Model Checking Message Delivery Times in SpaceWire Networks

Andrii Kovalov
German Aerospace Center (DLR)
Institute for Software Technology
Braunschweig, Germany
Andrii.Kovalov@dlr.de

Girish Patil
German Aerospace Center (DLR)
Institute for Software Technology
Braunschweig, Germany

Vishav Bansal
German Aerospace Center (DLR)
Institute for Software Technology
Braunschweig, Germany

Andreas Gerndt
German Aerospace Center (DLR)
Institute for Software Technology
Braunschweig, Germany
University of Bremen
Bremen, Germany
Andreas.Gerndt@dlr.de

ABSTRACT

This paper presents a model checking framework in Uppaal for finding worst-case message delivery times for periodic and event-driven message flows in a SpaceWire network with wormhole switching. In particular, we focus on segmentation of large messages into smaller packets. We present a collection of timed automata for SpaceWire links and network messages, that capture message segmentation and wormhole blocking.

We evaluate our approach on a realistic example network with 4 routers and 16 message flows, two of which are large messages that need to be segmented. Our model can be used to determine the bounds on the possible segment size, and how this size affects the worst-case message delivery times. Model checking time for these experiments ranges from several minutes to several hours, and we further investigate how it depends on the number of flows, the segmentation size, and the message periods.

CCS CONCEPTS

• **Networks** → **Network performance modeling**; • **Software and its engineering** → **Model checking**; • **Computing methodologies** → **Model verification and validation**.

KEYWORDS

Model checking, Uppaal, SpaceWire

ACM Reference Format:
Andrii Kovalov, Girish Patil, Vishav Bansal, and Andreas Gerndt. 2022. Model Checking Message Delivery Times in SpaceWire Networks. In *ACM/IEEE 25th International Conference on Model Driven Engineering Languages and Systems (MODELS '22 Companion), October 23–28, 2022, Montreal, QC, Canada.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3550356.3561546

MODELS '22 Companion, October 23–28, 2022, Montreal, QC, Canada

ACM ISBN 978-1-4503-9467-3/22/10.
https://doi.org/10.1145/3550356.3561546

1 INTRODUCTION

SpaceWire is a serial communication network designed for space applications [6]. The project 'Scalable On-board Computing for Space Avionics' (ScOSA) [13] at the German Aerospace Center (DLR) aims to provide a scalable and reliable on-board processing platform, and uses SpaceWire as the primary network technology.

SpaceWire networks consist of nodes, point-to-point links, and routers. It uses the wormhole switching mechanism, which means every packet first blocks all the required ports in the routers along its path (creates a 'wormhole'), and then the packet content is transmitted along this path with a low latency. The approach does not require packet buffering, and supports arbitrary packet size. However, if a packet needs to be sent to an output port which is already in use, it is blocked until the port becomes available. This can lead to cascades of blocking, and potentially to deadlocks. SpaceWire protects from this by port time-outs: when a packet is stuck for too long, it is dropped.

Due to this blocking, transmission of large packets can cause other packets to be delayed or dropped. In ScOSA this problem is solved by splitting large messages into smaller segments, which are sent as separate SpaceWire packets. This functionality is provided by the SpaceWire-IPC protocol [15] at the transport layer. As a result, instead of a large message blocking the channel for a long time, the channel is blocked multiple times for a short duration. This enables other concurrent messages to pass through.

However, segmentation comes at a cost. First, there is some overhead in the overall message size, since every segment requires a header. Second, for each segment there is additional processing time in the sender, the receiver, and in the switches along the path. Therefore, determining the optimal size of a segment is an important practical question in the design of a ScOSA system.

In this paper we present a method to model the network and message flows as timed automata, and use the Uppaal model checker [1] to check properties such as deadlock freedom and the worst-case message delivery times considering segmentation. Such a model allows system engineers to explore different design options, such as different segment sizes or a different task-node mapping, and better understand their consequences.

The remainder of the paper is structured as follows. We review the related work in Section 2, introduce our modeling framework in Section 3, evaluate our approach on a realistic network example in Section 4, and conclude the paper in Section 5.

2 RELATED WORK

Model checking can be applied to different aspects of SpaceWire networks. There is some research on verifying low-level SpaceWire mechanisms such as link initialization and error detection [14] [4].

As for analyzing network traffic, [7] shows a SpaceWire model in Uppaal that can be used to check worst-case message transmission times. The model is evaluated on an example with 4 nodes and 9 message flows. The results are also compared with network calculus and recursive analysis, and model checking is shown to provide better bounds.

On the analytical methods side, [5] shows a method of delay estimation in networks with wormhole routing, where the traffic is described by statistical parameters.

For the SpaceWire protocol, earlier work on calculating worst-case delivery times has been performed using recursive computation [8] and network calculus [9], and a comparison of these methods has been shown for an industrial application [10].

One of the inspirations for our work is [3], which presents a framework for schedulability analysis using Uppaal. Although it is not directly related to networking, there are parallels with our domain. In SpaceWire, a message blocks links for exclusive use in a similar way to a task claiming resources in a schedulability problem.

3 SYSTEM MODEL

Our system model consists of timed automata for network links, processing tasks, and messages. The goal is to capture SpaceWire network message passing, especially wormhole switching and segmentation of large messages.

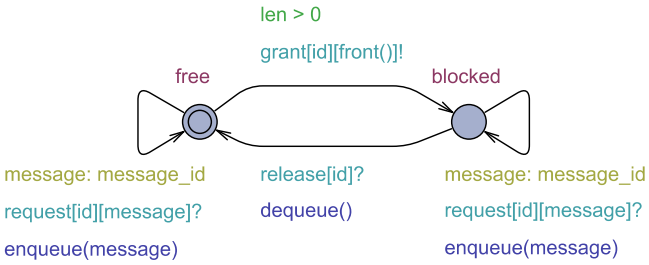
3.1 Link Model

SpaceWire links are full-duplex and bi-directional, so the traffic in one direction does not interfere with the traffic in the opposite direction. Therefore, every physical link can be modeled as two unidirectional links.

Our link model is shown in Fig. 1. A link can be used by one message at a time, and maintains a FIFO queue of messages that are waiting to use it. The queue is based on the train gate Uppaal example as described in [3].

The SpaceWire standard does not specify, in which order the blocked packets must be served. It just states that arbitration should be fair, e.g. each message should eventually be served. We decided on a FIFO queue for the simplicity of modeling. SpaceWire hardware can use other arbitration mechanisms, e.g. round-robin.

Every link has an `id` passed as a template parameter. A link starts in a `free` state with an empty queue and waits for a signal `request[id][message]?` from a message that needs to use it. On receiving this request, the message identifier is added to the queue. If the link is free, it sends a `grant[id][message]!` signal to the first message, which grants this message exclusive access to the link. After the




```
// Link template parameter
const link_id id

// Project declarations
const int L = 18;  // number of links
typedef int[0,L-1] link_id;

const int N = 25;  // number of messages
typedef int[0,N-1] message_id;

urgent chan request[L][N], grant[L][N];
chan  release[L];


// Link template declarations

message_id q[N];   // queue of waiting messages
int[0,N] len;      // current length of the queue

// Put a message at the end of the queue
void enqueue(message_id message) {
    q[len++] = message;
}

// Remove the first message in the queue
void dequeue() {
    int i = 0;
    len -= 1;
    while (i < len)
    {
        q[i] = q[i + 1];
        i++;
    }
    q[i] = 0;
}

// Get the first message in the queue
message_id front() {
    return q[0];
}
```

Figure 1: Uppaal template for a link (above) and the corresponding declarations (below).

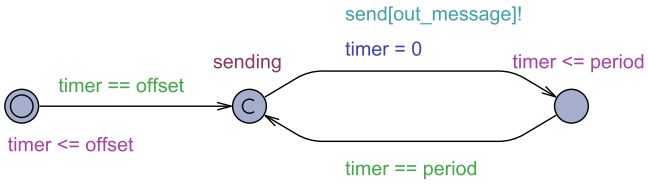


Figure 2: Uppaal template for a periodic task.

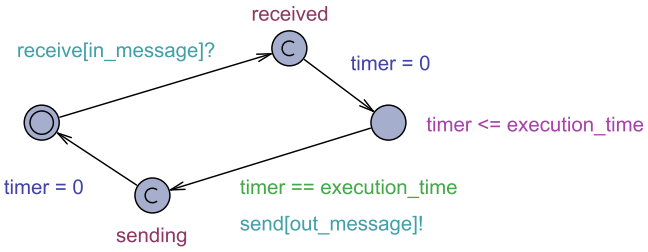


Figure 3: Uppaal template for a pipeline task.

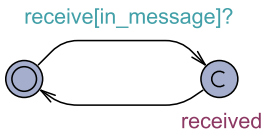


Figure 4: Uppaal template for a sink task.

message has finished transmitting, it sends a `release[id]!` signal, and the next message in the queue can be served.

3.2 Task Model

Tasks are sending and receiving messages in the system. Our model consists of three types of tasks - periodic tasks, pipeline tasks and sink tasks.

Periodic tasks (Fig. 2) send an `out_message` periodically with the given `period` and `offset`.

Pipeline tasks (Fig. 3) wait for an `in_message`, then wait for `execution_time`, and send an `out_message`. This template can be extended to handle multiple input and output messages.

Sink tasks (Fig. 4) just consume an `in_message` without any additional logic.

In reality, tasks run on processing nodes and compete for processing time. Scheduling of task execution on nodes depends on the tasking implementation and the operating system, and it is beyond the scope of this paper. Our model assumes that all tasks are always ready to execute.

3.3 Message Model

Messages in the network are modeled as timed automata as illustrated in Fig. 5. This model captures the segmentation of large messages, the blocking of network links along the message path, and the transmission delay. The template shown in Fig. 5 assumes a path of three hops, but it can be adjusted to any number of hops.

The same template is used for both large and small messages, and the segmentation is based on the payload size.

The timed automaton starts in the `idle` state, and waits for a `send[id]?` signal from the task that needs to send this message. Then the payload is either split into segments, if it is greater than the configured segment size, or sent as one message.

The `request[link][id]!` and `grant[link][id]?` transitions capture the wormhole routing as specified in the section 5.6.8.7 of the SpaceWire standard [6]. When every link on the message path is acquired, the automaton enters the `transmission` state for the duration, which is defined based on the packet size and the configured SpaceWire bandwidth.

When the transmission is finished, the receiver of the message is notified with a `receive[id]!` signal, and all the claimed links are released.

Note that a message can only receive the `send[id]?` signal when it is in the `idle` state. Therefore, if the next instance of a message is ready to be sent while the previous is still being processed, the `send` signal will not synchronize, causing a deadlock. In a real network, the second message would be queued after the first one. However, this scenario potentially indicates a network congestion issue, so it is useful to detect it.

Our model introduces several assumptions:

- SpaceWire data rate is taken from [2] to be 100 Mbit/s (or 80 ns per byte)
- constant delay is 100 µs. This is based on our experiments with sending a SpaceWire packet over two routers
- every small message has a header of 15 bytes, and a segment of a large message has a header of 22 bytes (SpaceWire-IPC implementation detail).

These values are are not essential to our model, and can be changed as necessary. One Uppaal time unit is taken to be 1 µs, but it can be set to an arbitrary number of nanoseconds with the variable `TIME_UNIT_NS`.

3.4 Properties of Interest

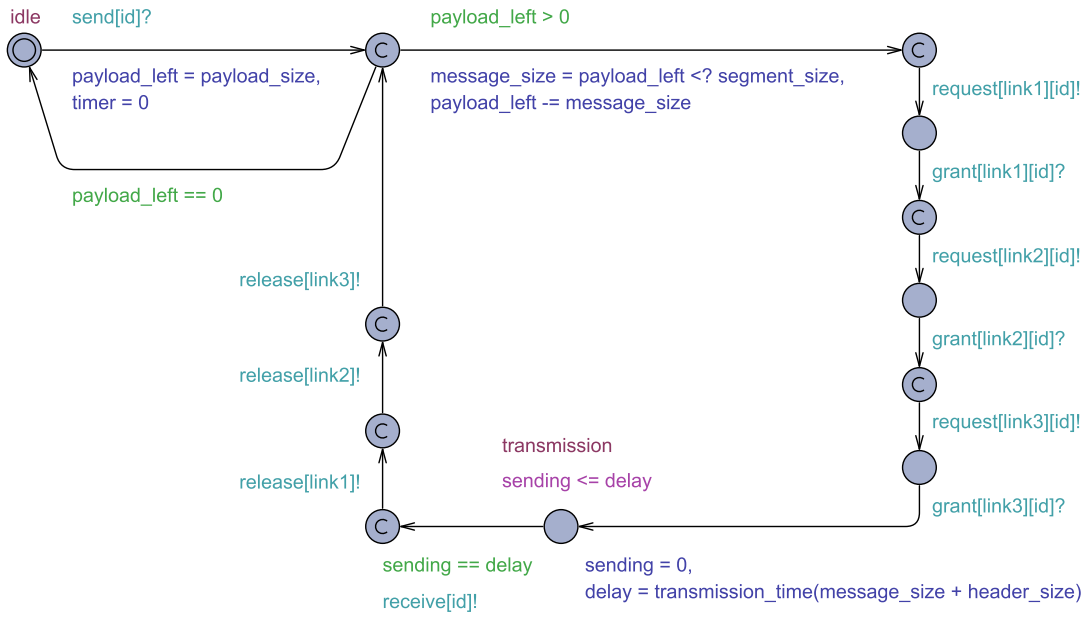
With a system model consisting of links, tasks and messages, we can verify a number of properties.

The most basic property is the deadlock freedom: `A[] not deadlock`. Some possible reasons for a deadlock are:

- a misconfigured model (e.g. a message without a receiving task, or claiming a non-existent link)
- a situation when a message is being transmitted, and its sender task is ready to send the next instance of this message
- a deadlock in the SpaceWire network, when several messages cannot proceed because they need to use already blocked links. This can happen in a ring topology with circular routing.

Secondly, we can check the upper bounds on message delivery times with `sup{Message.transmission}: Message.timer` for a property of interest. This gives us the maximal value of message `timer` (which is reset when the message is sent) when it is in the `transmission` state. For periodic tasks, a more efficient equivalent property is `sup{Message.transmission}: SenderTask.timer`, which does not use the message `timer` clock.

Additionally, there can be application-specific properties. For example, in a ScOSA system, the fault detection mechanism uses heartbeat and acknowledgement messages. We can check that the




```
// Message template parameters
const message_id id, const int payload_size, const link_id link1, const link_id link2, const link_id link3

// Declarations
clock timer, sending;
int delay;
chan send[N], receive[N];

const int TIME_UNIT_NS = 1000;              // One Uppaal time unit is 1000 ns (1 μs)

const int transmission_overhead_time = 100000 / TIME_UNIT_NS;
const int segment_header_size = 22;
const int small_message_header_size = 15;

const int segment_size = 10000;             // Segment payload size, bytes
int[0, segment_size] message_size;
int[0, payload_size] payload_left;

const int header_size = payload_size <= segment_size ? small_message_header_size : segment_header_size;

int transmission_time(int packet_size) {
    // 80 ns transmission time per byte
    return transmission_overhead_time + (packet_size * 80 / TIME_UNIT_NS);
}
```

Figure 5: Uppaal template for a message (above) and its declarations (below).

acknowledgement is always received before the next heartbeat: `A[] TaskHB.sending imply MessageAck.idle`. If this property is violated, a node can be wrongly considered failed due to a heartbeat or an acknowledgement message being delayed by other traffic. Furthermore, we can check the worst-case round trip time of a heartbeat and an acknowledgement with the following property: `sup{SinkAck.received}: TaskHB.timer`.

Table 1: ATON tasks and their parameters. Estimated periods of the message-triggered tasks are shown in brackets.

Task	Execution time [ms]	Period [ms]	Message size [B]
IMU	0.01	10	70
Altimeter	0.01	80	22
Cameras	0.01	50	1048614
Star tracker emulator	0.1	(10)	74
Feature tracker	20	(50)	15728
Undistortions	52	200	1048614
Crater navigations	696	700	146
Navigation filter	1	(10)	350
Flight controller	2	(10)	18
Evaluation module	1	(10)	-

4 EVALUATION

We evaluate our modeling framework on the ATON (Autonomous Terrain-based Optical Navigation) project [11]. In 4.1 we introduce the ATON system, then in 4.2 we verify different ATON configurations, and in 4.3 we discuss the scalability of our model, and the impact of certain parameters on the verification time.

4.1 ATON System

ATON is a DLR project taken as an example of a system of a realistic size. Fig. 6 reproduces its task graph and two possible configurations for a network of 4 nodes [12]. All the tasks are either periodic or triggered by an incoming message from another task. The details of the tasks are given in Table 1.

The 4-node ScOSA network, on which these tasks are running, is shown in Fig. 7. In addition to the application tasks and messages, there is a failure detection mechanism, which sends heartbeat and acknowledgement messages. Node N0 (the Master node) periodically sends heartbeats to all other nodes, and node N1 (an Observer) sends additional heartbeats to the Master. The size of heartbeats and acknowledgements is 16 bytes and 24 bytes respectively, and the period of heartbeats is taken to be 100ms.

4.2 Checking ATON Configurations

The goal of our model checking experiments is to verify that the configurations shown in Fig. 6 can run on the SpaceWire network in Fig. 7 such that all messages are delivered in time (within the task period), and to find an appropriate segment size for this system.

We are mainly focusing on the 'traffic minimization' configuration (Fig. 6, bottom left). Since we are interested in the network communication, we only model tasks that communicate over the network.

The complete network traffic in this configuration consists of 8 failure detection messages and 8 application messages, two of which are large (from Undistortion to Crater Navigation), as shown in Fig. 8.

To run these experiments, we constructed an ATON model of the templates described in Section 3. The full model is shown in Fig. 9 and is a composition of 61 state machines: 18 unidirectional links, 16 messages, 11 periodic tasks, 5 pipeline tasks, and 11 sink tasks.

The scenario which we are modeling is following. The system starts in the initial state (without any traffic), then periodic tasks start producing messages, which are then sent through links, possibly triggering other messages, and this continues indefinitely.

Although task execution time and message transfer duration are deterministic in our model, there is nondeterminism when multiple messages enter the network simultaneously, competing for links. In such cases one message is forwarded, and the others are blocked, causing downstream effects. This often arises when multiple tasks are triggered by the same periodic event.

We then use a model checker to traverse the whole tree of possible executions of the model, to prove or disprove its certain properties.

4.2.1 Bounds on segment size. We ran a series of experiments which showed that the lower bound on segment size is between 1 kB and 2 kB. The configuration with 1 kB segments failed the deadlock check because a large message U1 could not be completely transmitted within its period. The reason for this is the constant transmission overhead for every segment.

The upper bound on the segment size lies between 29 kB and 30 kB. For segments of 30 kB and more we identified the following problematic scenario. When the high frequency messages from Navigation Filter (NF1, NF2, NF3) appear in the network, there are messages HB1 and Ack0 already queued on the path N0 - R0 - R1 - N1. Each of these two messages has to wait for a large segment of U2 on a link R1 - N1. Furthermore, there is also a large segment of U1 in the queue, blocking the link N0 - R0. Then, when Navigation Filter messages can finally be sent, they are competing with each other on the link N0 - R0, and the first of these messages, NF1, has to wait for another U2 segment on the link R1 - N1. In this scenario, the Navigation Filter messages have to wait for 4 large message segments. If the size of each segment is 30 kB or more, the total waiting time exceeds the period of these messages, thereby causing a deadlock in the model.

4.2.2 Effect of segment size on message delivery times. The segment size significantly affects message delivery times. Figures 10 and 11 show worst-case message delivery times for small and large messages respectively for 10kB, 5kB and 3kB segments.

Changing the segment size from 10 kB to 5 kB decreases the worst-case delivery time approximately by half for many small messages. For the tasks that are not directly competing for links with large messages, the segment size has no effect (HB0, Ack1, Ack3, CN1, CN2). Further decreasing the segment size to 3 kB seems to have only marginal benefit for small tasks (and for HB1 even increases the time), but substantially increases the delivery times of large messages. Therefore, for this configuration, segment size of 5 kB appears to be a good choice.

Interestingly, the worst-case round trip times for a heartbeat followed by its acknowledgement is in all cases better than the sum of their individual worst-case times.

4.2.3 Checking the 'load balancing' configuration. We also checked the 'load balancing' configuration (Fig. 6, bottom right), and it turns out to be invalid for our SpaceWire network. In this configuration the Camera 2 task sends large messages with high frequency to the task Undistortion 2. This message flow alone exceeds the bandwidth

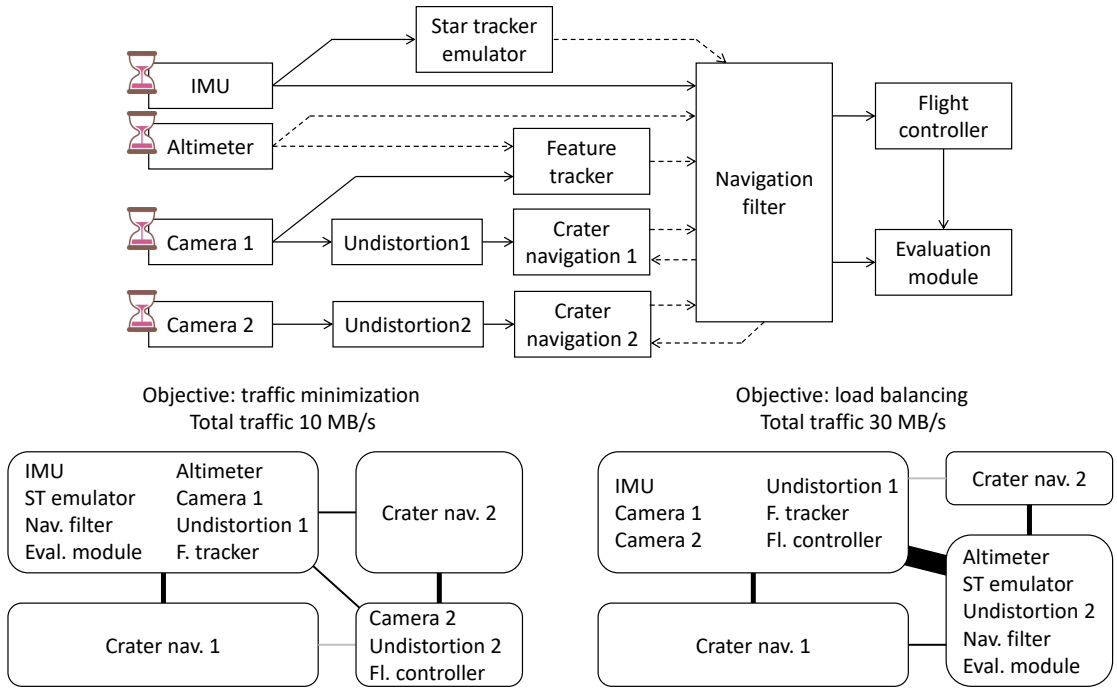


Figure 6: ATON task graph and two possible configurations for a 4-node network [12].

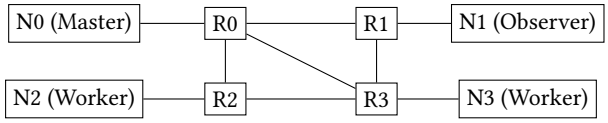


Figure 7: A network with 4 nodes and 4 routers for the ATON example.

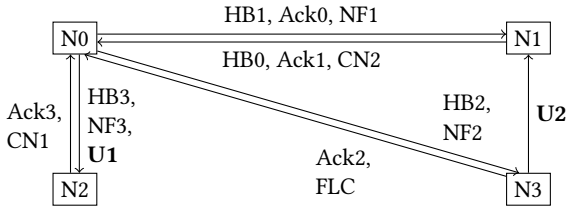


Figure 8: Network flows for the ATON 'traffic minimization' configuration. NF1, NF2, NF3 - messages from Navigation Filter to Crater Navigation 2, Flight Controller, Crater Navigation 1 respectively. Large messages (from Undistortion 1 and 2) are shown bold.

of the link. A solution would be to move Camera 2 and Undistortion 2 to the same node, which makes the configuration similar to the previous one.

4.3 Verification Time Sensitivity and Scalability

In this subsection we investigate how verification time depends on the number of message flows in the system, the segment size, and the heartbeat period. We used the same network configuration with 4 nodes and 16 message flows, and the property we are checking here is deadlock freedom. In most cases a deadlock check takes approximately the same time as a worst-case delivery time check for one message. All runs were made on a laptop with a 3GHz processor and 32 GB RAM using `uppaal64-4.1.25-5`. The time measurements are taken from the 'CPU user time used' report in `verifyta`.

Figure 12 shows that the verification time grows exponentially with the number of message flows, which is typical in model checking. The two large messages (U1 and U2) do not seem to particularly increase the verification time. Slight spikes occur when introducing a new period into the system (CN1, NF1). Configurations of 12 messages (including large) were checked in about 10 seconds, and checking all 16 messages took about 3 minutes.

We also investigated how changing the segment size and the heartbeat affects the verification time. Figure 13 (left) shows the effect of the segment size. Configurations with smaller segments take longer to verify because there are more segments that can interleave with the other messages in the network. Besides, for smaller segments the whole large message transfer time increases, and may overlap with more messages.

Figure 13 (right) shows the effect of the heartbeat periods on the verification time. Smaller periods tend to take more time to verify, especially 10 ms, which took 16 hours to check. This can be explained by the fact that shorter periods produce more possibilities for interactions with other messages. However, one message period itself should be not as important as the combination of periods and offsets of all messages in the system.

Figure 9: Complete model of the ATON 'traffic minimization' configuration composed of 61 UPPAAL state machines.

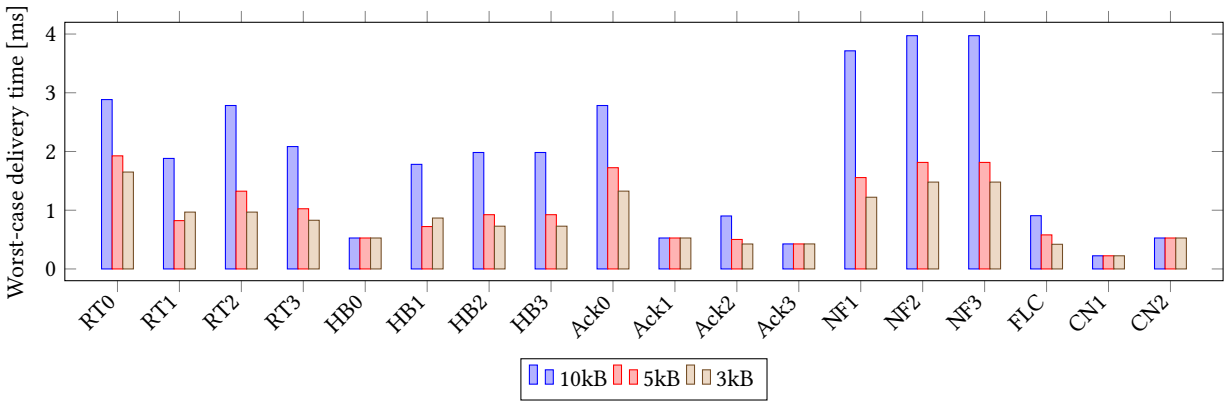


Figure 10: Worst-case delivery times for small tasks. RT - worst-case round trip duration of a heartbeat and an acknowledgement.

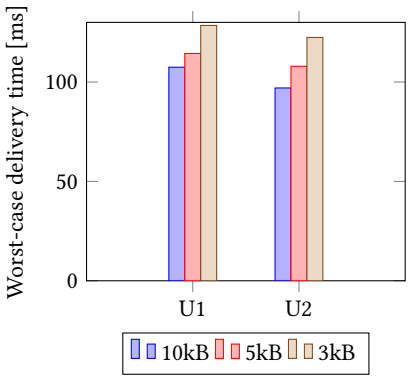


Figure 11: Worst-case delivery times for large tasks.

5 CONCLUSION AND FUTURE WORK

We presented a model checking framework for the UPPAAL model checker that allows modeling of an arbitrary SpaceWire network, in which periodic and event-driven tasks exchange messages. In particular, we focused on the segmentation of large messages into smaller packets. Our evaluation shows that our model can be used to detect problematic scenarios, and check system properties such as worst-case message delivery times.

We evaluated our approach on an example system with 4 nodes, 4 routers, and 16 message flows, in which we identified bounds on possible segment sizes, and the influence of the segment size on the worst-case delivery times. Verification time in our experiments ranged from several minutes to several hours for different scenarios, which indicates that our approach can be applied to small real-world systems.

In the future we plan to compare the results obtained by model checking to the behavior of the actual system running on the

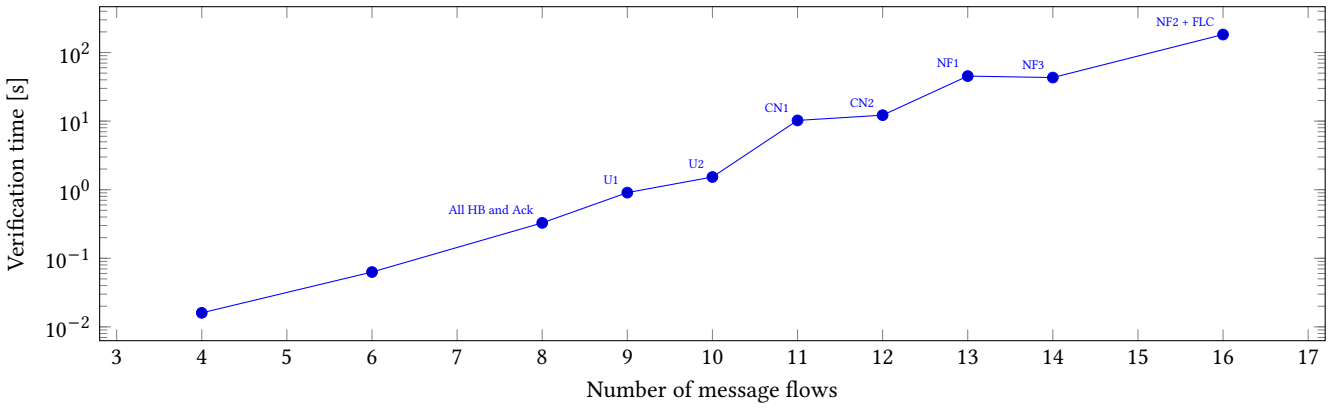


Figure 12: Scaling of the verification time with the number message flows in the system (segment size is 10kB, heartbeat period is 100ms).

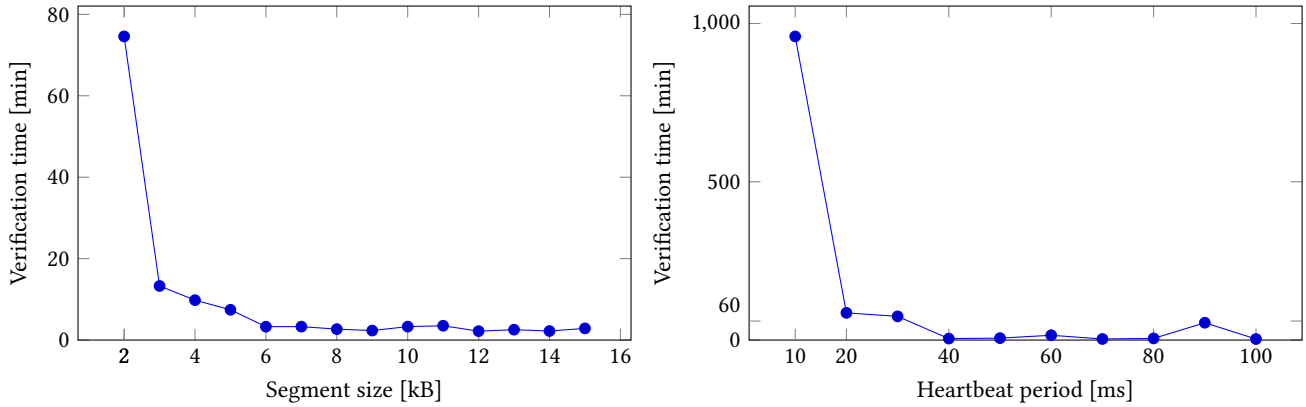


Figure 13: Verification time for different segment sizes (left) and different heartbeat periods (right).

SpaceWire network. Additionally, we would like to experiment with adding jitter to the model. This would make it closer to the real system, presumably at the cost of increased verification time. Other potential directions for future work are incorporating task scheduling, and a comparison with analytical methods such as network calculus.

REFERENCES

[1] Johan Bengtsson, Kim Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. 1996. UPPAAL — a tool suite for automatic verification of real-time systems. In *Hybrid Systems III*, Rajeev Alur, Thomas A. Henzinger, and Eduardo D. Sontag (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 232–243. https://doi.org/10.1007/BFb0020949

[2] Kai Borchers, Daniel Lüdtke, Görschwin Fey, and Sergio Montenegro. 2018. Time-Triggered Data Transfers over SpaceWire for Distributed Systems, In IEEE Aerospace Conference. *IEEE Aerospace Conference Proceedings*, 1–11. https://doi.org/10.1109/AERO.2018.8396435

[3] Alexandre David, Jacob Illum, Kim G. Larsen, and Arne Skou. 2009. Model-based Framework for Schedulability Analysis Using Uppaal 4.1.

[4] Ling Ling Dong, Yong Guan, Xiao Juan Li, Zhi Ping Shi, Jie Zhang, and Wei Hua. 2013. Model Checking for SpaceWire Error Detection Module. In *Industrial Instrumentation and Control Systems (Applied Mechanics and Materials, Vol. 241)*. Trans Tech Publications Ltd, 3020–3025. https://doi.org/10.4028/www.scientific.net/AMM.241-244.3020

[5] Jeffrey T. Draper and Joydeep Ghosh. 1994. A Comprehensive Analytical Model for Wormhole Routing in Multicomputer Systems. *J. Parallel and Distrib. Comput.* 23, 2 (1994), 202–214. https://doi.org/10.1006/jpdc.1994.1132

[6] ECSS-E-ST-50-12C Rev.1. 2019. *Space engineering - SpaceWire - Links, nodes, routers and networks*. Standard. European Cooperation for Space Standardization.

[7] Jérôme Ermont and Christian Fraboul. 2013. Modeling a Spacewire architecture using Timed Automata to compute worst-case end-to-end delays. In *Proceedings of 2013 IEEE 18th Conference on Emerging Technologies & Factory Automation, ETFA 2013, Cagliari, Italy, September 10-13, 2013*, Carla Seatzu (Ed.). IEEE, 1–4. https://doi.org/10.1109/ETFA.2013.6648072

[8] Thomas Ferrandiz, Fabrice Frances, and Christian Fraboul. 2009. A method of computation for worst-case delay analysis on SpaceWire networks. In *IEEE Fourth International Symposium on Industrial Embedded Systems, SIES 2009, Ecole Polytechnique Federale de Lausanne, Switzerland, July 8-10, 2009*. IEEE, 19–27. https://doi.org/10.1109/SIES.2009.5196187

[9] Thomas Ferrandiz, Fabrice Frances, and Christian Fraboul. 2011. Using Network Calculus to compute end-to-end delays in SpaceWire networks. *SIGBED Rev.* 8, 3 (2011), 44–47. https://doi.org/10.1145/2038617.2038627

[10] Thomas Ferrandiz, Fabrice Frances, and Christian Fraboul. 2012. A sensitivity analysis of two worst-case delay computation methods for SpaceWire networks. In *24th Euromicro Conference on Real-Time Systems, ECRTS 2012, Pisa, Italy, July 11-13, 2012*, Robert Davis (Ed.). IEEE Computer Society, 47–56. https://doi.org/10.1109/ECRTS.2012.35

[11] Tobias Franz, Daniel Lüdtke, Olaf Maibaum, and Andreas Gerndt. 2016. Model-Based Software Engineering for an Optical Navigation System for Spacecraft. In *Deutscher Luft-und Raumfahrtkongress*. Braunschweig, Germany. https://doi.org/10.1007/s12567-017-0173-5

[12] Andrii Kovalov, Elisabeth Lobe, Andreas Gerndt, and Daniel Lüdtke. 2017. Task-Node Mapping in an Arbitrary Computer Network Using SMT Solver. In *Integrated Formal Methods - 13th International Conference, IFM 2017, Turin, Italy, September 20-22, 2017, Proceedings (Lecture Notes in Computer Science,*

Vol. 10510), Nadia Polikarpova and Steve A. Schneider (Eds.). Springer, 177–191. https://doi.org/10.1007/978-3-319-66845-1_12

[13] Andreas Lund, Zain Alabedin Haj Hammadeh, Patrick Kenny, Vishav Vishav, Andrii Kovalov, Hannes Watolla, Andreas Gerndt, and Daniel Lüdtke. 2021. ScOSA system software: the reliable and scalable middleware for a heterogeneous and distributed on-board computer architecture. *CEAS Space Journal* (31 May 2021). https://doi.org/10.1007/s12567-021-00371-7

[14] Ping Luo, Rui Wang, Xiaojuan Li, Yong Guan, Hongxing Wei, and Jie Zhang. 2013. Model Checking for SpaceWire Link Interface Design Using Uppaal. In *2013 IEEE 37th Annual Computer Software and Applications Conference Workshops*. 181–186. https://doi.org/10.1109/COMPSACW.2013.56

[15] Ting Peng, Benjamin Weps, Kilian Höflinger, Kai Borchers, Daniel Lüdtke, and Andreas Gerndt. 2016. A New SpaceWire Protocol for Reconfigurable Distributed On-Board Computers: SpaceWire Networks and Protocols, Long Paper. In *2016 International SpaceWire Conference (SpaceWire)*. 1–8. https://doi.org/10.1109/SpaceWire.2016.7771624