Certifying Network Calculus in a Proof Assistant

Etienne Mabille*, Marc Boyer†, Loïc Fejoz*, Stephan Merz‡

*RealTime at Work, Nancy, France
†ONERA, Toulouse, France
‡Inria & LORIA, Nancy, France

1 Result Certification for Network Calculus

In critical embedded systems such as those used in avionics, ensuring the quality of the software (using tests, qualification, and certification) represents a significant fraction of the overall cost of development. Several complementary methods exist for ensuring the quality of systems, including rigorous development processes, testing of components and systems, and the use of formal methods. The latter have long been considered unable to scale up to industrial applications, and the cost of full-fledged proof remains prohibitively high in most cases. Nevertheless, advances in formal methods research have resulted in techniques that have proved practical and effective for specific problems. The Network Calculus [5] and supporting tool sets have found widespread use for determining bounds on message delays and for dimensioning buffers in embedded networks, such as in the design and certification of the Airbus A380 AFDX backbone [1, 3, 4]. However, results produced by existing tools based on the Network Calculus have to be trusted: although the underlying theory is generally well understood, implementation errors may result in faulty network designs, with unpredictable consequences.

For application domains subject to strict regulatory requirements, it appears mandatory to ensure the correctness of the results computed by the tools supporting network designs. In this contribution, we suggest a technique for certifying these results based on a "proof by instance" approach. In a nutshell (cf. Figure 1), the tool outputs a trace of the calculations it performs, as well as their results. The validity of the trace (both w.r.t. the applicability of the computation steps and the numerical correctness of the result) 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 outputting a trace is much easier (and hence less expensive) than attempting a full-fledged correctness proof, and checking the correctness of a computation is essentially trivial.
- The calculator 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 requalified, as long as it still produces certifiable computation traces.
- Proof by instance is a good match for industrial processes based on testing. Nevertheless, a design that has been checked is guaranteed to be correct for any inputs matching the hypotheses of the model.

Checking the trace of a Network Calculus tool is similar to checking a mathematical proof: applying a given rule requires establishing its hypotheses, beyond pure calculation. We suggest to implement the checker by taking advantage of the trusted kernel of a proof assistant, specifically Isabelle/HOL [6].

In the remainder, we sketch a prototypical implementation of our proposal and illustrate its use for a toy example in order to assess its applicability.

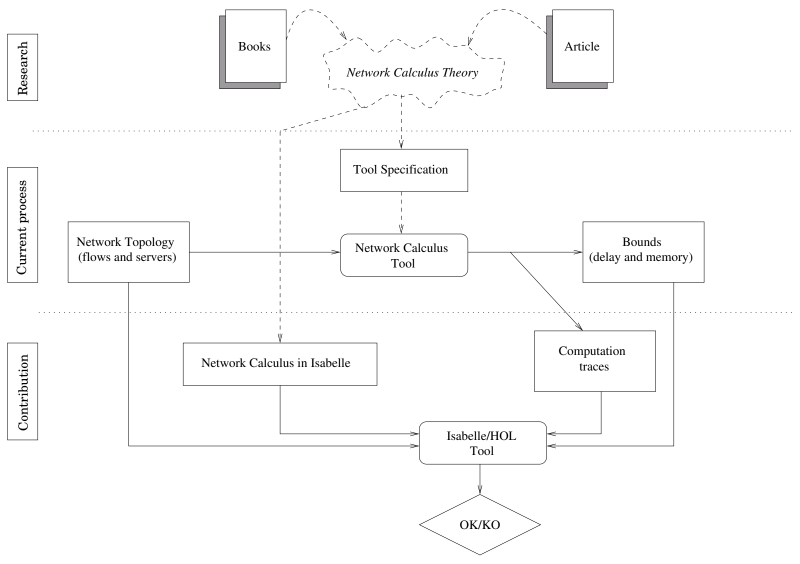

Figure 1: Proof by instance process

2 Encoding Network Calculus in Isabelle

In order to develop a result certifier within Isabelle, we need to formalize the theory underlying Network Calculus, to the extent that it underlies the algorithms whose results we intend to certify. As a side benefit of this formalization, we obtain a rigorous development of Network Calculus, including all possible corner cases that may be overlooked in pencil-and-paper proofs. Due to space constraints, we only give an outline of our formal development.

The fundamental notion in Network Calculus is that of a *flow*, modeled as a non-decreasing function $f : \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0} \cup \{\infty\}$. We define a suitable type in Isabelle as

typedef $\mathit{ndf} = \{f :: \mathit{real} \Rightarrow \mathit{ereal}\ .\ (\forall r \leq 0.\, f\ r = 0) \wedge \mathit{mono}\ f\}$

where *real* and *ereal* are pre-defined types corresponding to $\mathbb{R}$ and $\mathbb{R}\cup\{\infty\}$, and *mono* f holds if f is a weakly monotonic function.[1] Note that we extended the domain of f to arbitrary real numbers, fixing the result to be zero over negative reals, as this turned out to simplify the subsequent definitions. Over this function type, we define operations such as $+$, $*$, and $\leq$ by pointwise extension over the arguments, and establish basic algebraic properties: for example, the resulting structure forms an ordered commutative monoid with 0 and 1.

Of particular interest are special classes of functions such as linear functions

$$\beta_{r,T}(t) = \begin{cases} 0 & \text{if } t \leq T, \\ r \times (t - T) & \text{otherwise} \end{cases}$$

$$\gamma_{r,b}(t) = (rt + b)\, \mathbb{1}_{>0}(t) \quad \text{for} \quad \mathbb{1}_{>T}(t) = \begin{cases} 1 & \text{if } t > T, \\ 0 & \text{otherwise} \end{cases}$$

We also introduce characteristic operations on flows such as *convolution*, defined as

$$(f \oplus g)(t) = \inf\{f(t-s) + g(s) : 0 \leq s \leq t\}$$

and prove characteristic properties such as

lemma *convol-sub-add-stable*:
  assumes *is-sub-additive* f **and** *is-sub-additive* g
  shows *is-sub-additive* $(f \oplus g)$

[1]Application of functions and predicates to their arguments is written by juxtaposition in Isabelle/HOL.

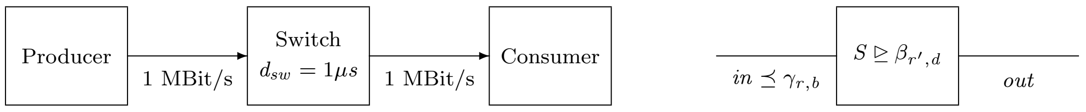

Figure 2: A producer and a consumer linked by a switch, and the Network Calculus representation.

Network Calculus represents a *simple server* as a left-total relation between (input and output) flows such that the output flow is not larger than the input flow:

typedef $\mathit{server} = \{s :: (\mathit{ndf} \times \mathit{ndf})\ \mathit{set}.\ (\forall \mathit{in}.\ \exists \mathit{out}.\ (\mathit{in}, \mathit{out}) \in s) \wedge (\forall (\mathit{in}, \mathit{out}) \in s.\ \mathit{out} \leq \mathit{in})\}$

and we define what it means for a flow to be constrained by an arrival curve α and for a server to provide minimum service β:

$$f \preceq \alpha \ \equiv\ R \leq R \oplus \alpha \qquad S \trianglerighteq \beta \ \equiv\ \forall (\mathit{in}, \mathit{out}) \in S : \mathit{in} \oplus \beta \leq \mathit{out}.$$

Again, we prove results relating these constraints to bounds on delays and backlogs. For example, the following theorem provides a bound on the delay of a simple server.

theorem *d-h-bound*:
  assumes $\mathit{in} \preceq \alpha$ **and** $S \trianglerighteq \beta$
  shows *worst-delay-server in* $S \leq$ *h-dev* $\alpha\ \beta$

where the horizontal deviation is defined as

$$\mathit{h\text{-}dev}(\alpha, \beta) = \sup_{t \geq 0}\ \inf\{d \geq 0 : \alpha(t) \leq \beta(t + d)\}.$$

Moreover, we define constructs such as composing servers in sequence or packetization. Finally, these concepts are extended to multiple-input multiple-output servers that takes vectors of flows as input and output.

3 Certifying a Simple Network Computation

In order to illustrate the use of our theories on a trivial example, let us consider the producer-consumer setup shown in Fig. 2. The producer sends at most one frame every $T = 20$ms. Let us say that the payload is of maximal size 980 bytes. Thus with an overhead of 20bytes per frames, the maximum frame size is $\mathit{MFS} = 8000$bits. The physical links have a bandwidth of 1MBit/s, and the switching delay is assumed to be $\delta_{sw} = 1\mu s$.

The Network Calculus model appears in the right-hand side of Fig. 2. Flow *in* is constrained by the arrival curve $\gamma_{r,b}$ where b equals MFS and $r = \frac{\mathit{MFS}}{T} = \frac{8000}{20 \times 10^3} = \frac{4}{5}$.

The service curve β is given by the function $\beta_{r',d}$ where the bandwidth $r' = 1\text{bit}/\mu s$, and the delay d equals the switching delay δ_{sw}.

We are interested in the maximal delay that frames may incur. Using the PEGASE Network Calculus tool [2], we obtain a delay of $673\mu s$, and this computation can be certified in Isabelle. The trace shown in the appendix consists of output of PEGASE interleaved with Isabelle lemmas whose assertions and proofs were automatically generated by instrumenting the tool.

4 Conclusion

We argue that the computations of tool sets used in the applications of formal methods such as Network Calculus should be certified in order to increase the confidence in the correctness of the designs. The work reported here attempts to evaluate the feasibility of such certification, based on a formalization of Network Calculus in the interactive proof assistant Isabelle/HOL.

Developing a Network Calculus engine that is able to handle an AFDX configuration requires about one or two years of implementation. The effort for developing a qualified version of such an engine, using state-of-the-art techniques (documentations, testing, peer-review, etc.) is higher by a factor of 5 or 10.

The proof-by-instance approach promises to reduce this effort while increasing the confidence in the results produced by the software. We have so far invested less than 1 development year for encoding the fundamental concepts of Network Calculus in Isabelle, and for instrumenting an existing tool to produce a full formal proof of the correctness of bounds for one single switch. We estimate that the overall effort for producing the proof for a realistic network should be between 2 and 3 years. In particular, it will be important to speed up the computations on real numbers inside the proof assistant.

In other words, we believe that result certification could lower the overhead for developing a trustworthy version of a Network Calculus tool to a factor of 2 or 3, while significantly improving its quality.

References

[1] AEEC. Arinc 664p7-1 aircraft data network, part 7, avionics full-duplex switched ethernet network. Technical report, Airlines Electronic Engineering Committee, september 2009.

[2] 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/`.

[3] F. Frances, C. Fraboul, and J. Grieu. Using network calculus to optimize AFDX network. In *Proceeding of the 3thd European congress on Embedded Real Time Software (ERTS'06)*, Toulouse, January 2006.

[4] 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.

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

[6] 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.

A Example Trace

```
##################################
# Used algorithm: SFA FIFO
##################################
# Time unit: microsecond
# Frame size unit: bit
##################################
delta0 := delay(0)
##################################
# Input flows
##################################
# unique_flow entering at EndSystem1>port-P1
lmax_unique_flow := 8000



##################################
# EndSystem : EndSystem1>port-P1
##################################


# Latency in transmission: 'internal delay'
id_unique_flow_EndSystem1portP1 := 0
# Packet arrival curve for unique_flow on EndSystem1->Router
unique_flow_EndSystem1portP1_P := star(uaf([(0,0)]](0,8000)2/5(+Infinity,+Infinity)[))


##################################
# Server : Router>port-P2
##################################
# Cumulated arrivals
cumA_RouterportP2 := zero
# unique_flow on EndSystem1->Router
cumA_RouterportP2 := cumA_RouterportP2 + unique_flow_EndSystem1portP1_P

assert(cumA_theRouterportP2 = uaf([(0,0)]](0,8000)2/5(+Infinity,+Infinity)[))
```

```
lemma AC_sub_add_closure: "sub_add_closure ((gamma (2/5) 8000) + (0::ndf)) = gamma (2/5) 8000"
using sub_add_closure_gamma by auto
```

```
# Common service
S_RouterportP2 := uaf([(0,0)0(1,0)]](1,0)1(+Infinity,+Infinity)[)
# Common delay
d_RouterportP2:=hDev(cumA_RouterportP2, S_RouterportP2)

assert(d_theRouterportP2 = 8001)
```

```
lemma delay_unique_server:
assumes "(R, R') ∈ Rep_server S" and "S ⊵ beta 1 1" and "R ≺ gamma (2 / 5) 8000"
shows "worst_delay_server R S ≤ ereal(8001)" proof -
  from assms d_h_bound have "worst_delay_server R S ≤ h_dev (gamma (2 / 5) 8000) (beta 1
1)" by auto
  with hor_dev_beta_gamma show "worst_delay_server R S ≤ ereal (8001)" by simp
qed
```

```
##################################
```

```
# WCTT
#################################
# Bound for unique_flow received by EndSystem2
# internal delay in sending server EndSystem1>port-P1
b_unique_flow_EndSystem2 := 0
# delay induced by Router>port-P2
b_unique_flow_EndSystem2:= b_unique_flow_EndSystem2 + d_RouterportP2

assert(b_unique_flow_System_2 = 8001)
```

lemma *"ereal (8001) + 0 = ereal (8001)"* **by** *auto*