Checking validity of the min-plus operations involved in the analysis of a real-time embedded network

Marc Boyer, Pierre Roux, Hugo Daigmorte

Checking validity of the min-plus operations involved in the analysis of a real-time embedded network

Marc Boyer[1], Pierre Roux[1],
and Hugo Daigmorte[2]

[1] ONERA / DTIS – Université de Toulouse
F-31055 Toulouse – France
[2] RealTime-at-Work
F-54600 Villers-lès-Nancy – France

Abstract. The network calculus theory is widely used to check that a network satisfies its real-time requirements. Such checking involves a lot of computations in the min-plus dioid. This paper shows how such computations can be formally checked using the Coq proof assistant in a realistic industrial context.

1. Introduction

Network calculus is a theory widely used in industry to check that a real-time network (like AFDX or TSN) provides guaranteed latency bounds to the real-time data flow [7, 8, 15]. When these networks are embedded in a critical system, where faults can lead to severe damages or injuries, a very high level of confidence on these bounds must be provided. Since such bounds are commonly computed by a dedicated tool [21], the correctness of the bounds depends both on the correctness of the algorithms and the correctness of their implementation.

The correctness of the algorithms is commonly ensured by open publication and peer-reviewing while the correctness of the implementation is mainly ensured by code review and tests. More confidence in the implementation can be insured by developing several pieces of software and comparing on the fly that all programs give the same result. Nevertheless, such approach can not detect an error in the algorithm itself [11]. Using proof assistants, such as Coq or Isabelle/HOL, is a way to increase the confidence in a software: one may prove the correctness of the algorithms and derive an implementation from the proof [13]. One may also use a *skeptical approach*, where the proof assistant is not able to check an algorithm or to compute a result, but still able to check that a result is correct. Indeed, checking a solution is commonly much easier that solving a problem (for example, finding vs checking the roots of a polynomial, the decomposition into prime factors, matrix inversion,...). Of course, this approach can be used only for software used at design: if there is a mistake in the algorithm or a bug in the implementation, it will only be detected at runtime, while a completely proved approach will ensure that the software is conform to its specification. Nevertheless, the skeptical approach often requires significantly less effort.

Network calculus is based on the min-plus dioid theory [3], and analyzing a network involves a lot of operations in this theory (like physics involves a lot of matrix manipulations). In a recent work [17], this skeptical approach has been applied to the min-plus dioid of real functions. This paper shows how this approach can be used in an industrial context.

2. Network calculus

Network calculus is a theory designed to compute upper bounds on delay an memory usage in networks. Data transiting through the network are called flows. A flow is modeled by *cumulative curves* at each point in the considered network, $A : \mathbb{R}^+ \to \mathbb{R}^+$ where $A(t)$ represents the cumulative amount of data observed in the flow up to time t at a given point of the network. Possible cumulative curves are specified by envelopes called *arrival functions*. A flow A satisfies an arrival function α when: $\forall t, d \geq 0 : A(t+d) - A(t) \leq \alpha(d)$. For instance, a periodic flow sending frames of size L every T time unit admits as arrival curve $\nu_{L,T} : d \mapsto L \left\lceil \frac{d}{T} \right\rceil$, where $\lceil \cdot \rceil : \mathbb{R}^+ \to \mathbb{N}$ is the ceiling function. All network elements are modeled by *servers*. A n-server S transforms n input flows $(A_1, \ldots, A_n)$ into n output flows $(D_1, \ldots, D_n)$, where all A_i and D_i are cumulative curves. The performance of these servers is specified by *service curves*. A n-server S admits a strict service curve β when, for all busy interval $(t, t+d]$, the aggregate output is at least $\beta(d)$, i.e., $\sum_{i=1}^{n} D_i(t+d) - D_i(t) \geq \beta(d)$.

Among others, a result of network calculus states that, if a server S uses a FIFO policy, if it admits a strict service curve β and each of its incoming flow

A_i admits as arrival curve a function α_i, then the delay experienced by each flow in the server is upper bounded by

$$hDev\left(\sum_{i=1}^{n}\alpha_i, \beta\right) \tag{1}$$

(2)
$$\text{with } hDev(f,g) = \sup_{t\geq 0}\{\inf\{d \mid f(t) \leq g(t+d)\}\}.$$

Network calculus also uses other operators, like the the min-plus convolution $*$ and deconvolution $\oslash$, defined by: $\forall f, g : \mathbb{R}^+ \to \mathbb{R}^+, \forall t \in \mathbb{R}^+$,

$$(f * g)(t) = \inf_{0\leq s\leq t}\{f(t-s) + g(s)\}, \tag{3}$$

$$(f \oslash g)(t) = \sup_{s\geq 0}\{f(t+s) - g(s)\}. \tag{4}$$

3. Related work

To perform actual min-plus computations, one has to settle on a given class of real functions. Two main classes of functions are used in network calculus: the set of concave or convex piecewise linear functions, C[x]PL [18], and the, strictly larger, set of ultimately pseudo-periodic piecewise linear functions, commonly known as UPP [6]. The data structure and algorithms for the CPL class are so simple that they, to our knowledge, have never been published. Nevertheless, they cannot accurately model packetized traffic, whereas the UPP class gives more precise results at the expense of higher computation times [7]. The algorithms of the operators on the UPP class are given in [6].

An open source implementation of the operators on the C[x]PL class can be found in the DISCO network calculus tool [2]. An open source implementation of the UPP class has been developed [4] but is no longer maintained to our knowledge. The Real-Time Calculus toolbox (RTC) does performance analysis of distributed real-time systems [19, 20]. Its kernel implements Variability Characterization Curves (VCC's), a class very close to UPP. None of these implementations has been formally proved correct.

The first work on the formal verification of network calculus computation were presented in [14]. The aim was to verify that a tool was correctly using the network calculus theory. An Isabelle/HOL library was developed, providing the main objects of network calculus and the statement of the main theorems, but not their proofs. They were assumed to be correct, since they have been established in the literature for long. Then, the tool was extended to provide not only a result, but also a proof on how network calculus has been used to produce this result. Then, Isabelle/HOL was in charge of checking the correctness of this proof. The result was taking the form of an algebraic min-plus expression, yet to be computed.

Another piece of work, presented in [16], consists in proving, in Coq, the network calculus results themselves: building the min-plus dioid of functions, the main objects of network calculus and the main theorems (statements and proofs).

The tool CertiCAN [12] is able to produce Coq proofs for some real-time analyses of the CAN protocol. These analyses are not based on the network calculus theory.

The PROSA library also provides proofs of correctness for the response time of real-time systems, but focuses on scheduling tasks for processors [10].

4. Existing tools

4.1. RTaW-Pegase. RTaW-Pegase is a proprietary tool written in Java and developed by the company RealTime-at-Work that can provide network analyses and optimization. It can support many technology types and networks such as: automotive, aerospace and industrial Ethernet TSN, CAN (FD,XL), LIN, Arinc as well as wireless networks for off-board communication.

Its first functionality is to model a network, visualize it and set configuration parameters: priorities, offsets, routing, shapers, transmission schedule, preemption,... The graphical interface allows to manipulate the configurations as shown in Figure 1. In this example, the network is made of 5 switches (in blue) and 14 end systems (in orange) and one of the flows is shown, between "CAM1" and "ECU1".

In addition to timing-accurate simulation, RTaW-Pegase can compute guaranteed upper bounds on message delays using a state-of-the-art analysis in network calculus.

RTaW-Pegase can also generate complete traces of its analyses. These traces contain all the computations performed during the analysis in a format both readable by a human and that can be interpreted by an additional RTaW tool, the Network calculus interpreter. Figure 2 shows a simplified example of such a trace. This trace gives the calculations for two periodic flows (Flow1, Flow2) crossing the first link of a network, arbitrated with FIFO policy. The `assert` expression at last line checks that the computed value is not greater than the value 156/5 computed by RTaW-Pegase.

Thereby, a network calculus expert can read this trace and check that the mathematical operations performed correspond to the result of the network calculus theory. Using the calculation capability of the interpreter, it can perform other computations,

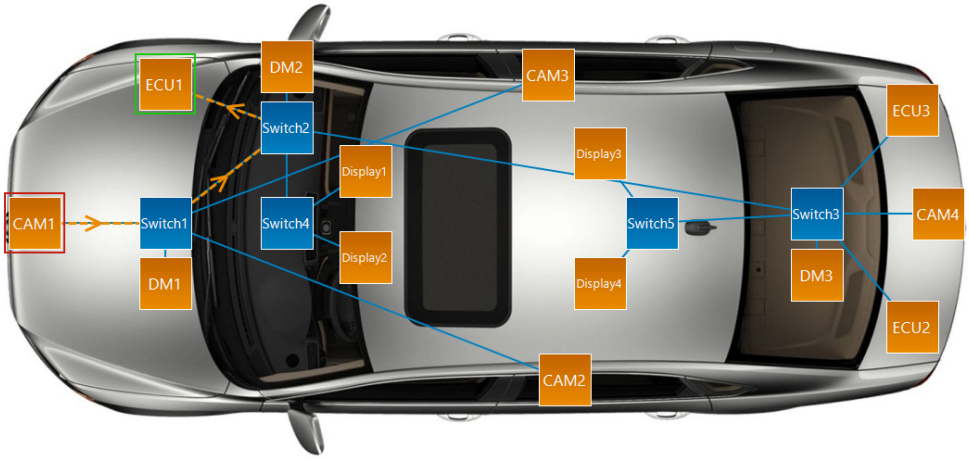

FIGURE 1. Example of visualization in RTaW-Pegase.

```
1 ##################################
2 # Time unit: microsecond
3 # Frame size unit: bit
4 ##################################
5 # Input flows
6 ##################################
7 # Flow1 entering at Node1>port-P1
8 Flow1 := stair(0,10000,1360)
9
10 # Flow2 entering at Node1>port-P1
11 Flow2 := stair(0,5000,1760)
12
13 ##################################
14 # EndSystem : Node1>port-P1
15 ##################################
16 # Computations at priority level 0
17 cumA_Node1 := zero
18
19 # Flow1 in Node1 -> P1
20 cumA_Node1 := cumA_Node1 + Flow1
21
22 # Flow2 in Node1 -> P1
23 cumA_Node1 := cumA_Node1 + Flow2
24
25 # Common service at level 0
26 S_Node1 := affine(100, 0)
27
28 # Common delay
29 d_Node1 := hDev(cumA_Node1, S_Node1)
30
31 assert(d_Node1 <= 156/5)
```

FIGURE 2. Example of a trace that can be produced by RTaW-Pegase

get more values and also plot the functions. An online version is available for non-commercial academic use [1].

4.2. Minerve. Let's look at the last `assert` on Figure 2. To perform a formal proof of this result within the Coq proof assistant, one needs to define the three functions `Flow1`, `Flow2` and `S_Node1`, state the property $hDev(Flow1 + Flow2, S\_Node1) \leq \frac{156}{5}$ and prove this property. Such a proof is given in Figure 3.

First, one loads our tool Minerve [17] (for MIN-plus ExpRession VErification) providing a formalization of the UPP class of functions and an automatic proof tactic to check min-plus computations on those functions. The next line simply instructs Coq to interpret all subsequent numeric constants as arbitrary precision rationals.

Then one needs to define the considered UPP functions. The first function `Flow1` is, as seen on line 8 of Figure 2, a stair function that is incremented by 1360 every 10000, starting from 0. The second function `Flow2` is also a stair function, as seen on line 11 of Figure 2, whereas the third function `S_Node1` is, still according to Figure 2 (line 26), a linear function of slope 100. These functions are encoded by their period `sequpp_d`, increment `sequpp_c` and an initial segment `sequpp_T` (to allow a specific initial behavior before the regular periodic one) and a list of linear segments following the pattern $(x, (y, (\rho, \sigma)))$ where (x, y) are the coordinates of the leftmost point of the segment, ρ is its slope and σ its limit on the right of x (discontinuities are allowed on the right of x, as seen for instance on `Flow1` where the function is $y = 0$ at $x = 0$ and immediately jumps to $\sigma = 1360$ just after 0). Not all combinations of parameters are valid. For instance the period must be positive. Such validity of the parameters is checked by the `F_of_sequpp` function.

Finally, the property to prove is expressed after the `Goal` keyword and automatically proved by our

```
Require Import minerve.tactic.
Local Open Scope bigQ.

Definition Flow1_js : @seqjs bigQ := [::
  (0,  (0%:E, (0,  1360%:E)));
  (5000,  (1360%:E, (0,  1360%:E)));
  (10000,  (1360%:E, (0,  2720%:E)))].
Definition Flow1 := F_of_sequpp {|
  sequpp_T := 5000;
  sequpp_d := 10000;
  sequpp_c := 1360;
  sequpp_js := Flow1_js
|}.

Definition Flow2_js : @seqjs bigQ := [::
  (0,  (0%:E, (0,  1760%:E)));
  (2500,  (1760%:E, (0,  1760%:E)));
  (5000,  (1760%:E, (0,  3520%:E)))].
Definition Flow2 := F_of_sequpp {|
  sequpp_T := 2500;
  sequpp_d := 5000;
  sequpp_c := 1760;
  sequpp_js := Flow2_js
|}.

Definition S_Node1_js : @seqjs bigQ := [::
  (0,  (0%:E, (100,  0%:E)))].
Definition S_Node1 := F_of_sequpp {|
  sequpp_T := 0;
  sequpp_d := 1;
  sequpp_c := 100;
  sequpp_js := S_Node1_js
|}.

Goal hDev_bounded (Flow1 + Flow2) S_Node1 (156/5).
Proof. nccoq. Qed.
```

FIGURE 3. Example of Coq proof

nccoq tactic [17]. This tactic is a reflexive tactic, which means it makes use of Coq efficient computation capabilities to perform proofs. This is key in getting a very pervasive tactic being entirely developed in Coq. Of course, the proofs automatically performed by the tactic offer the same strong correctness guarantees as any other handmade Coq proof.

It is worth noting the application of the *skeptical approach* here, the horizontal deviation $\frac{156}{5}$ is computed by RTaW-Pegase but only checked with Coq.

4.3. Reading the Coq Specification. To fully trust a Coq proof, one must inspect its statement in order to agree that the proof is indeed proving what the user is expecting. Indeed, whereas Coq can automatically check proofs, it cannot read the user mind to ensure the formal statement match its understanding by the user.

So lets inspect our freshly proved theorem. If one types

```
Unset Printing Notations.
Check hDev_bounded (Flow1 + Flow2) S_Node1 (156/5).
Set Printing Notations.
```

Coq reprints the statement with all notations expanded

```
UPP_PA_refinement.hDev_bounded (F_plus Flow1
  Flow2) S_Node1 (bigQ2rat (BigQ.div (BigQ.Qz
    (BigZ.Pos (BigN.N0 156))) (BigQ.Qz
    (BigZ.Pos (BigN.N0 5)))))
```

in particular, one can see that + was a notation for *F_plus*. One can for instance inspect the latter by asking Coq to print its definition

```
Print F_plus.
```

and Coq answers

```
F_plus = fun f g : F ⇒ f \+ g
```

By deactivating printing of notations again, one could see that \+ is the pointwise addition of functions.

Proceeding with such investigations, one can check that `hDev_bounded` indeed states that the horizontal deviation is bounded and that the definitions of functions introduced with `F_of_sequpp` perfectly match what one is expecting.

Digging further, one could even look at the definition of the field of real numbers $\mathbb{R}$ given by the library we are using.

5. Checking RTaW-Pegase traces with Minerve

The RTaW-Pegase tool allows to compute upper bounds on the delays on the flows crossing a network. As presented in Section 4.1, the engineer enters into the tool the network description, the flow characteristics, and the tool can apply network calculus to check that the system satisfies its latency requirements. The correctness of these bounds depends on a chain of responsibilities. The first link of the chain is the correctness of the network calculus theorems (for example, the result presented in (1)). In case of mistake in the proof, the confidence in the result collapses. Such issue have been addressed in [16]. A second link is the correct application of network calculus results by the tool: if the tool applies a result whose hypotheses are not satisfied by the system, the confidence also collapses. The trace generated by the tool, illustrated in Figure 2, has been designed to allow proofreading by a human expert. A formal proof can also be generated [14]. The last link is the exactness of the computation: the basic operations (sum,

convolution, deconvolution, *hDev...*) are too complex to be checked by humans. This is the aim of Minerve, presented in Section 4.2.

The approach presented in the current work boils down to checking that all computations done in a given RTaW-Pegase trace have provided a sound result, i.e., verifying that the transformations of expressions into values have been performed correctly.

But the traces generated by RTaW-Pegase have been designed to be read by a network calculus expert, to increase confidence in the operations, whereas Minerve has been designed for proof simplicity.

To make both tools run together, we had to solve a few issues:

(1) *Function syntax:* Both tools do not have the same syntax to represent functions. In particular, the domain of the non-periodic part (the initial prefix) of a function in Minerve is always a right open interval $[0, T)$, whereas it can be open $[0, T)$ or closed $[0, T]$ in the syntax of RTaW-Pegase.
(2) *Single assignment:* The RTaW-Pegase trace is a script in an imperative programming language, in which identifiers are variables, whose value can be updated along the script lines. On the opposite, Coq is designed to state mathematical definitions, where a given identifier represents the same value all along the proof. There are several ways to solve this problem, and we have basically chosen to resort on a single static assignment transformation.
(3) *Performance tricks:* While going from handmade examples, with dozens of operations, to realistic examples with thousands of operations, we faced some performance problems, not related to the core of the algorithms but related to some details in implementations.

5.1. Function syntax. One problems is that, although both tools represent functions as sequences of segments and spots, RTaW-Pegase's input language accepts segments that can be open or closed on both ends, whereas NCCoq always considers a sequence of spots and open segments.

Consider a function $f : \mathbb{R}^+ \to \mathbb{R}^+$, defined by

$$f(t) = \begin{cases} 1 & \text{if } t \leq 2, \\ t & \text{if } t > 2. \end{cases} \tag{5}$$

In RTaW-Pegase, such a function can be represented as a sequence of two segments, the first one going from point (0,0) to (2,0) with slope 0 (printed as `[(0,0)0(2,0)]`) and a second one going from (2,2), excluded, up to infinity with slope 1 (then printed as `](2,2)1(+Inf,+Inf)[`). In Minerve input, such a function is represented as a sequence of two pairs (spot, open segment), the first one having a spot (0,0) and a segment with origin 0 and slope 0, and the second one having a spot (2,0) and a segment with origin 0 and slope 1. Then, the translation from RTaW-Pegase to Minerve has to split the closed segment into a spot and an open segment, and push the right extremity as a spot of the second segment. Eventually, the translation can be done on a per-segment basis, pushing spots from one segment to another.

A second problem was that RTaW-Pegase may have no periodic part when the function ends with an infinite segment. The transformation into Minerve has to chose an arbitrary period value (we chose 1).

The third problem was that RTaW-Pegase accepts two expressions of the periodicity, whereas Minerve admits only one. In RTaW-Pegase, one may either state that a function is pseudo-periodic when it exists T, d, c such that: $\forall t > T, f(t + d) = f(t) + c$ or: $\forall t \geq T, f(t + d) = f(t) + c$, whereas Minerve only knows about the latter definition. Both conditions are equally expressive, but one has to increase the value of T in order to cast a definition from the former format into the latter one.

Consider, for instance, the function

$$g : t \mapsto \max\left(t + 1, \left\lfloor \frac{t}{2} \right\rfloor\right)$$

plotted in Figure 4. In RTaW-Pegase, it can be represented by a the closed first segment $[(0, 2) - 1/2(2, 0)]$ followed by the left-open segment $](2, 1)0(4, 0)]$ repeated with periodicity 2 and increment 1, as shown in the upper part of Figure 4. In Minerve, such a representation is impossible, and one has to extend the non-periodic prefix, as illustrated in the lower part of Figure 4.

5.2. Static Single assignment. Looking at the trace presented in Figure 2, the correctness of the trace relies on a correct computation of the variable `cumA_Node`. But since this variable is assigned three times in the trace, there is no single value to check. We may consider the line number of the expression as a tie-breaker, but it is somehow fragile. Then, RTaW-Pegase can generate an enhanced trace, augmented with comments giving an number identifier to each expression to check. This enhanced trace also contains as assertions the expected values of the operands and results for each operation. For instance, the line 20 in the listing of Figure 2 is replaced by the set of lines presented in Figure 5.2.

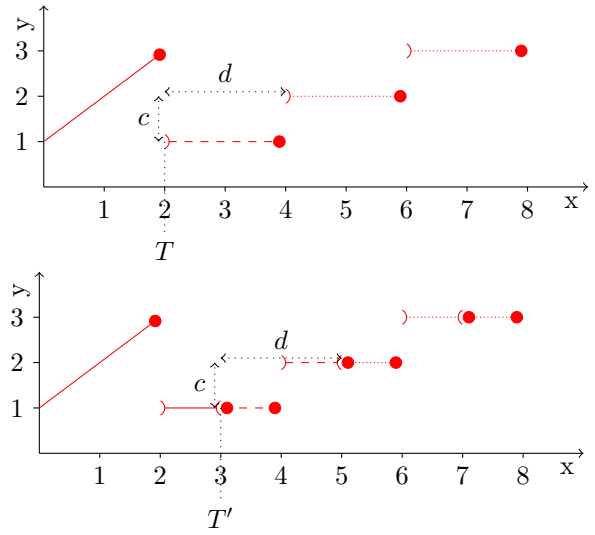

FIGURE 4. Prefix extention due to translation from RTaW-Pegase to NCCoq

```
# MP-Coq-Check 1
assert( cumA_Node1 = uaf([(0,0)0(+Infinity,0)
    []))
assert( Flow1 = upp([(0,0)], period(](0,1360)
    0(10000,1360)]), 1360, 10000))
cumA_Node1 := cumA_Node1 + Flow1
assert( cumA_Node1 = upp([(0,0)], period
    (](0,1360)0(10000,1360)]), 1360, 10000))
# Coq Check 1: cumA_Node1 = (cumA_Node1 +
    Flow1)
```

FIGURE 5. Example of a trace that can be produced by RTaW-Pegase

Then a Coq file is generated with a list of definitions and proof obligations. For instance, from the part of the trace presented in Figure the Coq code presented in Figure 6 is generated. The `idtac` and `Time` commands are there to log the time taken by Coq to perform the proof.

5.3. Performance tricks. Going from simple hand-made examples, with dozens of operations, to realistic examples with thousands of operations, we encountered some performance issues, not related to the core of the algorithms but related to some details in implementations. We sum them up here.

In a first version of the files we used `Let` instead of `Definition`. While this was apparently inocuous with only a few occurrences, this became a major issue with hundreds or thousands of occurences as this means the terms computed by Coq for the last check was embedding all previous definitions, dramatically slowing down computations. Replacing the keyword `Let` by `Definition` was enough to fix the issue.

```
(* Proof for Check 1*)
(* Conversion of upp([(0,0)0(1,0)[, period
    ([(1,0)0(2,0)[), 0, 1) *)
Definition lop_1_js : @seqjs bigQ := [::
  (0, ( (0)%:E, (0, (0)%:E)));
  (1, ( (0)%:E, (0, (0)%:E)))]
%bigQ.
Definition lop_1 := F_of_sequpp {|
  sequpp_T := 1;
  sequpp_d := 1;
  sequpp_c := 0;
  sequpp_js := lop_1_js
|}%bigQ.
(* Conversion of upp([(0,0)0(0,0)], period
    (](0,1360)0(10000,1360)]), 1360, 10000) *)
Definition rop_1_js : @seqjs bigQ := [::
  (0, ( (0)%:E, (0, (1360)%:E)));
  (5000, ( (1360)%:E, (0, (1360)%:E)));
  (10000, ( (1360)%:E, (0, (2720)%:E)))]
%bigQ.
Definition rop_1 := F_of_sequpp {|
  sequpp_T := 5000;
  sequpp_d := 10000;
  sequpp_c := 1360;
  sequpp_js := rop_1_js
|}%bigQ.
(* Conversion of upp([(0,0)0(0,0)], period
    (](0,1360)0(10000,1360)]), 1360, 10000) *)
Definition res_1_js : @seqjs bigQ := [::
  (0, ( (0)%:E, (0, (1360)%:E)));
  (5000, ( (1360)%:E, (0, (1360)%:E)));
  (10000, ( (1360)%:E, (0, (2720)%:E)))]
%bigQ.
Definition res_1 := F_of_sequpp {|
  sequpp_T := 5000;
  sequpp_d := 10000;
  sequpp_c := 1360;
  sequpp_js := res_1_js
|}%bigQ.
Goal res_1 = (lop_1 + rop_1).
Proof. idtac "Check 1". Time nccoq. Qed.
```

FIGURE 6. Coq proof generated from listing in Figure 5.2

As a first step, our automatic tactic `nccoq` maps each min-plus operation, like $+$ or $*$ introduced in Section 2, on arbitrary functions to effective operators on UPP functions, our tactic uses the typeclass resolution mechanism of Coq. This mechanism can perform exponential searches and is known to easily lead to serious slow downs. After doing some profiling, we discovered that more time was spent in this resolution than actually performing the computation of the reflexive tactic. Putting the definitions of the sequences `sequpp_js` in a separate definition

(they were originally inlined) solved the issue by enabling the type class search procedure of Coq to find the expected solution much earlier.

Finally, the largest computations first appeared much slower than expected. Inspecting the intermediate values computed, we discovered rational numbers such as $\frac{17498164897827\times10^{5000}}{38753975857\times10^{5000}}$. Further investigation revealed that we were using non normalizing operations on arbitrary precision rational numbers. Replacing them with the normalizing operations solved the issue.

In Minerve, all functions are periodic with a given period (c.f. d in Figure 4). For affine functions, this means that an arbitrary period must be chosen. A bad choice can be an issue when computing an operator whose operands have wildly diferent periods. To avoid such issues, a preprocessing was added to change the (fake) period of affine functions in accordance to the period of the other operand before any binary operation.

When performing a proof with Coq, the proof is first elaborated with tactics (everything between `Proof` and `Qed`) then rechecked by the kernel of Coq at `Qed` time. This means any expensive computation performed by the tactics will be performed a second time by the `Qed`. To avoid such duplications, a trick using the `abstract` tactic of Coq is used[1].

6. Benchmarks

We evaluated our approach on three midsize to large case studies representative of actual industrial use cases.

The first case study, is a medium size network made of 8 end systems and 2 switches. It is crossed by 57 flows, each having 1 to 5 receivers. The links data rate is set to 100Mb/s except for the link between the two switches which is at 1000Mb/s. For the service policy used, all the flows are distributed between 5 priority levels, at the same level of priority flows respect the FIFO rule. The analysis of such a network with RTaW-Pegase as well as the writing of the trace takes about 1 second.

In the second and third case studies, the considered network is much larger. Comprising 104 end systems and 8 switches, it is crossed by a thousand flows. The links data rate is set to 100Mb/s. For the second case study, all the flows have the same level of priority and are processed in a single FIFO queue. For the third case study, the flows are now distributed in different priority levels and are processed by 5 FIFO queues. RTaW-Pegase analyzes these configurations in about 4 and 8 seconds.

Checking each of these benchmarks involved verifying thousands of operations in each case. We ran the benchmarks on an average few years old laptop with 4 GiB of RAM. The total run time where kept within a couple of hours thanks to checking time per operation well below the second in most cases, with only a handful of operations requiring a few dozen of seconds to be proved correct by Coq. All timings are summarized in Table 1. Note that the last benchmark had to be divided in eight separate Coq files to avoid Coq running out of memory on a huge file.

Although much slower than the initial runs of RTaW-Pegase, we consider these runtimes for verification to be perfectly acceptable considering that they would constitute the last part of the development or certification process of an embedded network. The fat RTaW-Pegase can still be used without extra verification for dimensioning or development purposes.

Last but not least, no bug was found in RTaW-Pegase during the experiments.

The benchmarks, as well as detailed instructions to reproduce those results, are available at https://doi.org/10.5281/zenodo.5849594.

7. Conclusion

The skeptical approach (that formally proves the correctness of a result given by a program), is an efficient way to get a high level of confidence in the results of a program. It has been applied in the context of the real-time performances of embedded networks: the correctness of the min-plus operations involved in the computation of real-time bounds can now be checked using the Coq proof assistant. The theoretical part have been presented in [17]. This paper presents how it can be used in an industrial context.

This research experiment can be seen as a success both in terms of development effort and scalability of the verification. Developing a min-plus toolbox requires about one man-year of development [5, 9], developing our min-plus checker required one PhD year of development [17] and plugging the min-plus checker and RTaW-Pegase required about 1 month of development. On run-time, analyzing a network with the RTaW-Pegase requires typically a few minutes using a common laptop whereas checking the results requires a few hours on the same hardware. For software with such a confidence level, the overhead in development time and running time is fully acceptable.

As a future work, one could consider unifying the previous works in [16], [14] and the current work in

[1] This tactic basically seals a computation into an auxiliary lemma (with its own `Qed`), then the final `Qed` just checks the auxiliary lemma statement rather than completely rechecking it.

benchmark	#op	time	time / op	max time
MEDIUM_SP	1923	4:44	0.15	15.23
BIG_FIFO	34121	47:35	0.08	7.61
BIG_SP	81333	4:19:38	0.19	20.28

TABLE 1. Benchmarks: "#op" is the number of operations, "Time" the total time for Coq to check all operations (hours:minutes:seconds), "time / op" the average time per operations (in seconds), "max time" the maximum time to check a single operation (in seconds).

a single Coq proof. This would greatly reduce the trusted code base and avoid having a Network Calculus expert check the output trace of RTaW-Pegase and its mapping to the verified Coq file.

References

[1] RealTime-at-Work online Min-Plus interpreter for Network Calculus. https://www.realtimeatwork.com/minplus-playground.

[2] Steffen Bondorf and Jens B. Schmitt. The DiscoDNC v2 – A Comprehensive Tool for Deterministic Network Calculus. In *Proc. of the International Conference on Performance Evaluation Methodologies and Tools*, ValueTools '14, pages 44–49, December 2014.

[3] Anne Bouillard, Marc Boyer, and Euriell Le Corronc. *Deterministic Network Calculus – From theory to practical implementation*. Number ISBN: 978-1-119-56341-9. Wiley, 2018.

[4] Anne Bouillard, Bertrand Cottenceau, Bruno Gaujal, Laurent Hardouin, Sebastien Lagrange, Mehdi Lhommeau, and Eric Thierry. COINC library: a toolbox for the network calculus. In *Proc. of the 4th int. conf. on performance evaluation methodologies and tools (ValueTools)*, volume 9, 2009.

[5] Anne Bouillard and Éric Thierry. An algorithmic toolbox for network calculus. *Discrete Event Dynamic Systems*, 18(1):3–49, october 2008. http://www.springerlink.com/content/876x51r6647r8g68/.

[6] Anne Bouillard and Eric Thierry. An Algorithmic Toolbox for Network Calculus. *Discrete Event Dynamic Systems: Theory and Applications*, 18, 03 2008.

[7] Marc Boyer, Jörn Migge, and Marc Fumey. PEGASE, a robust and efficient tool for worst case network traversal time. In *Proc. of the SAE 2011 AeroTech Congress & Exhibition*, Toulouse, France, 2011. SAE International.

[8] Marc Boyer, Nicolas Navet, and Marc Fumey. Experimental assessment of timing verification techniques for AFDX. In *Proc. of the 6th Int. Congress on Embedded Real Time Software and Systems*, Toulouse, France, February 2012.

[9] Marc Boyer, Nicolas Navet, Xavier Olive, and Eric Thierry. The PEGASE project: precise and scalable temporal analysis for aerospace communication systems with network calculus. In T. Margaria and B. Steffen, editors, *Proceedings of the 4th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2010)*, LNCS. Springer, 2010.

[10] Felipe Cerqueira, Felix Stutz, and Björn B Brandenburg. PROSA: A case for readable mechanized schedulability analysis. In *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 273–284. IEEE, 2016.

[11] Robert Davis, Alan Burns, Reinder Bril, and Johan Lukkien. Controller area network (CAN) schedulability analysis: Refuted, revisited and revised. *Real-Time Systems*, 35:239–272, 2007. 10.1007/s11241-007-9012-7.

[12] Pascal Fradet, Xiaojie Guo, Jean-François Monin, and Sophie Quinton. Certican: A tool for the coq certification of CAN analysis results. In Björn B. Brandenburg, editor, *25th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2019, Montreal, QC, Canada, April 16-18, 2019*, pages 182–191. IEEE, 2019.

[13] Pierre Letouzey. Extraction in coq: An overview. In Arnold Beckmann, Costas Dimitracopoulos, and Benedikt Löwe, editors, *Logic and Theory of Algorithms, 4th Conference on Computability in Europe, CiE 2008, Athens, Greece, June 15-20, 2008, Proceedings*, volume 5028 of *Lecture Notes in Computer Science*, pages 359–369. Springer, 2008.

[14] Etienne Mabille, Marc Boyer, Loïc Fejoz, and Stephan Merz. Towards certifying network calculus. In *Proc. of the 4th Conference on Interactive Theorem Proving (ITP 2013)*, Rennes, France, July 2013.

[15] Lisa Maile, Kai-Steffen Hielscher, and Reinhard German. Network calculus results for tsn: An introduction. In *Proc. of the Information Communication Technologies Conference (ICTC 2020)*, pages 131–140. IEEE, 2020.

[16] Lucien Rakotomalala, Marc Boyer, and Pierre Roux. Formal Verification of Real-time Networks. In *JRWRTC 2019, Junior Workshop RTNS 2019*, Toulouse, France, November 2019.

[17] Lucien Rakotomalala, Marc Boyer, and Pierre Roux. Verifying min-plus computations with coq. In *Proc. of the 13th NASA Formal Methods Symposium (NFM 2021)*, May 24-28 2021.

[18] Hanrijanto Sariowan, Rene L. Cruz, and George C. Polyzos. SCED: A generalized scheduling policy for guaranteeing quality-of-service. *IEEE/ACM transactions on networking*, 7(5):669–684, October 1999.

[19] E. Wandeler. Modular performance analysis and interface based design for embedded real time systems. 2006.

[20] Ernesto Wandeler and Lothar Thiele. Real-Time Calculus (RTC) Toolbox, 2006.

[21] Boyang Zhou, Isaac Howenstine, Siraphob Limprapaipong, and Liang Cheng. A survey on network calculus tools for network infrastructure in real-time systems. *IEEE Access*, 8:223588–223605, 2020.