: IEEE/AIAA Digital Avionics Systems Conference. pp. 1–12 (2006)
12. Xu, Q., Yang, X.: Performance analysis on transmission estimation for avionics real-time system using optimized network calculus. International Journal of Aeronautical and Space Sciences **20**(2), 506–517 (2019)
13. Zhang, K., Wu, J., Liu, C., Ali, S.S., Ren, J.: Behavior modeling on arinc653 to support the temporal verification of conformed application design. IEEE Access **7**, 23852–23863 (2019)