TSN: Checker tool for output validation

Kelvyn L. M. Morais, Renan M. Silva, Aellison C. T. Santos, Vivek Nigam, Iguatemi E. Fonseca

***Abstract*— Time Sensitive Networking (TSN) provides high performance deterministic communication using time scheduling. Tools such as TSNsched are used to generate this time schedule given a topology and a set of constraints. The tool presented in this paper uses the schedule and log files generated by TSNsched and proves its reliability by evaluating its outputs in different criteria.**

***Keywords*— Checker, TSNSched, Time Sensitive Networking.**

I. INTRODUCTION

Time Sensitive Networking (TSN) is a recent OSI layer 2 network protocol standard (IEEE 802.1QV [1]) that extends Ethernet. It was developed to provide deterministic communication between devices in a local network. As it aims for guaranteed packet delivery, it is being considered in many safety-critical applications, with strict latency requirements, such as in the communication of safety-critical flows inside a vehicle [2].

Technically, TSN guarantees packet delivery by enforcing schedules using timeslots where only a specific type of flows are allowed to use. TSN schedules are enforced by the packet queuing machinery in TSN switches. Since only the traffic specified in the schedule can be sent in the network during these timeslots, there are no packet collisions and no need for re-transmission.

As the network size grows, designing TSN schedules becomes increasing more difficult. Indeed, it has been shown that this problem is NP-hard [2] and [3]. Developing TSN schedules manually becomes unfeasible and error-prone for larger networks. A much better approach is to use automated tools, such as TSNsched [4], developed by authors, that takes the network topology and flow specification, e.g., expected traffic, and requirements, e.g., maximum latency and jitter, and produce suitable TSN schedules.

However, a key problem in using such tools is that they are seen as black-boxes and since they contain thousands of lines of code and external tools, it is not clear whether their output can be trusted. Indeed, during the development of TSNsched, we encountered several mistakes on the schedules due to bugs which were subsequently corrected. This occurred despite we are using formal methods to generate schedules [4].

Kelvyn L. M. Morais e Renan M. Silva, Universidade Federal da Paraíba-UFPB, João Pessoa-PB, kmartinslenis30@hotmail.com, rmoreira@cc.ci.ufpb.br; Aellison C. T. Santos, Runtime Verification Inc., Urbana, Illinois, USA, cassimiroaellison@gmail.com; Vivek Nigam, Huawei Alemanha – Munique – Alemanha, PPGI. Prof. Colaborador no PPGI, UFPB, João Pessoa-PB, vivek.nigam@gmail.com; Iguatemi E. Fonseca, PPGI, UFPB, João Pessoa-PB,iguatemi@ci.ufpb.br. Este trabalho foi parcialmente financiado pelo CNPq e pela CAPES.

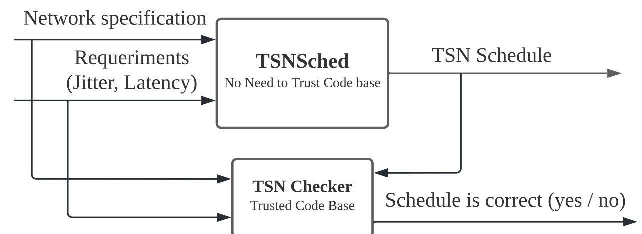

Fig. 1. Illustration of the TSNchecker integration with TSNsched.

This paper addresses the problem of checking whether TSN schedules are correct, that is, comply with TSN and satisfy the network specifications and requirements. In particular, we take inspiration on safety monitors. As safety monitors, we implemented a TSNchecker that, as illustrated by Figure 1. The TSNchecker contains much fewer lines of code (962 sloc) that perform the check of whether the TSN schedule produced is correct. This means that there is no longer need to trust the tool that generates the schedule (and its thousands of lines of code), but only trust TSNchecker and its 962 sloc lines of code. If the TSNchecker returns that the TSN schedule is correct, then it is safe to use it.

For the implementation of the TSNchecker, we made the following contributions:

- **Classification of Types of Checks:** We identified 9 different categories of checks related to the TSN standard and to the network specification/requirements.
- **Implementation and Tool-Chain Integration:** We implemented the checks in the TSNchecker and integrated into the TSNsched tool-chain. This means that TSNsched now calls the TSNchecker to validate the TSN schedule produced to checks for its correctness. If it is correct, then the schedule can be used. Otherwise, an error message is returned. The code for the TSNchecker and tool-chain can be found at [5].
- **TSNchecker Validation:** We validate TSNchecker with several benchmarks that we accumulated during the past years that we have been developing TSNsched. It demonstrates that TSNchecker successfully identifies errors in TSN schedules.

Finally, while we have used TSNchecker for validating TSNsched outputs, since TSNchecker uses friendly interfaces based on JSON (JavaScript Object Notation) inputs, it can also be used, in principle, to check TSN schedules produced by other tools. We leave this investigation to future work.

In Section II, we present the Classification of Types of Checks: the Syntax and semantic categories. Section III presents an implementation and results of our validation

tool. In section IV, we show the related works. In section V, we review our work and conclude talking about the future work.

II. Categories

The amount of validations needed can be overwhelming to a normal user, so we considered an approach to ease management and issue tracking. The approach used is to categorize the validations made by the tool into two major groups: Semantic and Syntactic. These categories are presented in more details in Sections II-A and II-B.

A. Syntactic

As the name suggests, this category validates the syntactic aspect of the files, for instance, misspellings and incorrect data types. This checker receives two files from the user and the first step is to ensure all data meet the requirements, such as data and file types to proceed the validations as needed for the semantic validation. Syntactic errors are easier to be noticed as they raise attention as soon as spotted; for instance, some letters in the place of a number.

1) **Data Type:** As mentioned earlier, a letter in the place of a number, and other situations where we do not get what is expected to configure a syntactic error. The checker verifies the data type of information contained on the file to ensure they are correct.
2) **File Type:** Just like the one before, this validation now aims for the files types. This checker receives a JSON (JavaScript Object Notation) and a log files that can have the .log format extension or a simple text (.txt) extension. So we make a quick verification to ensure they correspond to these extensions.

B. Semantic

The semantic validations mean the correctness of the data into a deeper analysis. The semantic differs from the syntax at adopting a more technical view of the situation, so it may not be so clear to the user that does not comprehend the behavior of the TSNSched, some errors may even pass unnoticed by experienced users as the topology grows too large. To help the user, this semantic validations target logical question, such as, whether the transmission of a packet respects the transmission window of its port.

1) **Switches:** A switch stores the data related to the cycle duration, the priorities assigned to flows, and the TSN time slots allocated for each priority at each port.
   - **Ports:** The ports contain important information for TSNSCHED, such as, the start of packet transmission, its cycle and transmission duration (also known as transmission window).
2) **Flow:** This is a topic which validates the flow data. There are on the JSON file the average latency, jitter, data time of the packets and hops to end devices. The complete details of each flow fragment can be found on the log file, which provides the flowfragment information needed for further verifications. The flow then can be divided into a couple more specific subtopics, they are: Paths and Packet Times. We discuss more about them in their respective subtopics.
   - **Paths:** The flows contain data about the hops from the origin device to the end device. It describes each hop from the first device to the end device by presenting the current and destination nodes, and the priority of the hop. Considering that, it might happen that a node may be missing or the scheduling inserts a node in the wrong place the checker validates the path to the end devices ensuring it follows as pretended.
   - **Packet Times:** A flow is broken into fragments, each fragment containing data of the hops. The JSON file includes the time when it left the previous node(*departure time*), the time it arrived at the current node(*arrival time*), and the time when it was sent to the next node(*scheduled time*). The log file presents more details of the flow fragment, such as the current hop (the origin and destination nodes) and the priority of the fragment. This information is the core of TSNsched, fitting them on a schedule without inconsistencies is vital for the deterministic network desired, nonetheless it is important to verify them to ensure correctness and reliability of the schedule.

III. Implementations and experimental results

The TSNSCHED provides a tool for scenario generation that generates a topology based on the configuration passed by the user [4]. The parameters for the configuration are the Number of Flows, Size of the Configuration (1 - small, 2 - medium and 3 - large), Packet Periodicity (2000 - normal, 1000 - high or 500 - very high) and Max Branching. We made a scenario with the following setup:

1) Number of Flows: 11.
2) Size of the Configuration: 2.
3) Packet Periodicity: 500.
4) Max Branching: 2.

When TSNSCHED is executed, we can set up how the output is generated [4]. We opted to generate a JSON and Log files with the time of the packets included. Both of them containing data about the execution of TSN, such as the time the packets took to be sent and the interval in which it was sent. The Checker works based on six criteria defined for evaluation those are: Type checking value, Well formed hops, Consistent path nodes and transmission windows consistency which is divided into 3 parts the cycle, window and sent order. They are further explained in their respective subsections. Table I shows a resume of the criteria.

A. Type checking value

Checks for any negative number on the output because it is an impossible value of time when considering the real

TABLE I
CHECKER CRITERIA OF VALIDATION.

Validation	Description
Type checking value	Checks whether all timestamps of time of departure, arrival and scheduled are positive.
Well-formed hops	Checks if the sum of the Time of sent and Duration time of transmission is equal to the scheduled time.
Consistent path nodes	Checks the path to a certain device given on the flow and compares it through the file to ensure it is consistent.
Transmission windows consistency - Cycle	Checks if a packet does not overpass its cycle and if two packets are being transmitted at same time on the same port.
Transmission windows consistency – Priority Window	Checks if each packet is being transmitted at its correspondent priority window.
Transmission windows consistency – Sent Order	Check if the packets at the same port of same priority are being transmitted in arrival order(FIFO like).

world scenario because everything timestamp is relative to at least its own start, therefore the minimum value is 0. Both Log and JSON files brings us the data needed to verify, those are the departure, arrival, scheduled, Slot Duration and Slot Start times as can be seen in Figure 2 and Figure 3.

```
{
   "flow0Fragment1": [
    {
      "packet0ArrivalTime": 8.576,
      "packetNumber": 0,
      "packet0DepartureTime": 0.576,
      "packet0ScheduledTime": 9.152
    }
  ]
},
{
 "flow0Fragment3": [
    {
        "packet0ArrivalTime": 17.152,
        "packetNumber": 0,
        "packet0DepartureTime": 9.152,
        "packet0ScheduledTime": 17.728
        }
    ]
},
```

Fig. 2. Packet Arrival, Departure and Scheduled times.

B. Well formed hops

TSN schedules and controls the transmission of packets on the flows based on time, as it is a time-sensitive network, therefore time is the most important subject to be considered. It is crucial that the flows respect their schedules to minimize latency and jitter. On the output there are three variables of time on the flow fragments, they are: the arrival time, departure time and scheduled time. Those three are chained together since the arrival time represents when the packet arrived at the current node, the departure time represents when the packet left the previous node and the scheduled time represents when the packet is to be sent to the next node. The tool checks if the nodes are well scheduled by comparing this information of a node with its previous, more precisely, there is a need to ensure that the departure time of a node is equal to the scheduled time of the previous node as seen in Figure 2 otherwise there is a scheduling problem.

Although it is not critical, it is good that a port remains open until the transmission is done which is can be verified by the adding of slot start and slot duration (slot start + slot duration = scheduled time) otherwise a warning is generated.

C. Consistent path nodes

At runtime, it might happen that TSNSCHED inserts a node on the path to a certain device that is not supposed to exist. Therefore, checking whether the paths are consistent with its schedule is important to avoid packets from following a route not intended, which is to raise the jitter. This tool validates it by comparing the path for each device on each flow with each hop on each *flow fragment*, so we can ensure they are the same.

For instance, assume that the path to a device called *dev5* has its origin on *dev38* and passes through *switch7*, *switch0* and *switch1* finishing at *dev5*, we expect that each *flowFragments* associated to this path to be consistent with the one described earlier.

```
 Path to dev5:
dev38,
switch7(flow1Fragment1),
switch0(flow1Fragment2),
switch1(flow1Fragment3),
dev5,

Fragment name: flow1Fragment1
        Fragment node: switch7
        Fragment next hop: switch0
Fragment name: flow1Fragment2
        Fragment node: switch0
        Fragment next hop: switch1
Fragment name: flow1Fragment3
        Fragment node: switch1
        Fragment next hop: dev5
```

D. Transmission windows consistency - Cycle

There cannot be two or more packets being transmitted at the same time on the same port, and the transmission cannot overpass its cycle duration in order to avoid delay problems on the network. For instance, if a packet is to be sent somewhere between 0 and 500 microseconds, it cannot be sent after this interval which is to be paused and then resumed afterwards based on the priority policy. Each switch have an array of ports, each containing an array of slots. Inside *prioritySlotsData* there is the *slotsData* for each

```
"prioritySlotsData": [
    {
      "slotsData": [
        {
          "slotDuration": 0.576,
          "slotStart": 9.115
        }
      ],
      "priority": 7
    },
    {
      "slotsData": [
        {
          "slotDuration": 0.577,
          "slotStart": 8.843
        }
      ],
      "priority": 7
    }
  ]
```


Fig. 3. Each slot data of a port.

transmission, it shows us the start and the duration of the transmission. The purpose here is to validate each *slotsData* to ensure their start is not the same and also if they do not finish their transmission after the cycle is over. Figure 3 is an example of the prioritySlotsData brought on the JSON. Assume we have a certain port with a cycle duration of 20 microseconds and two slots of transmission, we must ensure that both slots are started at different moments and that they will not last until after the 20th microsecond.

E. Transmission windows consistency - Priority Window

Each packet has a priority assigned to it, and so it must be transmitted at its correspondent priority window. This checker parses the JSON file looking for the flows and switches data to get the priority and node of each hop. By parsing the flow data, it takes each hop and save its priority, then compares whether there is a port on a specific switch transmitting a packet with the saved priority.

For example, assume there is a hop from the *switch9* to the *switch1* with a *priority* assigned of 7, there is need to be a port on the *switch9* with a transmission of priority: 7 as can be seen in Figure 4 and Figure 3.


```
flows": [
  "hops": [
    {
     "nextNodeName": "switch1",
     "currentNodeName": "switch9",
     "priority": 7
     },
]
```


Fig. 4. Hop from a switch1 to a switch9 with a priority assigned of 7.

F. Transmission windows consistency - Sent Order

When there are multiples packets on a port with the same priority, we must ensure they are being sent in the order of arrival just like a queue, where the first that comes in is the first that goes out. This tool validates it by checking the start of transmission of each slot data on a port and compare them. An array indexes its items in the order they are added, so we can assume the index of the array represents the order of arrival, based on that, the first item must have a *slotStart* lesser than the next one and so on. Figure 3 shows an example of a forwarding packet situation where two slots have a priority of 7, but the first of the queue is set to be started after the second one and they both have the same priority, this is to raise an error because the first one have arrived earlier and so must be sent before the next one.

G. Results

We generated a scenario to show the implementation of each criteria of validation, then we measured the time each method took to be executed, as can be seen in Table II. One problem was found and raised an error on this scenario which have not passed in the Sent Order criteria as explained on III-E and shown in Figure 3.

TABLE II
EXECUTION TIME FOR EACH METHOD ON THE SCENARIO TESTED.

Method	Execution time (ms)
Well Formed Hops	452
Check Logs	683
Port Transmission	1
Transmission Window	0
Time Packets	2097
Check Hops	12
Priority Window	7649
Packets ASAP	346
Total Program time	12312

This scenario is one of several that were tested. The result for the others scenarios can be seen on Table III, Figure 5 and Figure 6. In Figure 5 is illustrated in a graphic the amount of problems and warnings raised for each file, the one used to explain our implementations throughout the paper is identified as file *F*. In Figure 6 we present the mean time in milliseconds that each method took to be executed and also the program's total execution time which is way less than what a human would take to check each criteria manually.

TABLE III
PROBLEMS AND WARNINGS FOUND FOR EACH FILE TESTED.

File	Problems Found	Warnings
A	0	9
B	0	13
C	0	3
D	0	4
E	0	12
F	1	8
G	0	11
H	0	12

Fig. 5. Amount of problems and warnings found for each file.

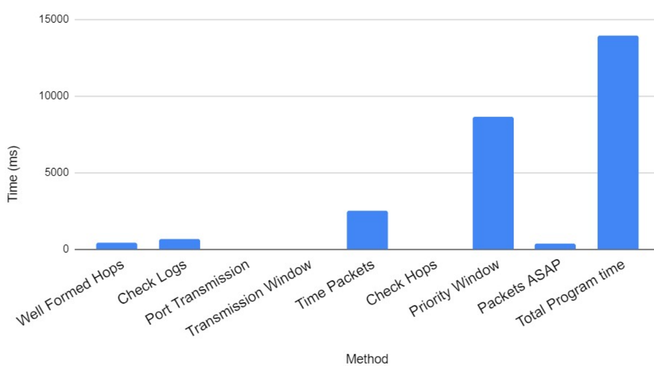

Fig. 6. Illustration of the overall execution time for each method.

IV. Related Work

In general, validation is often an important subject of study, as it can help to avoid or fix any problem that may occur. There are several others studies in the area of computer-aided verification and checkers as shown on Conference on Computer-Aided Verification [6] which most of them are not actually TSN specific, however they served to inspired our work. M. Illarramendi et al. [7] presented a tool that enables the developers to generate automatically reflective UML State Machine controllers and the Runtime Safety Properties Checker (RSPC) which checks a component-based software system's safety properties defined at design phase. The checker detects when a system safety property is violated and starts a safe adaptation process to prevent the hazardous scenario. Thus demonstrating that the safety of the system is enhanced.

Aravind Machiry et al. [8] presented on their work a fully-automated static analysis tool capable of performing general bug finding using both pointer and taint analyses that are flow-sensitive, context-sensitive, and field sensitive on kernel drivers. A more close related work is presented by Tatjana Kapus [9] which employed a probabilistic model checker using PRISM that work on network simulation by receiving the network as a kind of state machine and the queries about the probabilities sought in the form of logical formulas and then calculates the probabilities.

V. Conclusion and Future Work

We had ran this tool over several scenarios, each being generated with different settings to validate a great variety of scenarios and complexities to measure Checker's overall performance on finding problems, whereas no errors were found among the many files we have checked. It did raise some warnings.

We found this tool to be very useful at quickly looking for problems on the schedule by looking at its output, allowing us to quickly work on fixing them to improve the scheduling reliability and to validate TSNSCHED when running large scenarios where it would be unfeasible and unpractical to check manually. The results found were very satisfying, and we will continue working on developing new criteria to extend our validation versatility to improve the checker's usefulness and worth. As TSNSCHED is constantly being updated, so will be the checker.

References

[1] *IEEE 802.1Qbv - Enhancements for Scheduled Traffic*. Available online at http://www.ieee802.org/1/pages/802.1bv.html, last accessed on October 30th 2021. IEEE.

[2] Silviu S Craciunas and Ramon Serna Oliver. "Combined task- and network-level scheduling for distributed time-triggered systems". In: *Real-Time Systems* 52.2 (Mar. 2016), pp. 161–200.

[3] Wilfried Steiner. "An Evaluation of SMT-Based Schedule Synthesis for Time-Triggered Multi-hop Networks". In: *2010 31st IEEE Real-Time Systems Symposium*. 2010, pp. 375–384.

[4] Aellison Cassimiro T. dos Santos, Ben Schneider, and Vivek Nigam. "TSNSCHED: Automated Schedule Generation for Time Sensitive Networking". In: *2019 Formal Methods in Computer Aided Design (FMCAD)*. 2019, pp. 69–77.

[5] "Kelvyn Lenis M. de Morais". "TSN-Checker". In: Checker's source code available on https://github.com/KelvynLenis/TSN-Checker.

[6] Hana Chockler and Georg Weissenbacher, eds. *Computer aided verification*. en. 1st ed. Lecture notes in computer science. Cham, Switzerland: Springer International Publishing, July 2018.

[7] Miren Illarramendi et al. "CRESCO Framework and Checker: Automatic generation of Reflective UML State Machine's C++ Code and Checker". In: *2020 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. 2020, pp. 25–30.

[8] Aravind Machiry et al. "DR. CHECKER: A Soundy Analysis for Linux Kernel Drivers". In: *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, Aug. 2017, pp. 1007–1024. ISBN: 978-1-931971-40-9.

[9] Tatjana Kapus. "Using PRISM model checker as a validation tool for an analytical model of IEEE 802.15.4 networks". In: *Simulation Modelling Practice and Theory* 77 (2017), pp. 367–378. ISSN: 1569-190X.