Proof-by-Instance for Embedded Network Design

From Prototype to Tool Roadmap

Marc Boyer*, Loïc Fejoz†, Stephan Merz‡

*ONERA – The French Aerospace Lab, 31055 Toulouse, France
†RealTime-at-Work, Nancy, France
‡Inria & LORIA, Nancy, France

Abstract

Proof-by-instance is a technique that ensures the correctness of the results of a computation rather than proving correct the tool carrying out the computation. We report here on the application of this idea to check the computations relevant for analyzing time bounds for AFDX networks. We have demonstrated the feasibility of the approach by applying a proof-of-concept implementation to an AFDX network of realistic size, and we outline the roadmap towards a mature industrial toolset. This approach should lead to a reduction of the time and cost of developing analysis tools used in the design of embedded networks where certification is mandatory.

1. Introduction

Embedded systems are often safety-critical, and their development therefore calls for techniques ensuring high quality. In particular, domains such as avionics (fly-by-wire, control) and medicine (infusion pumps, pacemakers) are subject to regulation requiring certification, and similar quality levels are justified for domains such as transportation (automotive, train control) or for systems where loss or malfunctioning has high costs, e.g. satellites. Traditionally, quality is ensured by applying rigorous development processes, testing of components and systems, and the use of formal methods. The verification tools used in such an approach are themselves critical and therefore subject to certification. In particular, some regulatory standards (e.g. DO178-C) require the same level of confidence in verification tools as in the systems in whose development they are used, adding to the overall cost of development and complicating the maintenance and improvement of such tools.

For instance, the PEGASE project [4] focused on determining bounds on message delays and dimensioning buffers in the design of networks for embedded systems such as AFDX backbones used in avionics. The formal underpinnings of the project were provided by Network Calculus (NC) [6], a domain-specific formal method, and as explained above, its use for certifying the design requires high confidence in the results computed by the tools. Although the underlying theory is generally well understood, implementation errors may result in faulty network designs, with unpredictable consequences.

Although the use of formal methods is encouraged in certified developments, formally verifying a complex tool such as an NC analyzer, for any possible network design, looks like a daunting task. Proof-by-instance (also known as result certification) addresses the simpler problem of checking that the result computed for a specific input is correct. In our application, the NC analyzer only needs to record the computations that led to the result that it produced, and their correctness can be established by checking the applicability of the computation rules and the accuracy of the numerical computations themselves.

We studied the feasibility of developing a proof-by-instance method for NC. In order to have high assurances about correctness, checking is performed within the kernel of the Isabelle proof assistant, based on a library for NC that we developed in Isabelle/HOL [8]. As a side effect, this also increases our confidence in the version of NC underlying the PEGASE verifier.

We have produced a proof-of-concept implementation of this technique for the timing verification tool from the PEGASE project [4], and have successfully applied this prototype to an industrial-size case study. Based on this experience we lay out a road-map of how a full-fledged result verifier could be developed that could play a valuable role in certifying network designs for embedded systems.

2. Product-based assurance for verification tools

Whereas formal methods promise high assurances for correctness, the cost of full-fledged proof remains prohibitively high in many cases, and the ability to scale up formal verification for usage in industrial applications has been questioned. Nevertheless, advances in formal methods research have resulted in techniques that have proved practical and effective for specific problems.

Our contribution is based on a technique that ensures the correctness of the results computed by the tool, not the tool itself. This approach is called "proof by instance" or "certificate-based result checking". In a nutshell, the tool outputs not only the result, but also a certificate for why the result is correct, for example a trace of the computation. The validity of the certificate can be established offline by a trusted checker. For tools used at design time, we argue that this approach has several advantages over proving the calculator correct.

- Instrumenting the calculator for generating a trace is much easier, and hence less expensive, than attempting a full-fledged correctness proof.
- Certificates should be designed in such a way that checking their correctness is essentially trivial. This is usually the case for checking the validity of the trace of a computation. Certifying the checker is therefore much easier than certifying the original tool.
- The calculator producing the certificate is treated as a black box: it can be implemented using any software development process, programming language, and hardware by a tool provider separate from the checker. It can be updated without having to be re-qualified, as long as it still produces certifiable computation traces.
- Proof by instance is a good match for industrial processes based on testing.

Checking the trace of an NC analyzer is similar to checking a mathematical proof: its computation consists in applying theorems of NC whose hypotheses need to be established, as well as the accuracy of calculation. We implemented our checker within the proof assistant Isabelle/HOL [8]. A companion paper [7] describes this work from the perspective of interactive theorem proving. The correctness of our checker therefore relies on the correctness of the proof assistant, which in turn is reduced to the correctness of a relatively small kernel (a few thousand lines of ML code) that is mature and widely used.

3. Proof-by-instance applied to timing verification

3.1 Chain of trust

Figure 1 sketches the overall approach. The designer provides a network topology for which he wants to establish some timing bounds. These are computed by the NC tool, which also provides a trace of its computation. In the case of PEGASE, a trace contains calculations over certain classes of functions, as explained in section 3.2, as well as reasoning steps similar to a mathematical proof, corresponding to the network topology as explained in section 3.3.

The checker, separate from the NC analyzer, then verifies that all reasoning steps are justified and that all calculations are carried out correctly. The confidence in the correctness of the result therefore depends only on the trustworthiness of the checker, not on the implementation of the NC tool.

In order to implement a checker in the proof assistant Isabelle/HOL, one must write an Isabelle "theory" that contains a library of theorems of NC applied by the analyzer. This work is done once and for all. As a side benefit, formal proofs of these theorems give us high confidence in the correctness of the theory that underlies the analyzer. Result checking amounts to verifying that only valid instances of theorems appear in the trace. The communication between the analyzer and the checker is explained in more detail in Section 3.3.

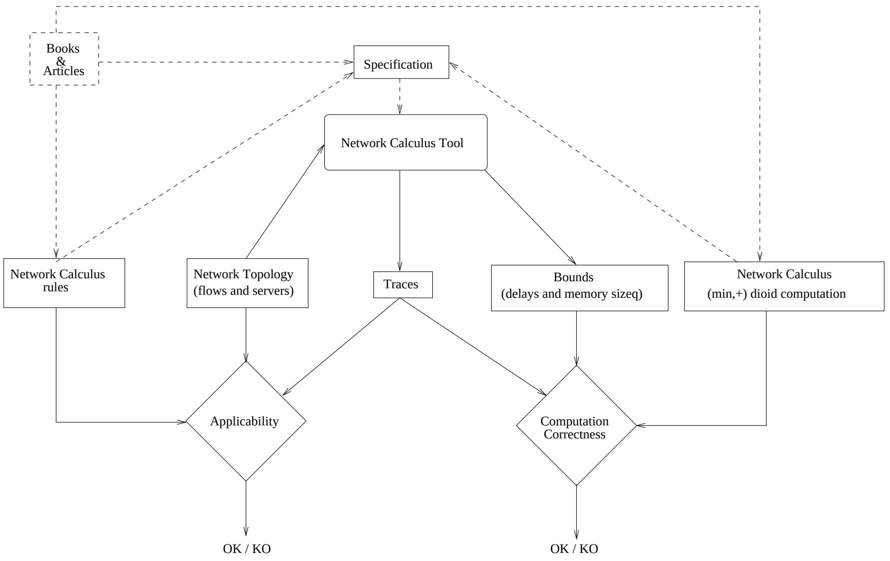

Figure 1: Proof by instance process

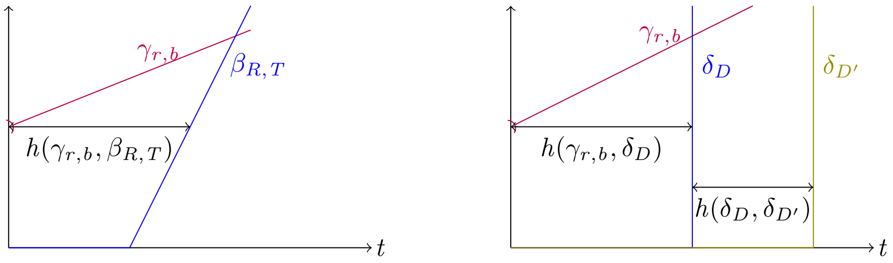

Figure 2: Common curves and delay.

3.2 Illustration of Network Calculus computation

Network calculus [6] is a theory for computing upper bounds in networks. Network elements that introduce delays are modeled by *servers*, and resources, e.g. messages, that can be delayed, by *flows*. A flow is represented by its cumulative function f, where $f(t)$ is the total number of bits sent by this flow up to time t. Hence the mathematical background of NC is a theory of the set of functions

$$\mathcal{F} = \{ f : \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0} \cup \{\infty\} \mid x \leq y \Rightarrow f(x) \leq f(y) \} \tag{1}$$

that form a dioid under the operations $\sqcap$ and $+$ defined as point-wise minimum and addition.

The network calculus theory also uses two operators, the min-plus convolution ($\oplus$) and deconvolution ($\oslash$), and the associated Kleene closure, aka sub-additive closure.

$$(f \oplus g)(t) = \inf_{0 \leq u \leq t} (f(t-u) + g(u)) \tag{2}$$

$$(f \oslash g)(t) = \sup_{0 \leq u} (f(t+u) - g(u)) \tag{3}$$

$$f^{\oplus} = \delta_0 \sqcap f \sqcap (f \oplus f) \sqcap (f \oplus f \oplus f) \sqcap \cdots \tag{4}$$

For practical applications, four families of functions are particularly interesting, which are defined as $\delta_d(t) = 0$ if $t \leq d$, ∞ otherwise, $\beta_{R,T}(t) = 0$ if $t \leq T$ and $R \cdot (t - T)$ otherwise, and $\gamma_{r,b}(t) = 0$ if $t \leq 0$ and $r \cdot t + b$ otherwise. These functions are illustrated in Figure 2. The horizontal deviation between two functions, $h(f, g)$ is also an operator of interest: no formal definition is given here, but an illustration is given in Figure 2.

The network calculus theory models the flows and servers with arrival and service curves. A flow A is said to have an arrival curve α (denoted $A \preceq \alpha$) iff the amount of data sent by

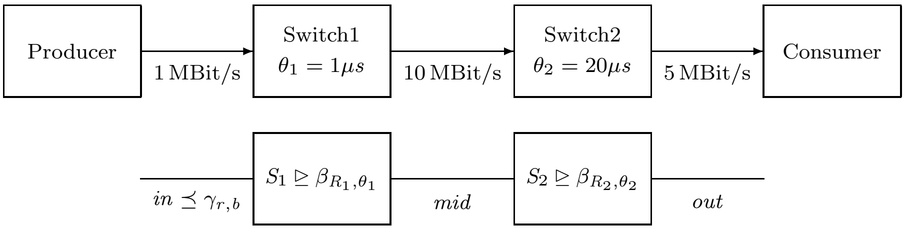
Producer
1 MBit/s
Switch1
$\theta_1 = 1\mu s$
10 MBit/s
Switch2
$\theta_2 = 20\mu s$
5 MBit/s
Consumer
$in \preceq \gamma_{r,b}$
$S_1 \trianglerighteq \beta_{R_1,\theta_1}$
mid
$S_2 \trianglerighteq \beta_{R_2,\theta_2}$
out

Figure 3: A simple system and its Network Calculus representation.

the flow throughout any interval of width Δ is less or equal than $\alpha(\Delta)$, formally defined as $\forall t, \Delta \geq 0 : A(t + \Delta) - A(t) \leq \alpha(\Delta)$. A server offers a service β (denoted $S \trianglerighteq \beta$) iff for each arrival A, the departure curve D ensures $D \geq A \oplus \beta$. For example, a function $\gamma_{r,b}$ models a token bucket flow, with a possible burst of size b and a long term rate r, a function δ_d models a server of bounded delay d, and a function $\beta_{R,T}$ models a service with an initial latency of T and a guaranteed throughput R.

NC theory contains fundamental results relating these concepts. The basic one states that if a flow with arrival curve α crosses a server with service curve β, then the crossing delay is bounded by $h(\alpha, \beta)$ and the output flow has arrival curve $\alpha \oslash \beta$.

$$\frac{A \preceq \alpha \qquad S \trianglerighteq \beta \qquad A \xrightarrow{S} D}{d(A, S) \leq h(\alpha, \beta) \qquad D \preceq \alpha \oslash \beta} \tag{5}$$

In order to illustrate the use of Network Calculus theories on a simple example, let us consider the producer-consumer setup shown in Fig. 3. The producer sends at most one frame every $T = 20\,\text{ms}$. We assume that the payload is of maximal size 980 bytes. Assuming an overhead of 20 bytes per frame, the maximum frame size is $MFS = 8000\,\text{bits}$. This flow is sent to a consumer via two switches with switching delays $\theta_1 = 1\,\mu s$ and $\theta_2 = 20\,\mu s$. The physical links between the producer, the switches, and the consumer have bandwidths of 1, 10 and 5 MBit/s, respectively.

The NC model appears in the lower part of Fig. 3. Flow *in* is constrained by the arrival curve $\alpha_{in} = \gamma_{r,b}$ where b equals MFS and $r = \frac{MFS}{T} = \frac{8000}{20 \times 10^3} = \frac{2}{5}$. Applying NC rules, one may determine arrival curves for the streams *mid* and *out*, and compute delays incurred by the frames at the two servers. For the first server, the maximum delay is $801\,\mu$s, whereas the delay at the second server is bounded by $h(\alpha_{mid}, \beta_{5,20}) = \frac{42102}{25}\,\mu$s. Consequently, the overall delay incurred by frames equals

$$801\,\mu\text{s} + \frac{42102}{25}\,\mu\text{s} = \frac{62127}{25}\,\mu\text{s}.$$

For the result to be trusted, one must be able to certify the computations of the individual results, and also ensure that composing intermediate results in sequence is correct w.r.t. the network topology, based on appropriate Isabelle theorems.

Checking the above computation is quite simple since it uses only rule (5), but in practice, analyzing a real network requires the use of dozens of rules, and the accuracy of the result depends on the choices of the appropriate theorems that were applied.

3.3 Tool chain

As explained in Section 3.1, the analyzer tool is completely separate from the result verifier. The analyzer is considered as a black box: it implements the NC algorithms and produces a trace (or certificate) that can be checked by the verification tool. Currently, the analyzer is a Java prototype developed from scratch, but our intention is to merge it into the RTaW-PEGASE AFDX analyser.

The verification tool is based on the Isabelle/HOL proof assistant augmented with dedicated NC libraries. Isabelle is a generic logical framework that provides syntax for defining logics and a trusted kernel for applying proof rules. Isabelle/HOL refers to the encoding of higher-order logic in this framework; it is the best-developed object logic within Isabelle and provides many convenience features for modeling and proving, including a language close to that of standard functional programming languages and interfaces to efficient automatic theorem provers. Nevertheless, ultimately all theorems proved in Isabelle/HOL are certified by the generic kernel.

We have developed Isabelle/HOL theories for encoding the basic objects of NC (including flows and servers) and their properties. In particular, these theorems contain machine-checked theorems of network calculus.

For a given network configuration, the NC analyzer computes upper bound on network delays ($\frac{62127}{25}\,\mu$s in the example from Section 3.2), and produces a trace that gives information about how the result was obtained. In particular, the trace contains references to theorems used to obtain intermediate results (e.g., the value of α_{mid} in the example from section 3.2). The checker inputs the trace and verifies that this reasoning is correct.

4. Road-map

4.1 Current prototype

In our current proof-of-concept implementation, the analyzer generates traces in the form of Isabelle proof scripts that are processed by the proof assistant in the context of our NC theories.

Using this prototype, we have been able to compute upper bounds for traversal times for a network drawn from an industrial case study (involving 8 switches and more than 5000 flows), and to provide a correctness proof for the computation. We compare this implementation to state-of-the-art tools, along three axes.

Accuracy of bounds. The current prototype implements only simple results of network calculus, for restricted classes of functions. The computed bounds are pessimistic compared to up-to-date implementations of network calculus [2]. In our case study, the bounds are overestimated by a factor of about 2.

Computation time. Computing the bounds on delay and generating the traces takes a few minutes. Checking the generated traces requires about 8 hours on a standard laptop. These first results demonstrate the scalability of the approach, and the ability to handle realistic problems. Note that checking time is not really critical, since the checker needs only be run on the final design when the architecture is believed to be stable, whereas intermediate results only require running the analyzer without checking.

Development effort. Once the feasibility of the approach demonstrated, the very good news is that the development effort of such a formal tool was quite reasonable. Although we have not precisely tracked the effort that went into producing this prototype, we claim that the cost of the formal part (developing Isabelle theories and generating traces) represents less than 3/4 of the global effort. In other words, the overhead for developing a trustworthy version of a Network Calculus analyzer does not exceed a factor of 2 or 3, while using state-of-the-art techniques (documentations, testing, peer-review, etc.) often require a factor of 5 to 10.

From this experiments, we can draw a road-map for transforming the prototype to an efficient tool.

4.2 An Isabelle theory for network calculus

So far, only basic objects and results of network calculus have been formalized in the NC Isabelle library, and some well-known theorems have not been formally proved. The remaining work can be separated into two parts:

1. The first step is to write more theorem statements, in order to be able to have more accurate models of the networks, and more accurate bounds. For example, it is known from [5] that modelling the shaping in the network can improve the bounds by a factor of 40%.

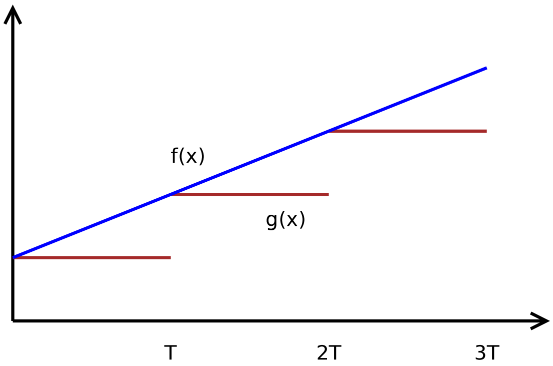

Figure 4: Affine and stair-case approximations of periodic flow

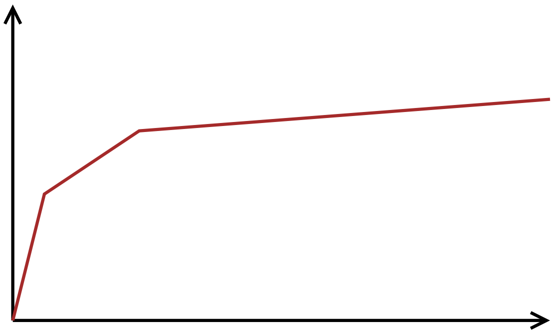

Figure 5: A concave piecewise linear function

2. The second step is to complete the proofs of all theorems. Up to now, the Network Calculus results have been considered as correct, since rigorous proofs are given in the literature, and only some simple proofs have been re-written in Isabelle.

Notice that even if NC theorems are left unproved, this framework still ensures that each trace conforms to the tool specification, since the NC theory is the specification of the tool. Providing proofs for NC theorems will increase our confidence in the specification of the tool and in the fact that the results are meaningful for the actual system. In the absence of formal proofs, the coherence of the specification must be ensured by careful rereading by domain specialists.

4.3 Checking $(\sqcap, +)$ operations on functions within Isabelle

NC theory uses operations on functions, including simple sum and minimum, but also more complicated ones such as convolution or deconvolution. The network calculus theory relies on such operations, based on their mathematical properties, but does not address their implementation. A correct implementation of these operations is important for the overall result to be correct. Our current prototype uses very simple affine functions, and it is quite easy, for example, considering two functions $f(x) = 3x + 2$, $g(x) = 4x + 10$ to prove that $(f + g)(x) = 7x + 12$, $(f \sqcap g)(x) = 3x + 2$.

But to provide more accurate timing result, more general classes of functions must be supported. For example, a periodic flow sending a message of size s every T time unit can be represented by a linear function $f(x) = \frac{x}{T} + s$ or a stair-case function $g(x) = s\lceil\frac{x}{T}\rceil$ (see Figure 4). The function g is more accurate, and can lead to better bounds. But while computing the sum of two flows of sizes s and s', and period T and T' is straightforward with linear functions, it requires a more general function representation with stair-case functions. The class of piecewise linear concave and convex functions (see Figure 5) can represent local bursts, shaping, and long term rate, while having a simple implementation. A very generic class, the ultimately pseudo-periodic functions, has been defined in [1]. It can represent any functions made of a set of piecewise linear segments, with a finite prefix and a periodic suffix. All algorithms implementing the basic operators (sum, minimum, convolution, deconvolution, etc.) over this class of functions are described in research papers but some corner cases are left out, and the implementation of complex algorithms is error-prone.

Each tool implements one or several classes of functions, and, as expected, the more generic functions allow for more accurate models, and give better (*i.e.* smaller) bounds.

One way will be to identify some simpler classes of functions, like the one of [3], combining concave piecewise linear functions and stair-cases functions, providing good accuracy in timing analysis but allowing for full and reasonable straightforward Isabelle proofs.

Another way could be to handle partial computations, over only a finite horizon. Instead of computing a function $f : \mathbb{R}^+ \to \mathbb{R}$ for all possible arguments, one may consider a function f_T such that $f(x) = f_T(x)$ only for $x \leq T$. In some cases, such approximations are sufficient for computing bounds (see Figure 2).

4.4 Intermediate proof format

The current prototype outputs traces directly in Isabelle's input language. In the future, we plan to define a dedicated proof format, independent of Isabelle, for the following reasons:

1. A dedicated format will allow us to represent traces at the appropriate level of abstraction, whereas current proofs mix main results of network calculus such as (5) and simple reductions of (in)equations such as $a + b \leq c, b \geq 0 \implies a \leq c$. A dedicated format can be independent of any particular checker, and will be able to interface different NC tools and checkers.

2. It can be more compact (current proofs are quite long-winded) and thus enable more efficient checking, in space (smaller proofs) and time (avoiding to redo the same proof with different values, by creation of local lemmas).

3. Higher-level proof formats could be used to generate human-readable proof, for example as justifications for certification authorities.

5. Conclusion

Network Calculus has proven to be a powerful formalism that is well suited to provide guarantees on the worst-case performances of networks for large critical embedded systems, in particular in the avionics domain. The work described here provides insight on how a proof-by-instance technique can contribute to ensure the correctness of the results, thus helping turn a proof-of-concept verification software into an industrial-grade design tool. It also exhibits by-products of the approach such as a more coherent and formally checked theory and a more precise specification of tools.

Based on the successful development of a prototype, it gives a road-map for the development of a full efficient and trustworthy tool.

References

[1] A. Bouillard and E. Thierry. An algorithmic toolbox for network calculus. *Discrete Event Dynamic Systems*, 18(1):3–49, 2008.

[2] M. Boyer, J. Migge, and M. 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.

[3] M. Boyer, J. Migge, and N. Navet. An efficient and simple class of functions to model arrival curve of packetised flows. In *Proceedings of the 1st International Workshop on Worst-Case Traversal Time (WCTT'2011)*, pages 43–50, New York, NY, USA, novembre 2011. ACM.

[4] M. Boyer, N. Navet, X. Olive, and E. Thierry. The PEGASE project: Precise and scalable temporal analysis for aerospace communication systems with network calculus. In T. Margaria and B. Steffen, editors, *4th Intl. Symp. Leveraging Applications (ISoLA 2010)*, volume 6415 of *LNCS*, pages 122–136, Heraklion, Greece, 2010. Springer. http://www.realtimeatwork.com/software/rtaw-pegase/.

[5] J. Grieu. *Analyse et évaluation de techniques de commutation Ethernet pour l'interconnexion des systèmes avioniques.* PhD thesis, Institut National Polytechnique de Toulouse (INPT), Toulouse, Juin 2004.

[6] J.-Y. Le Boudec and P. Thiran. *Network Calculus.* Springer, 2001.

[7] E. Mabille, M. Boyer, L. Fejoz, and S. Merz. Towards certifying network calculus. In S. Blazy, C. Paulin-Mohring, and D. Pichardie, editors, *Interactive Theorem Proving (ITP 2013)*, volume 7998 of *LNCS*, pages 484–489, Rennes, France, 2013. Springer.

[8] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL. A Proof Assistant for Higher-Order Logic.* Number 2283 in Lecture Notes in Computer Science. Springer Verlag, 2002.