Formal Verification and Analysis of Time-Sensitive Software-Defined Network Architecture

Weiyu Xu[1], Xi Wu[2], Yongxin Zhao[1*], and Yongjian Li[3]
[1] Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, Shanghai, China
[2] The University of Sydney, Australia
[3] State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China

***Abstract*—Safety-critical traffic in Industrial Internet of Things (IIoT) requires real-time communications with high fault tolerance, bounded latency and low jitter. Time-Sensitive Software-Defined Network (TSSDN), which combines the deterministic transmission of Time-Sensitive Networking (TSN) with the centralized management of Software-Defined Networking (SDN), was recently proposed to support the real-time requirement in IIoT. The research on TSSDN has been receiving increasing interests, however, the existing work has limitations including 1) the functional safety of TSSDN cannot be guaranteed; and 2) the effect of the separation of data plane and control plane on the time-sensitivity of TSSDN has not been evaluated. Therefore, in this paper, we employ the timed model checker UPPAAL to formalize the TSSDN architecture. Firstly, we use the build-in checker in UPPAAL to verify deadlock-free property, functional safety property and starvation-free property of our model. Then, the total latency of frames forwarding and scheduling within a single switch is measured based on the model. We focus on the latency overhead of frames requesting processing rules from the controller, which is on average an additioanl 180μs latency in the worst case, but the impact of this delay on the time-sensitivity of TSSDN is tolerable. As far as we know, this is the first paper providing a formal verification and analysis approach for TSSDN architecture, which could benefit for both TSSDN designers as well as the researchers.**

***Index Terms*—TSSDN Architecture, Real-Time Communication, Formalization, Verification, Timing Analysis**

I. Introduction

The fourth industrial revolution (i.e., industry 4.0) focuses on the integration of physical objects, humans and smart digital echnology in a sophisticated information network, which is also known as the Industrial Internet of Things (IIoT) [1]. With the rapid development of the IIoT, its safety-critical applications increasingly request both the real-time communication and run-time flexibility, which, however, cannot be both supported in the existing networking (e.g., Real-Time Ethernet) [2]. In 2016, Nayak et al. proposed the Time-Sensitive Software-Defined Network (TSSDN) [3], which integrates the deterministic and reliability of Time-Sensitive Networking (TSN) with the flexibility, heterogeneity and reconfigurability of Software-Defined Networking (SDN), to support the above requirements in the IIoT.

Recently, the research on TSSDN has been receiving increasing interests, which mainly focus on architecture design [4]–[6], control strategy [7]–[9] and latency analysis [10], [11]. Various solutions on how to combine TSN and SDN to form the TSSDN architecture have been discussed, among which our work is based on the one proposed by Böhm et al. [5]. Specifically, TSN and SDN utilize a unified control plane, therefore all network devices are managed by a centralized TSSDN controller. Based on the timing analysis evaluation, Nayak et al. [10] showed that TSSDN provided deterministic end-to-end latencies with low and bounded jitter for the time-triggered traffic on a specific benchmark topology. However, there are few work on the formalization of the TSSDN architecture, especially lacking the verification on its properties (e.g., safety and time-sensitivity) and formal analysis about how the separation of data plane and control plane affects the time-sensitivity of TSSDN.

In this paper, we use the model checker UPPAAL to formally verify and analyze the TSSDN architecture. We formalize the TSSDN controller and the switch as timed automata. Then, we verify the deadlock-free property, functional safety properties and the starvation-free property in our model via the UPPAAL build-in checker. Finally, we measure the total latency of frames forwarding and scheduling within a switch, and assess the additional latency of the frames requesting processing rules from the controller in the worst case. Our formal verification results illustrate that the TSSDN properties in the model are satisfied, and our timing analysis shows that the transmission over the TSSDN architecture is still time-sensitive to some extent. To the best of our knowledge, this is the first work on the formal verification and analysis of the TSSDN architecture.

The main contributions of this paper are:

1) **Formal Modeling.** We present a formal model of TSSDN architecture via timed automata in UPPAAL. Properties (e.g., functional safety property) can then be verified within the model. This approach can facilitate both researchers and designers to assess the time performance and validity of TSSDN architecture before deployment.
2) **Timing Analysis.** Our timing analysis mainly focuses on how the latency overhead caused by the separation of data panel and control panel affects the time-sensitivity in TSSDN. It is the first work on TSSDN architecture combining formal verification with analysis approaches.

The remainder of this paper is structed as follows. A brief introduction of the TSSDN architecture and model checker

*Corresponding author: yxzhao@sei.ecnu.edu.cn
DOI reference number: 10.18293/SEKE2022-094

UPPAAL has been given in Section II. In Section III, we present the formal model of the TSSDN architecture in UPPAAL. Section IV presents an experimental implementation with formal verification and timing analysis. Finally, Section V concludes this paper and presents the future work.

II. Preliminaries

A. TSSDN Architecture

As specified by the Open Network Foundation (ONF) [12], TSSDN architecture is divided into three layers, including the application plane, the control plane and the data plane, which can be found in Fig. 1. The application plane provides a

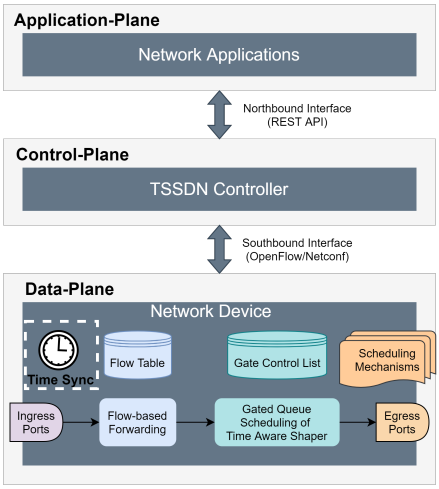

Fig. 1. TSSDN Architecture

programmable platform to users, which calls different services provided by the control plane through the northbound REST API. The control plane interacts with all network devices in the data plane to be aware of a global view of the network. According to the global state of the network, the centralized TSSDN controller can dynamically generate and distribute network configuration and control information to the data plane through the southbound API (e.g., Forces and OpenFlow [13]). The data plane consists of all network devices (i.e., bridges and switches). Its main responsibility is forwarding frames according to the rules in the flow table provided by the control plane.

Flow table is a finite set of flow table entries, which is used to control the forwarding of flows (i.e., sequences of frames with the same destination). According to the OpenFlow V1.0 [14], each flow table entry consists of $header\ fields$, $counters$ and $actions$. The head of each frame processed by the switch is compared against the $header\ fields$ of flow table entries. If a matching entry is found, any $actions$ for that entry will be performed on the frame (e.g., forward to a specified port, deliver to controller or drop). The controller is responsible for determining how to handle frames delivered from the switch. At the egress ports of network devices, the transmission order of different traffic will be determined based on different scheduling mechanisms to guarantee the QoS.

B. UPPAAL

UPPAAL [15], [16] is a well-known model checker, widely used in modeling, simulation and verification of Cyber Physical Systems (CPS), aerospace systems, real-time scheduling systems and other areas. In UPPAAL, a real-time system is modeled as a collection of timed automata with real-valued clocks. A timed automaton can be represented by a six tuple $M_{st} = (L, C, \Sigma, E, I, I_o)$, where

1) L is a set of locations;
2) C is the set of clocks;
3) Σ is a set of actions over edges, which could change the value of variables or clocks;
4) $E \subseteq L \times \beta(C) \times \Sigma \times 2^C \times L$ is a set of edges, where $\beta(C)$ is the set of conjunctions over simple conditions of clock constraints and 2^C is the set of clocks to be reset;
5) $I : L \rightarrow \beta(C)$ is a mapping of invariant on locations;
6) I_0 is the initial location.

TABLE I
CTL Operators

	Operator	Meaning	Operator	Meaning
Logical	&&	conjunction	||	disjunction
	!	negation	$\rightarrow$	implication
Temporal	[]	always	<>	eventually
	X	next	U	until
Path	A	all	E	exist

UPPAAL has a simulator for quantitative analysis and a verifier for model checking. The simulator validates the model via system execution, whereas the verifier checks properties such as safety and liveness described by Computational Tree Logic (CTL) via on-the-fly traversing the entire state space. As shown in TABLE I, logical, temporal and path operators in CTL can be used to formally describe system properties.

III. Modeling of the TSSDN Architecture

The overview model of TSSDN architecture, which can be found in Fig. 2, consists of generator model, flow table model, controller model and scheduler model. The formal models mentioned above are represented in UPPAAL as timed automata templates which are then instantiated as one or more processes as necessary, shown as follows:

```
system Generator_Process,
       Flow_Table_Process,
       Scheduler_Process,
       Controller_Process;
```

The generator process is used to simulate the frames that have entered the switch and wait for matching with the flow table. The flow table process and scheduler process, respectively, represent forwarding and scheduling within a single switch. The controller process communicates with the flow table process and handles frames that request processing rules.

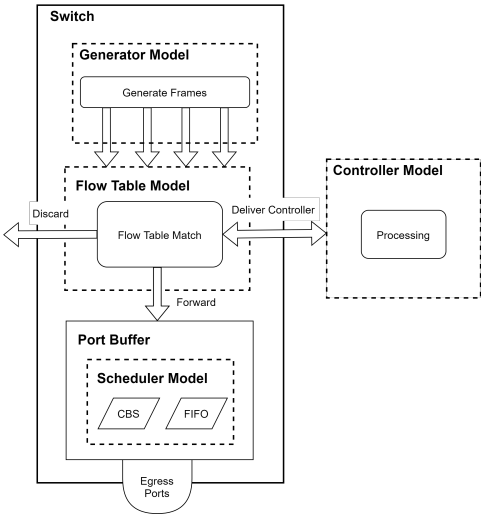

Fig. 2. An Overview Model of the TSSDN Architecture

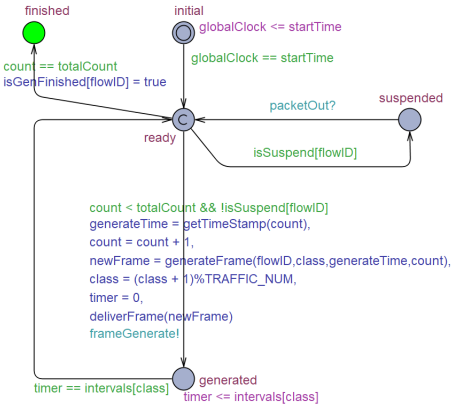

Fig. 3. Timed Automata of Generator Model

A. Generator Model

IEEE 802.1Qbv divides data frames into Control Data Traffic (CDT), Audio/Video Traffic (AVB_A and AVB_B) and Best Effort Traffic (BE) [17]. The frames of the above four classes will be generated and delivered to the flow table model on a regular basis in the generator model.

$Definition\ 1\ (Frame)$. A frame is represented by a quadruple $(flowID, class, transTime, timeStamp)$, where

1) $flowID$ identifies the flow to which the frame belongs, that is, the abstraction of the destination (e.g., MAC address, IP address, or VLAN ID);
2) $class$ defines the class of the frame, i.e., CDT, AVB_A, AVB_B and BE;
3) $transTime$ represents the duration of the frame passing through ports. On the premise of a constant port transmission rate, it usually depends on the length of the frame;
4) $timeStamp$ stands for the generation time instant of the frame, which can be used for latency analysis.

We formalize the generator as a timed automata template in UPPAAL shown in Fig. 3. The automaton consists of five states: $initial$, $ready$, $generated$, $finished$ and $suspended$. The variable $timer$ is a timer used for generating frames on a regular basis and the variable $globalClock$ is a global clock used for the synchronization of all processes in the system. The variable $packetOut$ will be explained in the controller model. Other variables and functions are defined as follows:

1) $flowID$ should be specified when instantiating a generator process because $flowID$ of the frames generated by each generator is presumed to be the same;
2) $class$ stands for the traffic to which the frame to be generated belongs, and `TRAFFIC_NUM` specifies the number of traffic classes;
3) $totalCount$ represents the total number of frames to be generated;
4) $count$ is used to record the number of frames that have been generated;
5) $frameGenerate$ is a synchronization channel used to notify the flow table process to start matching the delivered frame with the flow table;
6) $intervals[...]$ denotes the generation intervals of different classes of frames;
7) $isSuspend[...]$ indicates whether the generator needs to be paused;
8) $isGenFinished[...]$ indicates whether all frames have been generated and delivered;
9) $generateFrame(...)$ is a function used to generate a new frame which will be stored in $newFrame$;
10) $deliverFrame(...)$ is a function, which is responsible for delivering a frame to the flow table model.

The generator process is initially in the $initial$ state. When $startime$ equals $globalclock$, it enters the $ready$ state and begins generating the first frame. After that, the process alternates between $ready$ state and $generated$ state to periodically generate and deliver frames. When the value of $isSuspend[...]$ is $true$, the process will switch to the $suspended$ state and suspend working. This is typically caused by frames delivered by this generator waiting for processing rules from the controller. The generator process will migrate to the $finished$ state when $count == totalcount$, indicating that all frames have been generated and delivered.

B. Flow Table Model

In the flow table model, frames arriving at the switch will be matched with entries in a pre-configured flow table, which is defined as follows:

$Definition\ 2\ (FlowTable)$. A flow table is composed of multiple flow table entries, each of which consists of a triple $(header\ field, counter, action)$, where

1) $header\ field$ is used to match the $flowID$ of the frame;
2) $counter$ records the number of times a flow entry matches successfully;
3) $action \in \{\texttt{CONTROLLER}, \texttt{DROP}, \texttt{PORT\_A}, \texttt{PORT\_B}, ...\}$ indicates the action that the matched frame should take.

The timed automata template of the flow table model can be found in Fig. 4, which will be instantiated into a unique flow

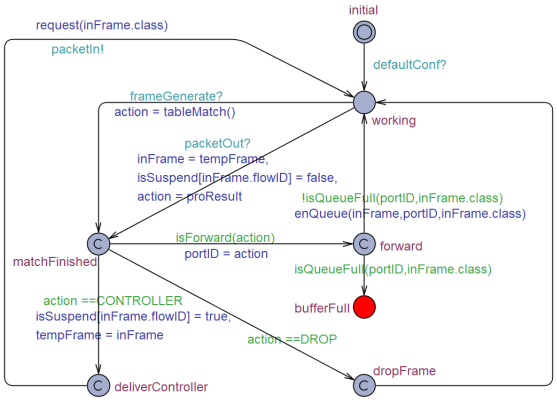

Fig. 4. Timed Automata of Flow Table Model

table process. After receiving the default flow table issued by the controller, the flow table process will move from $initial$ state to $working$ state. Whenever the process receives the synchronization signal $frameGenerated$ sent from generators, the delivered frame $inFrame$ will be matched with the flow table, then the process shifts to $matchedFinished$ state. According to the $action$ of the matched entry, the process will move to $forward$ state, $deliverController$ state or $dropFrame$ state. The forwarded frames will be placed in the matching egress port's buffer frame queues. Particularly, if the $action$ of the matched entry is `CONTROLLER`, $inFrame$ will be temporarily stored in $tempFrame$. Then the switch will notify the generator of the frame to suspend working and send a $packetIn$ message to the controller model requesting how to handle the frame. After receiving the $packetout$ message from the controller, $tempFrame$ will be discarded or forwarded to the specified egress port depending on the $proResult$ carried in $packetout$. Other variables and functions are defined as follows:

1) $tableMatch(...)$ is a function matching $inFrame$ with the flow table;
2) $isQueueFull(...)$ is a function checking the frame queue of a specific traffic in the port buffer is full. If the queue is full, the process will enter $bufferFull$ state;
3) $enQueue(...)$ is a function, responsible for enqueuing frames to the buffer queue of a specified port.

C. TSSDN Controller Model

The timed automata template of TSSDN controller can be found in Fig. 5. During the system initialization, instantiated controller process will send a default flow table to the switch and move to $working$ state. After receiving a $packetIn$ message from a switch, the controller will spend $maxProcessTime$ on calculating and generating a processing rule via function $processRule(...)$ for the requested frame based on the real-time global network state. The function can be defined by designers and the additional latency in the worst case can be measured based on $maxProcessTime$. Then the process will send a $packetOut$ message back to the switch to handle the frame waiting for the processing rule. It takes `PACKET_IN_TRANS_TIME` and `PACKET_OUT_TRANS_TIME` for $packetIn$ and $packetOut$ messages to pass through a port respectively. Here we assume that $inFrame$ is always included in these two messages. Therefore, the transmission time of these two messages depends on the maximum length of frames.

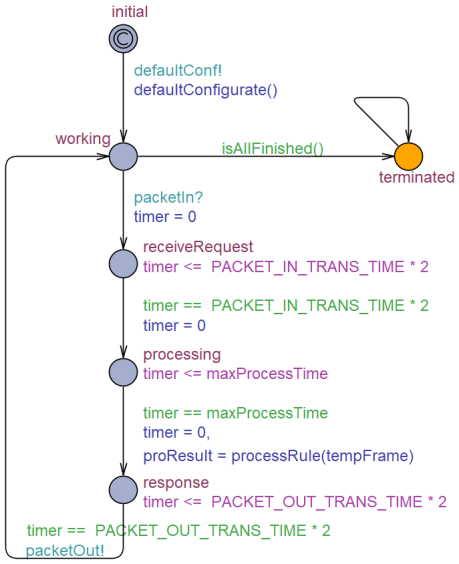

Fig. 5. Timed Automata of Controller Model

After scheduler process and all generator processes reach $finished$ state, controller process will move to $terminated$ state, indicating that the system has been ended. The remaining variables and functions are defined as follows:

1) $maxProcessTime$ represents the maximum processing time to calculate and generate processing rules, which is specified when instantiating the controller process;
2) $defaultConf$ is a synchronization channel used to active the flow table process;
3) $defaultConfigurate(...)$ is a function initializing system global variables and issuing the flow table.

D. Scheduler Model

According to IEEE 802.1Qbv, a scheduling period is divided into protected windows and unprotected windows. A guard band is introduced to prevent a new transmission of the non-time-critical traffic (i.e., AVB and BE traffic).

The timed automata template of the scheduler is shown in Fig. 6. A scheduler process will be employed to schedule frames according to different scheduling mechanisms for the frame queues at each egress port of the switch. The variables and functions are defined as follows:

1) $slots[...]$ is an array of time slots, whose values depend on the length of unprotected windows and protected windows;
2) $portID$ indicates the port associated with the scheduler;
3) $schClock$ is the clock of the scheduler, whose value is updated by function $updateClock(...)$;

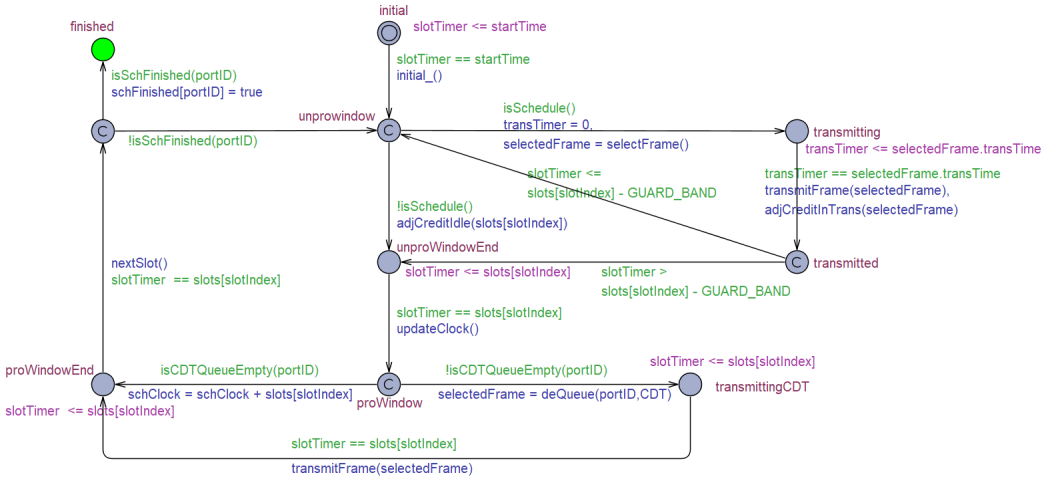

Fig. 6. Timed Automata of Scheduler Model

4) $schFinished[...]$ indicates whether all frames in the port buffer have been scheduled;
5) $selectFrame(...)$ is a key function used to schedule a frame from queues according to different scheduling mechanisms. CDT traffic is scheduled according to FIFO mechanism, whereas AVB traffic and BE traffic are scheduled based on credit-based shaping (CBS) mechanism, whose details are introduced in the Forwarding and Queuing enhancements for Time-Sensitive Streams (FQTSS) [18];
6) $adjCreditIdle$ is a function to update credits in CBS when no frame transmitted, while $adjCreditInTrans$ is to do so after a frame is transmitted;
7) $transitFrame(...)$ is a function used to calculate the latency of frames and to transmit frames through the egress ports of the switch;
8) $isSchedule(...)$ is a function to check whether scheduling a frame is allowed according to CBS mechanism.

When $slotTimer$ equals to $startTime$, the scheduler process transits from $initial$ state to $working$ state. Then the process enters three windows (i.e., unprotected, guard band, protected) in turn. CDT traffic can only be scheduled in protected windows, whereas AVB and BE traffic can be scheduled in unprotected windows. Once the process enters the guard band, it is not allowed to start the transmission of any frame. When all of the slots are exhausted, the process will return to the $working$ state and start a new cycle. Particularly, when all frames in the port buffer have been scheduled, the process will move to $finished$ state and stop scheduling.

IV. Formal Verification and Timing Analysis

In this section, we perform the formal verification and timing analysis over the formal model given in section III.

A. Experimental Configuration

We mainly focus on the transmission of four classes of data frames: CDT, AVB_A, AVB_B, and BE. The parameters of traffic classes are shown in Table II. For simplicity, the frame of each class is generated and delivered to the switch at a fixed interval and the transmission rate of the port is 100 Mbps in our experiment. In the table, $transTime$, which depends on the size of frame and the transmission rate of ports, indicates the transmission time of the frame through a switch port. $\beta^+$and β^- denote the increasing rate and the decreasing rate of credits in CBS mechanism, respectively.

TABLE II
Parameters of Traffic Classes

Class	CDT	AVB_A	AVB_B	BE
Length	300B	425B	275B	325B
transTime	18μs	34μs	22μs	26μs
Interval	100μs	30μs	40μs	40μs
Priority	4	3	2	1
β^+	-	3	4	-
β^-	-	4	3	-

In view of IEEE 802.1 TSN [19], [20], TSSDN period is set to 500μs in our experiment, which is divided into two unprotected windows and two protected windows. Note that the last 34 μs of each unprotected window is configured as a guard band, which is equal to the transmission time of the frame with maximum length.

In our experiment, four instantiated generator processes are employed to simulate four flows waiting to be processed by the switch. We instantiate a flow table process and two schedulers to simulate a switch with two egress ports (i.e., `PORT_1` and `PORT_2`). The flow table of the switch is shown in Table III. One controller process communicates with the flow table process and handles frames that request processing rules.

In particular, each frame of `FLOW_D` will be included in the $packetIn$ message to request processing rule from the controller. In order to minimize the average latency of `FLOW_D`, the controller process always decides to forward the frames of `FLOW_D` to the egress port with the fewest frames. Additionally, the length of $packetIn$ and $packetOut$ messages

TABLE III
FLOW TABLE OF THE SWITCH

Header Field	Counters	Action
FLOW_A	0	PORT_1
FLOW_B	0	PORT_2
FLOW_C	0	DROP
FLOW_D	0	CONTROLLER

is set as 450 bytes to accommodate the frame with maximum length, resulting in a transmission time of 36μs for both.

B. Formal Verification and Analysis

1) Formal Verification. We employ ten assertions to verify the model satisfies the following four properties.

Property 1: Deadlock-free Property

$A[\,]\ not\ deadlock$

This property asserts that the system will never progress into a deadlock situation, which is satisfied in our model.

Property 2: Starvation-free Property

$A <>\ scheduler.selectedFrame.class\ ==\ \texttt{CDT}$
$A <>\ scheduler.selectedFrame.class\ ==\ \texttt{AVB\_A}$
$A <>\ scheduler.selectedFrame.class\ ==\ \texttt{AVB\_B}$
$A <>\ scheduler.selectedFrame.class\ ==\ \texttt{BE}$

These assertions claim that frames of all classes will eventually be scheduled, whose results can be found in Fig. 7.

Property 3: Window Exclusive

$A[\,]\ scheduler.transmitting\ imply$
$\quad scheduler.selectedFrame.class! = \texttt{CDT}$
$A[\,]\ scheduler.transmittingCDT\ imply$
$\quad scheduler.selectedFrame.class == \texttt{CDT}$

The above assertions verify that CDT can only be scheduled in protected windows, which are satisfied in our model.

Property 4: Validity of Flow Table Matching

$A[\,]\ switch\_.dropFrame\ imply$
$\quad (action == \texttt{DROP}\ ||\ proResult == \texttt{DROP})$
$A[\,]\ switch\_.deliverController\ imply$
$\quad flowTable[matchedIndex][\texttt{ACTION}]$
$\quad\quad == \texttt{CONTROLLER})$
$A[\,]\ switch\_.forward\ imply\ (action == proResult$
$\quad ||\ action ==\ flowTable[matchedIndex][\texttt{ACTION}])$

These assertions claim that the switch always handles frames correctly according to the matched flow table entry. The variable $matchedIndex$ indicates the index of the matched flow table entry.

2) Timing Analysis. The latency of a frame within a switch, to be precise, is the time elapsed from the frame beginning to match the flow table entries to the last bit of the frame being transmitted from the egress port of the switch. We measure the latency of frames of three flows (i.e., FLOW_A, FLOW_B and FLOW_D) within the switch, whose results can be found in Fig. 8. Note that, frames of FLOW_C are not considered here because they will be discarded according to the flow table.

We can see from the figure that the latency of BE and AVB frames in the three flows is basically stable, and the latency of traffic with higher priority is lower overall. The latency of most BE and AVB frames in FLOW_D is higher than that in FLOW_A

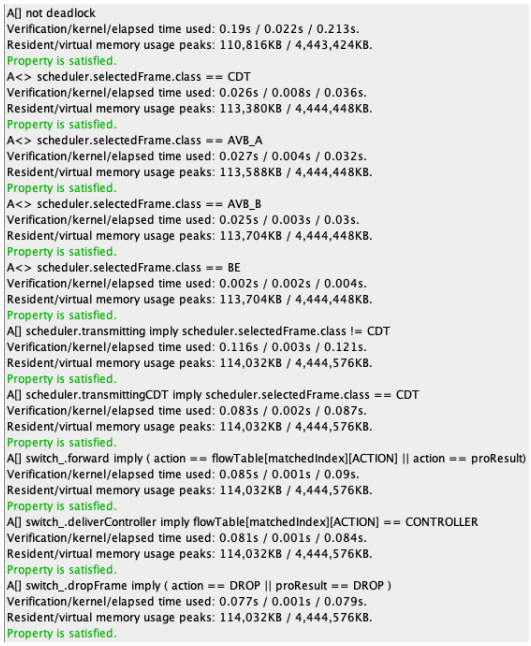

Fig. 7. Verification Results

TABLE IV
AVERAGE LATENCY OF FRAMES

Traffic\Flows	FLOW_A	FLOW_B	FLOW_C	FLOW_D
BE	205.2μs	219.4μs	-	330.8μs
AVB_B	128.6μs	137.5μs	-	329.4μs
AVB_A	128.8μs	103.5μs	-	340μs
CDT	626.3μs	820μs	-	447μs
Overall (exclude CDT)	154.2μs	153.6μs	-	333.4μs

and FLOW_B. However, the latency of CDT frames is much higher than that of frames of other traffic, and the latency of subsequent CDT frames continues to increase. Actually, this issue is caused by the division of time windows: there are only two protected windows in a scheduling cycle (i.e, a TSSDN period), and each protected window can only transmit one CDT frame. In other words, at most two CDT frames can be scheduled every 500μs, implying that a large number of CDT frames will be accumulated in the queue, resulting in a sharp increase in the latency of subsequent CDT frames.

Since our experiment mainly focuses on the additional latency of those frames that request the processing rules from the controller, CDT frames are excluded when calculating the overall average latency of the flows, which has little impact on our evaluation results. Table IV shows the average latency of various classes of frames in different flows, from which we find that the frames requesting the processing rule from the controller will incur an average of 180μs additional latency in the worst case.

V. CONCLUSION AND OUTLOOK

In this paper, we used the model checker UPPAAL to formally model and verify the TSSDN architecture. The results of verification demonstrate that the properties of the

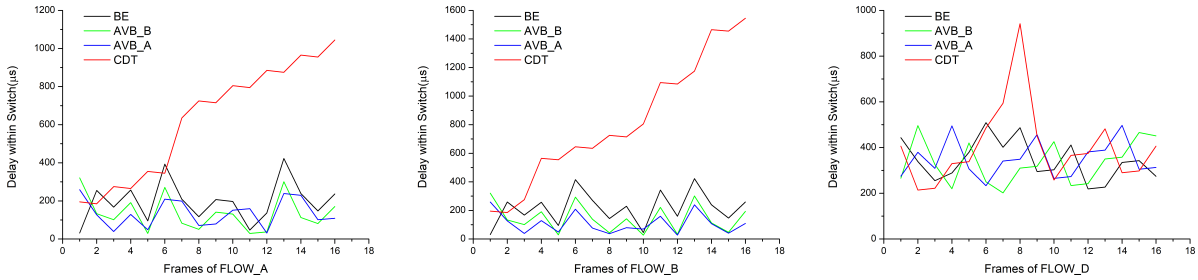

Fig. 8. Latency of Frames

TSSDN architecture are satisfied. Based on the model, we also performed a timing analysis and found that the frames requesting processing rules from the controller incurred an average of 180μs additional latency in the worst case, which shows that the transmission over the TSSDN architecture still retains time-sensitivity to some extent. By using our approach, both designers and researchers can conveniently verify the functional safety of the TSSDN architecture and assess the effect caused by the separation of data panel and control panel on time-sensitivity of TSSDN.

Limited by the size of the state space, our current model did not consider the process of frames entering the switch. State explosion is always a problem but can be mitigated by using abstraction and optimisation techniques (i.e. partial order reductions). [21] In the future, we would like to integrate the Per-Stream Filtering and Policing (PSFP) protocol into the model to capture the process by which frames are policed and filtered at ingress ports of the switch. Our future research will also focus on using various optimization techniques to enable scalability of the model.

ACKNOWLEDGEMENTS

This work is supported by Shanghai Science and Technology Commission Program under Grant 20511106002, Shanghai Trusted Industry Internet Software Collaborative Innovation Center and the Fundamental Research Funds for the Central Universities.

REFERENCES

[1] M. Wollschlaeger, T. Sauter, and J. Jasperneite, "The future of industrial communication: Automation networks in the era of the internet of things and industry 4.0," *IEEE industrial electronics magazine*, vol. 11, no. 1, pp. 17–27, 2017.

[2] L. Silva, P. Pedreiras, P. Fonseca, and L. Almeida, "On the adequacy of sdn and tsn for industry 4.0," in *2019 IEEE 22nd International Symposium on Real-Time Distributed Computing (ISORC)*. IEEE, 2019, pp. 43–51.

[3] N. G. Nayak, F. Dürr, and K. Rothermel, "Software-defined environment for reconfigurable manufacturing systems," in *2015 5th International Conference on the Internet of Things (IOT)*. IEEE, 2015, pp. 122–129.

[4] M. Bhm, J. Ohms, O. Gebauer, and D. Wermser, "Architectural design of a tsn to sdn gateway in the context of industry 4.0," in *23. ITG-Fachtagung "Mobile Communications", ISBN: 978-3-8007-4577-7*, 2018.

[5] M. Böhm, J. Ohms, M. Kumar, O. Gebauer, and D. Wermser, "Time-sensitive software-defined networking: a unified control-plane for tsn and sdn," in *Mobile Communication-Technologies and Applications; 24. ITG-Symposium*. VDE, 2019, pp. 1–6.

[6] T. I. ul Huque, K. Yego, C. Sioutis, M. Nobakht, E. Sitnikova, and F. den Hartog, "A system architecture for time-sensitive heterogeneous wireless distributed software-defined networks," in *2019 Military Communications and Information Systems Conference (MilCIS)*. IEEE, 2019, pp. 1–6.

[7] V. Balasubramanian, M. Aloqaily, and M. Reisslein, "An sdn architecture for time sensitive industrial iot," *Computer Networks*, vol. 186, p. 107739, 2021.

[8] T. Gerhard, T. Kobzan, I. Blöcher, and M. Hendel, "Software-defined flow reservation: Configuring ieee 802.1 q time-sensitive networks by the use of software-defined networking," in *2019 24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE, 2019, pp. 216–223.

[9] N. G. Nayak, F. Dürr, and K. Rothermel, "Incremental flow scheduling and routing in time-sensitive software-defined networks," *IEEE Transactions on Industrial Informatics*, vol. 14, no. 5, pp. 2066–2075, 2017.

[10] N. G. Nayak, F. Dürr, and K. Rothermel, "Time-sensitive software-defined network (tssdn) for real-time applications," in *International Conference*, 2016, pp. 193–202.

[11] D. Thiele and R. Ernst, "Formal analysis based evaluation of software defined networking for time-sensitive ethernet," in *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2016, pp. 31–36.

[12] O. N. Foundation. "sdn architecture". Tech. Rep. Issue 1, TR-502, 2014. [Online]. Available: https://www.opennetworking.org/wp-content/uploads/2013/02/TR_SDN_ARCH_1.0_06062014.pdf/

[13] N. K. Haur and T. S. Chin, "Challenges and future direction of time-sensitive software-defined networking (tssdn) in automation industry," in *International Conference on Security, Privacy and Anonymity in Computation, Communication and Storage*. Springer, 2019, pp. 309–324.

[14] O. S. Consortium *et al.*, "Openflow switch specification version 1.0.0," *http://www.openflowswitch.org/documents/openflow-spec-v1.0.0.pdf*, 2009.

[15] G. Behrmann, A. David, and K. G. Larsen, "A tutorial on uppaal," in *Formal methods for the design of real-time systems*. Springer, 2004, pp. 200–236.

[16] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, and W. Yi, "Uppaal—a tool suite for automatic verification of real-time systems," in *International hybrid systems workshop*. Springer, 1995, pp. 232–243.

[17] J. Lv, Y. Zhao, X. Wu, Y. Li, and Q. Wang, "Formal analysis of tsn scheduler for real-time communications," *IEEE Transactions on Reliability*, vol. 70, no. 3, pp. 1286–1294, 2020.

[18] I. S. Association *et al.*, "Ieee standard for local and metropolitan area networks—virtual bridged local area networks amendment 12 forwarding and queuing enhancements for time-sensitive streams," *IEEE Standard*, vol. 802, pp. 10 016–5997, 2009.

[19] "Ieee standard for local and metropolitan area networks–bridges and bridged networks," *IEEE Std 802.1Q-2014 (Revision of IEEE Std 802.1Q-2011)*, pp. 1–1832, 2014.

[20] S. Thangamuthu, N. Concer, P. J. Cuijpers, and J. J. Lukkien, "Analysis of ethernet-switch traffic shapers for in-vehicle networking applications," in *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2015, pp. 55–60.

[21] V. Klimis, G. Parisis, and B. Reus, "Towards model checking real-world software-defined networks," in *International Conference on Computer Aided Verification*. Springer, 2020, pp. 126–148.