Unité Mixte de Recherche 5104 CNRS - INPG - UJF
Centre Equation
2, avenue de VIGNATE
F-38610 GIERES
tel : +33 456 52 03 40
fax : +33 456 52 03 50
http://www-verimag.imag.fr

ac2lus: Bringing SMT-solving and Abstract Interpretation Techniques to Real-Time Calculus through the Synchronous Language Lustre

Matthieu Moy and Karine Altisen

Verimag Research Report n^o TR-2010-2

April 2, 2010

`ac2lus`: Bringing SMT-solving and Abstract Interpretation Techniques to Real-Time Calculus through the Synchronous Language Lustre

Matthieu Moy and Karine Altisen

April 2, 2010

Abstract

We present an approach to connect the Real-Time Calculus (RTC) method to the synchronous data-flow language Lustre, and its associated tool-chain, allowing the use of techniques like SMT-solving and abstract interpretation which were not previously available for use with RTC. The approach is supported by a tool called `ac2lus`. It allows to model the system to be analyzed as general Lustre programs with inputs specified by arrival curves; the tool can compute output arrival curves or evaluate upper and lower bounds on any variable of the components, like buffer sizes. Compared to existing approaches to connect RTC to other formalisms, we believe that the use of Lustre, a real programming language, and the synchronous hypothesis make the task easier to write models, and we show that it allows a great flexibility of the tool itself, with many variants to fine-tune the performances.

Keywords: Real-Time Calculus, Lustre, Modular Performance Analysis, Observer, Formal methods

Reviewers: Florence Maraninchi

Notes:

How to cite this report:

@techreport { verimag-TR-2010-2,
title = { `ac2lus`: Bringing SMT-solving and Abstract Interpretation Techniques to Real-Time Calculus through the Synchronous Language Lustre},
author = { Matthieu Moy and Karine Altisen},
institution = { Verimag Research Report },
number = {TR-2010-2},
year = { 2009},
note = { }
}

1 Introduction

Modern real-time embedded systems are increasingly complex and heterogeneous. Due to real-time requirements, the timing performances require accurate evaluation that help taking or validating decisions on the conception of a system as early as possible in the design process. Many modeling and analysis techniques have been developed among which we can distinguish two families. *Computational approaches* study fine-grain models of the system to represent its complete behavior. The validation of the system using such a model may involve simulation, testing and verification. Simulating precisely an embedded system gives very precise results, but only for one simulation, and one instance of a system. Formal methods such as model-checking, when possible, quickly face the state explosion problem if they are not associated to modular techniques. As opposed to this, analytical techniques, such as Real Time Calculus [1], use purely analytical models, based on mathematical equations that can be solved efficiently. These models can represent in a simple way the amount of events to be processed and how fast the processing occurs. Solving these equations is very fast and may provide, for example, the best and worst cases for performances. The main drawbacks of those techniques is that they imply rough abstraction of the system, leading to results that may lack accuracy. For example, Real-Time Calculus cannot handle the notion of state in the modeling of a system. Recent studies try to compositionally combine the approaches to take the best of both [2,3,4]. We present in this paper another way of combining Real-Time Calculus with computational models, in the case of synchronous programs written in Lustre [5].

Real-Time Calculus (RTC) RTC [1] is a framework to model and analyze heterogeneous system in a compositional manner. It relies on the modeling of timing properties of event streams and available resources with curves called arrival curves and service curves. A component can be described with curves for its input stream and available resources and some other curves for the outputs. For already-modeled components, RTC gives exact bounds on the output stream of a component as a function of its input stream. This result can then be used as input for the next component.

An arrival curve is an abstraction to represent the set of event streams that can be input to (resp. output from) a component; it is expressed as a pair of curves (α^l, α^u). For $k \geq 0$, $\alpha^l(k)$ and $\alpha^u(k)$ respectively provide, for any potential stream, the lower and upper bounds on the number of events that occur during *any* time interval of size k. Similarly, the processing capacity of a component is specified by a *service curve* (β^l, β^u). The number of events that may be processed in any time interval of size k is at least $\beta^l(k)$ and at most $\beta^u(k)$.

These curves do not allow to consider *sequences* of different abstract behaviors, since it is based on information valid for all intervals of a given length. This also implies that no notion of *states* is allowed in RTC components. When using such abstractions for components with several intrinsic modes, the obtained results are particularly coarse. For example, a component with an initial mode and then a stationary behavior cannot be described without unreasonable approximation.

Interfacing RTC with other formalisms to overcome these drawbacks, a commonly used approach is to individually analyze some state-based or new component of an RTC framework with tools that support it and to re-inject the results into the RTC framework. This implies making the interfaces between those tools and the RTC analysis compatible, namely, each component inputs and outputs arrival curves.

This approach has already been used successfully to connect RTC to various other formalisms in the past: [6] proposed a connection to a simulation model that allows testing models or actual embedded systems. [7] presents a connection to event-count automata and [3] a connection to timed automata: they both represent the component and the input arrival curve as event-count (resp. timed) automata and then use a model checker to compute the output arrival curve. The connection with timed automata has been further improved to reduce the size of the models by focusing on a particular shape of curves [2] or by changing the granularity of events [4].

Overview of the approach in this paper we propose to extend the RTC interfacing the Lustre synchronous data-flow programming language. When some component is well-suited for modeling and analysis with Lustre and its toolbox, we propose to individually analyze it and then to use the results in the RTC framework. Other components can still be analyzed with the usual RTC toolbox. We develop a method to analyze Lustre programs interfaced with RTC, with many options that can be tried to obtain some better results. A tool called `ac2lus` supports the framework.

An RTC component can be seen as an arrival curve transformer: taking some arrival curve as input, this specifies an output arrival curve for the component (see Figure 1 (a)). We now improve the expressiveness of the component by writing it in Lustre. This implies to make the interfaces between the Lustre component and the arrival curves compatible. Indeed, the new component, as Lustre is dataflow, computes on a given stream of events, whereas arrival curve describes a set of event streams in a very declarative way using relative time. To be able to come back and forth between the Lustre component and the arrival curves, we develop adapters and back adapters between them (see Figure 1 (b)).

We then run the analysis to compute the output arrival curve. It is done with an abstract interpretation tool (`nbac` [8]) and a model-checker (`kind` [9]) doing bounded model-checking and k-induction via SMT-solving ("Satisfiability Modulo Theory", i.e. SAT + a numeric solver).

We develop the tool `ac2lus`. It provides a library of predefined Lustre components (greedy processing components, fixed-priority scheduler, simple examples of power-managed components...) that the user can use to model a system. Some other components can easily be programmed. The user also provides the input arrival curves which `ac2lus` uses to generate the Lustre code for adapters and back-adapters. The tool implements some runners for the verification tools, which allows to compute the output arrival curves automatically.

The first goal of the approach it to extend the RTC framework by the local analysis of some Lustre component. It can also be used to evaluate some logical or quantitative property on the component. For example, given some arrival curve for a Lustre component, one can use the tool to compute the maximum buffer fill-level.

A similar approach was followed by [10]: they compute maximum delays of data-acquisition modules using Lustre models and abstract interpretation. The main differences with our work are that `ac2lus` allows modular analysis through the use of RTC arrival curves and enables the evaluation of any quantitative measure expressed in the Lustre model whereas [10] is better-suited to compute maximum delay.

Contributions we propose a new combination of the RTC framework with Lustre components which is fully automatized by the tool `ac2lus`. It increases the variety of validation techniques (abstract interpreters, SMT-solvers) available for analysis: if some system fails to be analyzed using some tools, one can change his strategy and try another one. With the same idea, the whole framework contains many variants and is easily configurable and extensible.

Comparing to the classical RTC analysis, those components are more expressive and allows to design state-based systems which is mandatory while studying e.g. power aware systems; in this case, the analysis via Lustre leads to much more precise evaluations than RTC.

Comparing to other state-based RTC interfacing frameworks, and in particular with [2] which is the most recent and the closest of our work :

– the shape of the curves they take into account is a bit less general (since we enable points between segments) and they strictly focus on discrete events whereas we can also handle fluid event models by changing to `real` the typing of event streams (with similar performance);

– we believe that the use of a programming language instead of any other formal models, makes the modeling of new components more intuitive and easier;

– the efficiency of the analysis and its precision are highly correlated, for both approaches. When the analysis succeed, the precision is the same since it is optimal. The performances bottleneck, in both cases, are due to the use of formal validation tools and to the size of the models. As usual, tools behave differently for various shapes of models and it is good having several strategies to try. For example, our approach scales nicely with the order of magnitude of numerical constants, while [2] scales better with respect to the timing constants.

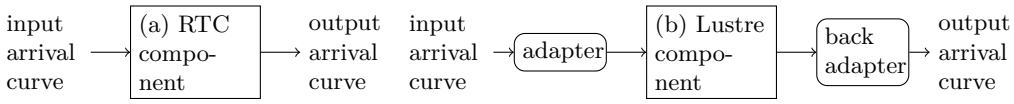

Fig. 1. From RTC analysis (a) to other analysis (b)

Organization the paper is organized as follows. Section 2 gives some details on the Lustre language and shows some RTC components written in Lustre; Section 3 details the adapters and back-adapters, explains the tool and the analysis; Section 4 shows applications of the framework, illustrated on an example and Section 5 gives conclusions and future works.

2 Lustre

We use the synchronous data-flow language Lustre [5] to program the component to be analyzed. This is a simple but formally defined language which is dedicated to program embedded system or abstractions of it.

The motivation for choosing Lustre is multiple. First it is supported by a wide variety of validation tools such as model-checkers [11,9], abstract interpreter [8], testers and simulators. By connecting RTC to the Lustre language, `ac2lus` opens the door to several other formal methods and tools that were not previously available within RTC. Second, the execution model of Lustre fits the RTC model rather well: both are dataflow-oriented. A Lustre program computes directly on event streams, hence, the constraints expressed by arrival curves correspond directly to properties of Lustre flows. This allows to program the interface (adapters) with arrival curve using the same language as the components; we use the technique of the so-called synchronous observers [12] to express the constraints of the arrival curve as a safety property of an event stream. Finally, an interesting property of Lustre is its simplicity of use. Lustre is not only a modeling language, but also an implementation language, that has already been successfully used to implement large systems. We believe that the use of an actual programming language makes the task of writing models for components easier than other formalisms.

Lustre programs manipulate infinite streams of values. When one writes `x = y + 2;` it should be read as "at each clock tick, the value of `x` is equal to the value of `y` plus 2", so if the stream of values for `y` is $1, 5, 12, 42, \ldots$, then the stream of values for `x` is $3, 7, 14, 44, \ldots$. It is also possible to refer to past values of a variable, by using the operator `pre` which means "previous value". Since "previous value" is meaningless at the first instant, the operator `pre` has to be used together with the "initialization operator" `->`: `x -> y` means "`x` at the first instant, and `y` afterwards".

A Lustre function is called a `node`; it takes and computes streams of values as input and output parameters. Figure 2 shows a simple node which computes identity on a stream of integers.

```
node infinite_capacity (in_seq: int)
                returns (out_seq: int)
let
  out_seq = in_seq;
tel
```

Fig. 2. Simple Lustre Node

Model of a Lustre Component to model a component in Lustre, we naturally represent its input/output event streams with input/output Lustre flows. As Lustre programs computes a value "at each clock tick", time is obviously discrete. At each clock tick, a given amount of events arrives

```
node gpc (in_seq: int; in_res: int)
 returns (out_seq: int; out_res: int)
var
  backlog: int; work: int;
  empty_queue: bool;
let
  -- events to compute at the current
  -- instant (accumulated + new work)
  work = in_seq -> (in_seq + pre(backlog));
  -- whether we'll empty the queue at the
  -- current instant
  empty_queue = (work <= in_res);
  -- amount of work accumulated in the past
  backlog = if (empty_queue) then 0
            else work - out_seq;
  -- events produced
  out_seq = if (empty_queue) then work
            else in_res;
  -- resource remaining after running
  out_res = in_res - out_seq;
tel
```

Fig. 3. Greedy Processing Component in Lustre

to (resp. is output by) the component. We do not consider individual events, but instead consider the number of events occurrences during a clock tick.

The node in Figure 2 can be seen as the Lustre model of a component with an infinite processing capacity: it receives events and immediately process them.

We consider the case where event count are integers: this model is usually called the *discrete event* model, as opposed to the *fluid event* model where the amount of data to process and resource can be continuous. Extension to the fluid model would be straightforward by replacing `int` Lustre flows with `real`.

A Common RTC Component in Lustre to illustrate and validate the approach, we first model well-known RTC components in Lustre. A typical example of RTC component is the so-called *Greedy Processing Component (GPC)*. It models a process that enqueues the incoming events in a buffer and treats the events in a greedy fashion while being restricted by the available resource. It can be modeled in Lustre by the program in Figure 3, while `in_res` represents the maximum number of events that can be processed at this instant, and `out_res` the unused resource during this clock tick.

Other examples of Lustre components, that RTC cannot model because of states, will be given in Section 4.

3 Interfacing Lustre and RTC

In this section, we explain how to interface a Lustre component C with RTC arrival curves. This means that we have to transform the constraints of the input arrival curve α_{in} into explicit Lustre flows that complies with those constraints; those flows will then be input into C. On the other side, we have to express a set of explicit flows coming out of C into constraints for the output arrival curve α_{out}.

In fact, for the output part, we do not directly infer α_{out} from the outputs of the component. We build a candidate arrival curve and then check with a verification tool if it conforms to the output streams. The building for the candidate curve is done with a binary search procedure. This

method allows to apply the same technique at the input and at the output; this choice is mainly due to the proof engines we use which are able to verify properties. A valuable extension to our work would be to try a tool such as `aspic` [13] which is able to infer numerical invariants on the program, from which we could deduce α_{out}.

The technique we use is to characterize an arrival curve with an *observer* encoded by a Lustre node. Observers do not *compute* a curve, but they give a yes/no answer to the question "is the event stream compliant with the curve?".

The models of observers we propose are restricted to either finite specification of an arrival curve, namely a finite number of points specifies the curve (individual values $\alpha^u(1), \alpha^u(2), \ldots$ and $\alpha^l(1), \alpha^l(2), \ldots$), a piecewise-affine function, which should be convex for lower-curve and concave for upper-curve (i.e. $\alpha^u(\delta) = \min\{a_1^u\delta + b_1^u, a_2^u\delta + b_2^u, ...\}$ and $\alpha^l(\delta) = \max\{a_1^l\delta - b_1^l, a_2^l\delta - b_2^l, ...\}$), or a combination of both. We call the individual $\alpha^u(i)$ and $\alpha^l(i)$ values *points*, and the linear portions of curves $\alpha^u(\delta) = a_i^u\delta + b_i^u$ $\alpha^l(\delta) = a_i^l\delta + b_i^l$ *segments*. Figure 4 shows an example with 3 points, two segments for the upper part, 3 points, one segment for the lower part.

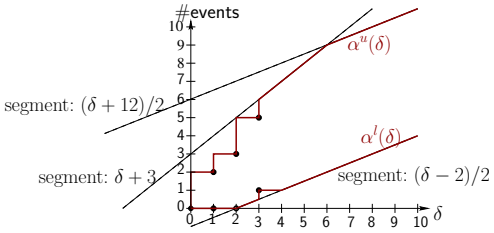

Fig. 4. An example arrival curve

In the rest of the section, we first give a generic modeling of an arrival curve that is available for input and output adapter, called *deterministic observers*. Afterwards, we show how to use them to actually search for the best valid curve and describe the `ac2lus` tool which performs the overall computation. We finish with some variants for the node describing either the input adapter or the output adapter.

3.1 Adapters using Deterministic Observers

In the family of synchronous languages, where the communication between parallel components is the synchronous broadcast, observers [12] are a powerful and well-understood mechanism: an observer is a node that observes the inputs and the outputs of another node and computes some safety property. Here, we model the set of event streams that satisfy an arrival curve with a *deterministic observer* (see Figure 5 for the node). The observer inputs a stream of events called `sequence` and checks whether it satisfies or not the constraints of a given arrival curve (it outputs the Boolean stream `ok`). It exactly characterizes the arrival curve, by the property: an input stream `sequence` satisfies the curve iff `ok` remains `true` forever. The property to be expressed (and the code of the node) is divided into subproperties, one for checking compliance with the points of the curve and one per segment to be satisfied. The last line of the node ensures the property of *permanent failure*: as soon as the arrival curve is falsified, `ok` remains false forever.

Deterministic Observer for Points the part of the arrival curve specified by the points (let us say N points) expresses N constraints, one per time interval from size 1 to size N. The idea of the deterministic observer is to use one counter `c`i per size i of interval and to check its constraint. `c`i represents the amount of events that occurred during the i previous instants. At each new instant, it is updated by the previous value of `c`$(i-1)$ (the number of events in the former window of size $(i-1)$) plus the amount of events that occurred at the instant (current value of `sequence`). The constraint to be verified is: for each i, `c`i is between $\alpha^l(i)$ and $\alpha^u(i)$.

```
node curve_det_obs (sequence: int)
           returns (ok: bool)
let
  ok = ok_points(sequence) and
       ok_segment_0(sequence) and
       ok_segment_1(sequence) and ... and
       (true -> pre(ok));
tel
```

Fig. 5. Deterministic Observer

The node shown in Figure 6 is the deterministic observer for the points of the example curve of Figure 4. The additional variables `inter`i are used at the beginning of the execution: for example, during the first two instants, the constraint on `c3` has no sense since it should hold on interval of size 3. `inter`i expresses the fact that less than i instants elapsed since the beginning of the execution. It can be computed with bounded integers or Boolean encoding (as shown in the code), leading to different analysis performances.

```
node ok_points (sequence: int)
       returns (ok: bool)
var
  c1, c2, c3: int;
  inter1, inter2: bool;
let
  c1 = sequence;
  c2 = sequence -> pre(c1) + sequence;
  c3 = sequence -> pre(c2) + sequence;
  inter1 = true->false;
  inter2 = true->pre(inter1);
  ok =             (0 <= c1 and c1 <= 2)
   and (inter1 or (0 <= c2 and c2 <= 3))
   and (inter2 or (1 <= c3 and c3 <= 5));
tel
```

Fig. 6. Deterministic Observer for Points

Deterministic Observer for Segments the parts of the curves expressed by $\alpha^u(\delta) = \min\{a_1^u\delta + b_1^u, a_2^u\delta + b_2^u, ...\}$ and $\alpha^l(\delta) = \max\{a_1^l\delta - b_1^l, a_2^l\delta - b_2^l, ...\}$ can be expressed as a conjunction of observers for affine functions $a_i\delta + b_i$, where a_i and b_i are constant. We call each observer a *segment* observer. The codes for lower and upper segments are a bit different but both use the common principle of the leaky bucket. We explain it for the upper segment first.

The constraints from the arrival curve expresses the fact that the cumulated amount of events must stay below the segment. We model it as a bucket, whose content correspond to the number of events that can be emitted in burst at the current instant. It initially contains b_i events. At each clock tick, the bucket is refilled by a_i events, and the amount of events that occurred is poured off. After this computation, the bucket's content must still be within $[0, b_i]$: if the new value is above b_i, the extra events are lost, and if it gets below 0, it means the arrival curve is violated.

Figure 7 shows the generic code for the upper segment. The scale factor is a constant input which is used to keep the variables integer, while allowing a_i to be rational.

The lower segment observer `segment_observer_l` is slightly different: the bucket is initially empty, and whenever the bucket's content crosses b_i, the property is violated. The lower segment

```
node segment_observer_u
        (sequence: int; init_val: int;
         refill_speed:int; scale_factor:int)
returns (ok: bool)
var bucket, new_bucket: int;
let
  new_bucket = (init_val -> pre(bucket))
             - (sequence * scale_factor)
             + refill_speed;
  bucket = min(init_val, new_bucket);
  ok = new_bucket >= 0;
tel
```

Fig. 7. Deterministic Observer for an Upper Segment

does not give upper bound on the number of events, hence, emitting more events than the bucket's capacity is allowed, and results in an empty bucket.

In practice, the code is made simpler by considering the value "bucket-content-b_i", which lives in the interval $[-b_i, 0]$. If this value gets above 0, the property is violated.

In the code, the `min` operator becomes a `max` and `>=0` becomes `<=0`, as shown in Figure 8.

```
node segment_observer_l(in_seq: int;
                        initial_value: int;
                        refill_speed: int;
                        scale_factor: int)
returns (in_ok: bool)
var
  bucket: int;
  new_bucket: int;
let
  new_bucket = (initial_value -> pre(bucket))
             - (in_seq * scale_factor)
             + refill_speed;
  bucket = max(initial_value, new_bucket);
  in_ok = new_bucket <= 0 and
    (true -> pre(in_ok));
tel
```

Fig. 8. Observer for lower-curve segments

The observer for the first upper segment of the curve in Figure 4 simply instanciates the generic observer: `ok = segment_observer_u(seq, 3, 1, 1);`

The leaky bucket idea comes from the ancestor of RTC, namely Network Calculus [14], and was already applied to the connection of RTC to computational models in [2], which directly inspired this part. The synchronous hypothesis greatly simplifies the implementation, which becomes just 3 simple Lustre equations including the property itself. Furthermore, the use of synchronous observers here makes the combination of segments easy: to describe a curve made of multiple segments (up and/or low), one can simply use an arbitrary Boolean combination of observers. Unlike [2], we do not have to introduce an additional scheduler automaton which does the combination by scheduling the "must emit" and "can emit" signals emitted by different automata.

3.2 The analysis procedure

This paragraph explains the procedure to compute one output arrival curve α_{out}, given an input arrival curve α_{in} and a Lustre component C. Although, the framework can handle multiple input and output arrival curves, we explain first the procedure for one input and one output, in this paragraph, and then extend it to multiple inputs and outputs in the next paragraph 3.3.

We first express the property that the triple $(\alpha_{in}, C, \alpha_{out})$ is conformant. The building of a candidate for α_{out} is done using a binary-search procedure.

Checking input and output arrival curves (together) the goal here is to validate that given an input arrival curve α_{in}, a Lustre component C and an output arrival curve α_{out}, this triple is conformant, namely: every streams that may be output from C, when it executes with input streams that satisfies α_{in}, should satisfy α_{out}. For both α_{in} and α_{out}, we instantiate deterministic observer nodes which respectively output the Boolean streams `ok_in` and `ok_out` and we combine them to obtain a global observer. This observer expresses the simple property: `ok = ok_in => ok_out`.

Proving that the property is invariant shows that the triple is conformant, if α_{in} has a good shape. Indeed, Lustre observers in general models safety properties and the above observer expresses that at each instant t, `ok_in(`t`) => ok_out(`t`)`. *But*, the observer for α_{in} (resp. α_{out}) characterizes the constraints expressed by the arrival curve with: $\forall t$, `ok_in(`t`)` (resp. $\forall t$, `ok_out(`t`)`). We thus need to express: ($\forall t$, `ok_in(`t`) => (` $\forall t$, `ok_out(`t`))` which is not equivalent to the former formula in general. However, the two formulations become equivalent if the following three conditions are verified [12]:

Permanent failure: if the precondition becomes false, it has to remain false forever. This is ensured by the last line of the observer node `curve_det_obs` in Figure 5.

Determinism: the precondition should be deterministic. It comes from the fact that inputs in the precondition are quantified existentially (since they are on the left part of the implication) while observers use universally quantified inputs.

Causality: the precondition is causal, that is, for any finite sequence s for which the precondition is true, there exists an infinite extension of s such that the precondition remains always true.

This last condition is not true in general since it depends on the shape of α_{in} (see [15] for a characterization of the property on arrival curves and an algorithm to make curves causal). If α_{in} is causal, the global observer exactly characterizes the fact that $(\alpha_{in}, C, \alpha_{out})$ is conformant; otherwise, the analysis is still conservative (the observer may answer false while the property is true).

Building the output arrival curve the tools we are using for verification of the global model was designed to prove properties, not to discover invariants, so a little additional work has to be performed to compute the output analytic description based on the input ones. This step is based on a binary search, with an algorithm which tries different values until it finds the best ones which are provably correct.

Let us explain the procedure for building the points of the curve on an example. To compute $\alpha^u(4)$, we start with the hypothesis $\alpha^u(4) = 0$, generate the observer for this particular curve (assuming $\alpha^l = 0$ and $\alpha^u(t) = +\infty$ for $t \neq 4$). If the curve is incorrect, we try with $\alpha^u(4) = 1$, and then 3, 7, (after trying value n, we try $2n + 1$) ... until we find one acceptable value. At that point, we have one provably correct value, and the previously tried one (or just 0) which is not, and we can proceed with a binary search between these two values. This implies launching the proof engine $log(n)$ times for each point (where n is the value to be computed), but this has not been a problem in practice.

The procedure is quite the same for building the segments of the curve. For each segment $a_i x + b_i$, the two values a_i and b_i has to be evaluated. We first fix b_i to an arbitrarily big value, and perform a binary search on a_i, and then fix a_i to the value just computed to find b_i.

Strictly speaking, a_i could be a rational number, and the binary search should not terminate. Indeed, our implementation uses an equation of the form $(a_i x + b_i)/s_i$, with a_i, b_i, and s_i being integers. We fix s_i to the same value as the input curve (or optionally some other user-supplied value), and do an integer binary search on a_i.

3.3 The `ac2lus` Tool

The approach is implemented in the toolbox `ac2lus`. From the user point of view, one provides a component C to analyze as a Lustre node, having any number of inputs I_i and output O_j. For each input, one must provide an arrival curve α_i. The tool computes the curves $(\alpha_{out})_j$ for each output. Technically, the user provides a *system*`.sys` file to describe the inputs/outputs and to give the name of the Lustre node to analyze, a *file*`.ac` file for each input, and a Lustre file.

The *system*`.sys` file allows one to specify the method to use for generating the adapter of each input/output. It can look like:

```
input: in_seq;
ac_file: power-aware-in.ac;

output: out_seq;
method: nondet_observer;

main_node: power_aware_1;
lustre_file: power_aware.lus;
```

The *file*`.ac` files specify points and segments of arrival curves, and are of the form:

```
points_up: 0, 10, 12;
segment_up: (5x + 2)/1;
points_low: 0, 0, 1;
segment_low: (3x - 9)/1;
```

Internally, the execution is done as follows.

For each I_i, the tool generates the Lustre code for the adapters (as described in section 3.1, or using one of the variants below)[1]. Then, it generates the main Lustre node, which instantiates the input adapters, the component C, and the output adapters. This nodes defines the property to verify, which is of the form `ok =` $(\bigwedge_i$ `ok_in`$_i)$ `=>` $(\bigwedge_j$ `ok_out`$_j)$.

An example of generated main program is given in Figure 9.

We then start the binary search. We compute each output independently. To compute the curve for O_j, we first generate stubs for each $O_k, k \neq j$ for which `ok_out`$_j$`= true`, and apply the binary search algorithm of section 3.2 for O_j to compute each of its points, and optionally a segment to describe the long-term rate. The search consists in trying values, and asking proof engines whether the value is a correct bound or not.

We delegate the verification of the model to different tools. The ability to deal with numerical variables is a must-have, hence plain model-checking (enumerative or symbolic with BDDs) is not an option. `ac2lus` can currently use the abstract interpretation tool `nbac` [8], and the `kind` [9] verifier which is based on k-induction and uses SMT-solving. From the user point of view, both accept Lustre programs as input, and can prove properties of the form "OK is always true". `nbac` works on an abstract domain, and cannot find counter examples, hence its output is either "true" or "don't know". The SMT solver used internally by `kind` actually provides a set of values when given a satisfiable problem, hence `kind`'s output is either "true" or "false with counter example", or can sometimes reach a timeout.

[1] Historically, this part was the first to be implemented, and was translating an arrival curve to a Lustre program, hence the name `ac2lus` for "Arrival Curves to Lustre"

```
include "utils.lus"
include "power_aware.lus"
include "generated-in_seq.lus"
include "generated-out_seq.lus"

node binary_search(in_seq_in_seq: int)
returns (OK_for_nbac, OK: bool)
var
  obs_ok_in_seq: bool;
  out_seq_out_seq: int;
  obs_ok_out_seq: bool;
let
  -- equations for inputs
  obs_ok_in_seq = in_seq_det_observer(in_seq_in_seq);

  -- equation for system
  out_seq_out_seq = power_aware_1(in_seq_in_seq);

  -- equations for outputs
  obs_ok_out_seq = out_seq_det_observer(out_seq_out_seq);

  -- equations for global OK
  OK = not (obs_ok_in_seq) or (obs_ok_out_seq);
  -- nbac complains with Fatal error: exception
  -- Failure("Syntax2semantic.check: final formula contains conditions
  -- on input variables instead of state variables only")
  -- but works if we add a "pre".
  OK_for_nbac = true -> pre(OK);
tel
```

Fig. 9. Generated Lustre Main Node

By default, `ac2lus` launches `kind` on the generated code. If `kind` proves the property or finds a counter-example, we move on to the next value to try. If `kind` reaches a timeout then `nbac` is tried. Optionally, on failure of `nbac`, we try `kind -loop` which sometimes give better results.

Internally, `ac2lus` defines a base class `lustre_proof_runner`, and classes to handle different tools inherit from it (doing the job of launching the tool and parsing its output to tell whether the result is "yes", "no", or "don't know"). We build a list of such objects, and go through the list, stopping after the first tool concludes. This allows trying multiple tools with various options on the same program in a flexible and yet automated way.

The binary search algorithm described above is also improved by an optimization. When computing $\alpha^u(\delta+1)$, we already know the previous values of α^u and α^l and can use them to bootstrap the search: simple RTC theorems show that $\alpha^u(\delta+1) \in [\alpha^u(\delta)+\alpha^u(1), \alpha^u(\delta)+\alpha^l(1)]$, hence, we can start the binary search from this interval.

For robustness, we actually start from such interval, but do not trust it: we check whether the upper bound is correct, and if not, increase it (applying `n = 2*n + 1;`) until we find a provable value. Likewise, we try decreasing the lower bound until we find a value for which the proof actually fails. This way, an incorrect heuristic can only impact the performance of the tool, but not its result.

3.4 Variants for Adapters

The adapter presented above has the advantage of being applicable both for inputs and outputs. However, this is not the only option. Some variants produce Lustre programs with different number of variables, some non-determinism, and yield different results in the proofs.

Generator one commonly used approach to implement an adapter for the input of a component is to write a generator (as done in [2,4,7]), which instead of telling whether an arbitrary event stream is correct, generates a stream which is correct by construction (using non-determinism to be able to generate all possible streams). When using a generator (if one input and one output), the global property to prove is no longer `"ok = ok_in => ok_out"`, but simply `"ok = ok_out"`.

The generator is implemented only for curves without segments (i.e. specified only by a finite set of points). It reuses the idea of the deterministic observer to compute, at each clock tick, the interval $[a, b]$ of admissible values for the flow. It then uses an integer oracle O. If $O \in [a, b]$, then the generator emits O, otherwise, it emits b. This way, we cover all the possible values, and never emit any incorrect one. We perform a causality closure [15] on the curves to ensure that $a \leq b$.

The extension of this generator to curves with segments would be straightforward (but is not implemented): the leaky bucket principle allows saying, for which segment, how many events *can* be emitted, which give an upper bound, and how many *must* be emitted, which give a lower bound. We would then consider the min of all upper bounds and the max of all lower bounds to get the interval $[a, b]$.

Non-Deterministic Observer while using observers, another variant is to allow non-determinism in the observer. In Lustre, non-determinism will be modeled with additional inputs called oracles. With deterministic observer, the property is false iff the observer outputs "OK = false" at some point in time. With a non-deterministic observer, the property is false iff *there exist an oracle* for which the observer outputs "OK = false" at some point in time. Such observer are obviously not convenient for testing, since showing a trace incorrect requires choosing an oracle, but are equivalent to deterministic observers for formal verification when computing the output curve (but as said above, they cannot be applied to the input).

At time t, the deterministic observer checks the number of events for all windows $[t-\delta, t]$. Instead, the non-deterministic observer can chose non-deterministically a time t_0 where it starts checking, and then at time $t > t_0$, checks only the window of time $[t_0, t]$. For any incorrect stream, there exists an oracle t_0 for which the observer emits "OK = false". The advantage of this method is that it requires only one numerical variable to count the number of variables in the interval $[t_0, t]$.

```
node   (oracle: int)
returns (input_stream: int);
var
  lower_bound, upper_bound, count0, count1 : int;
  inter0, inter1, inter2: bool;
let
  -- special-case the first instant
  inter0 = true  -> false;
  inter1 = false -> pre(inter0);
  count0 = if inter0           then 0 else pre (input_stream);
  count1 = if inter0 or inter1 then 0 else pre (count0);

  -- compute the interval of possibilities
  lower_bound = if inter0 then 4
           else if inter1 then max(4, 8 - count0)
           else max(max(4, 8 - count0), 13 - count1);
  upper_bound = if inter0 then 10
           else if inter1 then min(10, 15 - count0)
           else min(min(10, 15 - count0), 19 - count1);

  -- pick a value between lower_bound and upper_bound.
  input_stream = if (lower_bound <= oracle) and (oracle <= upper_bound)
                 then oracle else upper_bound;
tel
```

Fig. 10. Example of a Generator in Lustre

Since this is the only useful use-case, we implemented this observer only for single-points curves (i.e. $\alpha^l(\delta)$ or $\alpha^u(\delta)$ is specified only for one value of δ). The observer for $\alpha^u(3) = 19$ is given in Figure 11. In the implementation, the oracle is a Boolean variable, and the observer starts counting at the first instant when this variable is true. The variables `inter`i are a Boolean encoding of a bounded integer to count the time.

Introducing non-determinism in the segment observers would not be interesting: segment observers already use a small, constant number of numerical variables, which we could not reduce.

Other Variants in some cases, the Lustre code generated from arrival curves can be written differently to get better results in the proof engine. The variants we implemented are based on a trade-off between numerical and Boolean variables. The details are given below.

The two variants implemented are a periodic event generator (using N Boolean variables and no integer to generate events with period N), and a version of the segment observer with leaky bucket where the bucket refill is modeled with this periodic event generator. When the time granularity is fine, and the number of events processed per time unit smaller than 1, this gives a considerable improvement.

Periodic event generator When the arrival curve for an event stream describe a periodic stream, we can actually get rid of the numerical variables, and encode a cyclic automata with Boolean variables. Our encoding uses a one-hot encoding, hence N Boolean variables (but a very simple transition function). An example of generated code is given in Figure 12 for $N = 3$.

Boolean encoding for segment observers Once we have an optimized encoding for a periodic event generator, we can use it to optimize the observer for segments, in the case when the curve segment is of the form $(a_i x + b_i)/s_i$, with the scale factor s_i being a multiple of slope a_i. Instead of considering a leaky bucket refilling at a rate given by an integer variable, we can consider a leaky bucket refilled by 1 every s_i/a_i units of time. Hence, the refill of the bucket is periodic, and can

```
node  (sequence: int; oracle : bool)
returns (ok_points: bool)
var
  count : int;
  inter0: bool;
  inter1: bool;
  inter2: bool;
  inter3: bool;
let
  inter0 = oracle -> oracle or pre (inter0);
  -- count 3 time units.
  inter1 = false  -> pre(inter0);
  inter2 = false  -> pre(inter1);
  inter3 = false  -> pre(inter2);
  -- count the events
  count = 0 -> if inter0 and not inter3
              then (pre count) + sequence
              else pre(count);
  -- the property: 3 time units after the oracle
  -- became true, count must be <= 19.
  ok_points = not inter3 or count <= 19
      and (true -> pre(ok_points));
tel
```

Fig. 11. Example of a Non-deterministic Observer in Lustre

```
node period_3 (dummy: bool)
-- dummy just works around a syntactical limitation
-- of Lustre, which requires at least one input.
returns (clk: bool)
var
  st1: bool;
  st2: bool;
  st3: bool;
let
  st1 = not(st2 or st3);
  st2 = false -> pre(st1);
  st3 = false -> pre(st2);
  clk = st3;
tel
```

Fig. 12. Boolean encoding for a Periodic Event Generator

be encoded as above (in this case, it is a periodic *resource* generator). An example of generated code is given in Figure 13.

The variable `reset_resource` is used to restart the periodic resource generator, to make sure that one doesn't get a resource unit too early after emitting an event. Indeed, the reason are the same as the clock reset of the UTAs in [2].

```
node segment_observer_bool_u_1(in_seq: int;
                               initial_value: int)
returns (in_ok: bool)
var
  bucket: int;
  new_bucket: int;
  refill: int;
  reset_resource: bool;
let
  reset_resource = (in_seq <> 0) and (pre(bucket) >= initial_value);
  refill = if period_1(reset_resource) then 1 else 0;
  new_bucket = (initial_value -> pre(bucket)) - in_seq + refill;
  bucket = min(initial_value, new_bucket);
  in_ok = new_bucket >= 0 and
    (true -> pre(in_ok));
tel
```

Fig. 13. Boolean-optimized encoding for segment observers

Our experiments showed that this optimization is efficient in practice, and we decided to activate it by default in the cases where it is applicable.

4 Applications of the Framework

This section shows applications of the framework and illustrates them on a concrete example. These applications are: **Modular analysis** analyzes the components of a complex system one by one, by using the computed output of the first component as the input for the analysis of the next one;

Global analysis analyzes several components together to get better precision, but sacrificing performance; **Invariant discovery by proving a property:** in addition to computing arrival curves for the output of a system, one can directly prove properties on a component. With a simple binary search algorithm, one can find the best provable bounds for any variable of the Lustre program (for example, the buffer fill-level).

We show that `ac2lus` is able to give results for these 3 applications, in cases where RTC would not be applicable, or too approximate. When the proof engine is able to conclude, the results provided by `ac2lus` are *optimal*.

The experiments were made on a dual-core, Pentium D CPU running at 3.40GHz, with 2GB of RAM.

4.1 A Simple power-aware component

Our original motivation for this work was to be able to model power-managed components, which cannot be modeled and analyzed precisely in pure RTC, because their behavior is state-based.

Figure 14 gives an example of a simple power-managed component. It is not meant to be realistic, but illustrate a state-based behavior: the component starts in sleep mode, and wakes up only when its input buffer fill-level (backlog) reaches a certain threshold. When it starts processing, it does so according to its input resource until the buffer is emptied.

```
-- Parametrized component
node power_aware
  (in_seq, resource, threshold: int)
returns (out_seq: int)
var backlog, work: int;
    serving, empty_queue: bool;
let
  -- things to do at the current instant
  -- (accumulated work + new work)
  work = in_seq -> (in_seq + pre(backlog));
  -- whether we'll wake up
  serving = false -> if (pre serving)
    then (pre(backlog) > 0)
    else (work >= threshold);
  -- whether we'll empty the queue at the
  -- current instant.
  empty_queue = serving
    and (work <= resource);
  out_seq = if serving then
      if (empty_queue) then work
      else resource
    else 0;
  backlog = if serving then
      if (empty_queue) then 0
      else work - out_seq
    else work;
tel
-- Instantiation with parameters (4, 5)
node power_aware_1(in_seq: int)
returns (out_seq: int) let
  out_seq = power_aware(in_seq, 4, 5);
tel
```

Fig. 14. Lustre model for a simple power-aware component

The state of the system is modeled in the equation for `serving`, which means: initially, the component is asleep, then, if it used to be serving, it remains so until the backlog is 0, and otherwise, it goes in service state when the amount of work to do is greater than `threshold`. The other equations are a straightforward adaptation of the GPC component presented in Figure 3. The node `power_aware_1` is basically an instantiation of the previous node, setting the threshold and resource to constant values.

Computing an Output Arrival Curve we perform the analysis on the input curve: $\alpha^u(\delta) = \min\{9\delta, \delta + 15\}$ and $\alpha^l(\delta) = \delta$. The analysis lasts 46 seconds, and launches `kind` 76 times. It gives the following result:

α^u_{out} `= 0, 4, 8, 12, 16, 20, 24, 26, 27, 28, 29`
α^l_{out} `= 0, 0, 0, 0, 0, 2, 6, 6, 6, 6, 6`

Here comes an *explanation* of the result on α^u_{out}: we can see a first linear fragment up to $\alpha^u_{out}(6) = 24$, which corresponds to case where the component empties its buffer, at the speed of 4 events per time-unit. Starting from $\alpha^u_{out}(7)$, the rate decreases because the speed of the component is greater than the rate of arrival. Hence, the backlog is bounded, and the worst-case corresponds to the bursts of the input at the instant when the buffer crosses the threshold.

The result computed by the tool is *optimal* when `kind` answers (i.e. no time-out), namely, the obtained value is the tighter one. To explain this, let us consider a precise example: when trying to prove $\alpha^u_{out}(10) \stackrel{?}{=} 23$, `kind` concluded with a counter-example, which exhibits strictly more than 23 events in 10 time units. This counter-example corresponds to the input sequence $1, 1, 1, 1, 2, 1, 1, 5, 1, 5, 1, 7, 1, 1$. Hence, 23 is provably not a valid bound, while $\alpha^u_{out}(10) = 24$ is valid and thus the optimal value. In the complete example, `kind` concluded (with either true or false property) each time it was called, so this optimality result applies to each point of α^u_{out} and α^l_{out}, which are therefore optimal.

Notice that examining the *counter-examples* produced by `kind` is of great help! It allows the user to better understand the results and to find non trivial corner-cases leading e.g. to bursts or low number of emitted events. For example, on α^l_{out}, the tool shows that it's not possible to remain in sleeping mode for more than 4 time units. The point $\alpha^l_{out}(5) = 2$ corresponds to an execution where the component emits 2 events to finish emptying its buffer, and then emits nothing for some 4 time units. `kind`'s counter-example for $\alpha^l_{out}(5) \neq 3$ shows that this lower bound is obtained for `in_seq` $= 2, 3, 1, 1, 1, 1, 1$, which leads to `out_seq` $= 0, 4, 2, 0, 0, 0, 0$. Even on a simple system like this example, the above counter-example and the proof that $\alpha^l_{out}(5) = 2$ could hardly be found manually.

Comparing with RTC, it would produce very coarse results on the same example. Indeed, it would not be able to handle the two modes of operation and thus could only consider the best-case and worst-case for the availability of resources: the worst-case is the sleeping mode, which does not compute at all, so we would get $\alpha^l_{out}(\delta) = 0, \forall\delta$; in the other side, RTC service curves would not forbid the scenario where the component remains in sleeping mode, accumulating events in its buffer, for an arbitrarily long time, and therefore emit 4 events per seconds for an arbitrarily long time afterwards. Hence,the best we could get is $\alpha^u_{out}(\delta) = 4\delta$.

Quantitative properties to compute bounds on variables of the system in addition to being able to compute output arrival curves, we can also prove properties on the component itself, and find bounds on any of the variable of the system. For example, to compute a bound on the buffer, we add `backlog` as an output of the module, and replace the output observer in the main node with the equation `obs_ok_out_seq = backlog <= N;` By trying different values of N, we can find which is the best bound for backlog. On this example, a quick binary search yields $N = 13$ for which the property is proved, and $N = 12$ gives the counter example `in_seq` $= 8, 9$. Hence, 13 is the best possible bound on the backlog.

The proof do not need to previously compute the output arrival curve and is very quik since it took 0,4 seconds. Notice also that RTC would not be able to get any bound on the backlog of this system.

To sum up on the example, we got the optimal values for $\alpha^u_{out}(\delta)$ and $\alpha^l_{out}(\delta)$ up to $\delta = 10$, a bound for the size of the buffer, and counter-examples showing that any value more precise than the ones we computed would be incorrect, *in less than a minute*, in a case *where RTC does not apply*.

4.2 Discussion on performance

Non-deterministic Observer for Output The non-deterministic observer can be used as a replacement for the output observer (but it cannot apply to the input, as explained earlier). Doing so reduces the number of numerical variables, which can speed up the analysis, but increases the non-determinism of the model, which can have the opposite effect. In our example, the computation is considerably slower than the deterministic observer: it takes 8min5" (compared to 50" for the deterministic version). Also, the proof engine often reaches the timeout, and therefore fails to prove true properties, resulting in a less precise result:
$\alpha^u_{out} = 0, 4, 8, 12, 16, 20, 24, 28, 32, 36, 40$
$\alpha^l_{out} = 0, 0, 0, 0, 0, 2, 6, 6, 6, 6, 6.$

When `kind` reaches the timeout, we fall back to `nbac`. In this example, it happened 37 times, out of which `nbac` was able to conclude 26 times. Indeed, running `ac2lus` using only `nbac` as a back-end gives exactly the same output curves, in only 3min, and inversely, `kind` alone is not able to compute any point.

Generator for Input A generator can be used to specify the input. Since the generator for segment is not implemented, we have to ignore the segment in the input curve, which makes the output approximate regardless of the algorithm. In our case, it is once again slower than the deterministic observer (1min26"), and we also get the same (imprecise) results as with the non-deterministic observer.

Generator for Input, and Non-Deterministic Observer for Output On the other hand, using a generator to specify the input speeds up the analysis if the output is analyzed with a non-deterministic observer: it takes 6min11" (instead of 8min5"), but the computed curves are still suboptimal. But the best configuration is still to use deterministic observers everywhere for this example.

Boolean Encoding Vs Integer Encoding for Segments On the experiments above, the optimization of the segment observer with a Boolean encoding (see section 3.4) were enabled. If we disable them, we get the same output, but it takes 2min18" instead of 46".

4.3 Composing power-aware components

Approach for Analyzing Complex Systems the way to tackle complexity in RTC is to split a system into multiple small components, and reduce the analysis of the system to a set of local, simple analysis. `ac2lus` inherits from this possibility: a component is modeled with a Lustre node, having inputs and outputs, and the components can easily be plugged together.

This can obviously be used to model a system with several physical components, and applies particularly well when the architecture is well pipelined. Actually, this can also be used to model simple scheduling strategies: the Lustre flows can model event streams, but also resource units streams (denoted by β). Input flows represent the amount of resource available during a clock tick (β), and the output flows give the amount of resource remaining after the modeled task has ran (β_{out}). A fixed-priority scheduling between N tasks can therefore easily be represented by connecting the output β^n_{out} to the input β^{n+1} of the next one (component n having more priority than component $n + 1$). An example of a fixed-priority scheduler, re-using the GPC component

defined in Figure 3 page 4, with two tasks is written as:

```
node fp_scheduler(in_res: int;
                  in1, in2: int)
returns (out1, out2: int;
         out_res: int)
var remaining: int;
let
  out1, remaining = gpc(in1, in_res);
  out2, out_res   = gpc(in2, remaining);
tel
```

The connection from a component to another can be done in two ways. Either we compute the arrival curves for the output of the first component, and we apply the usual modular analysis for RTC (using a fix-point computation for circular dependencies [16]), or we actually connect the Lustre nodes to do a global computation.

An Example of Two-Components System we continue the experiment with a second load-dependent component: if its backlog is greater than 4, the component computes at the rate of 10 events per time unit. Otherwise, it computes only 1 event per time unit. The source code for this module is given in Figure 15 .

```
node load_depend_gpc
(in_seq, threshold: int;
 res_fast, res_slow: int)
returns (out_seq: int)
var
  backlog: int;
  work: int;
  empty_queue: bool;
  run_fast: bool;
  amount_computed: int;
let
  -- things to do at the current instant
  work = in_seq -> (in_seq + pre(backlog));

  -- load-dependent speed
  run_fast = (work >= threshold);
  amount_computed =
    if run_fast then res_fast
    else res_slow;

  -- whether we'll empty the queue
  empty_queue = (work <= amount_computed);

  out_seq = if (empty_queue) then work
    else amount_computed;
  backlog = if (empty_queue) then 0
    else work - out_seq;
tel
```

Fig. 15. Load-dependent behavior

We now consider a system composed of the two components. Assembling the components is done in a few lines of Lustre code:

```
node system(in_seq: int)
returns (out_seq: int)
var i: int;
let
  i = power_aware_1(in_seq);
  out_seq = load_depend_gpc(i, 4, 10, 1);
tel
```

Running `ac2lus` on this system, with the same arrival curve as in the previous section for the input, we get the output (in 10min44"):

α^u_{out} = 0, 5, 9, 13, 17, 21, 24, 25, 27, 28, 30
α^l_{out} = 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0

In this case, the curves are approximate: we run the proof engine with a timeout of 10 seconds, and the results show only the provable points of the curves, within this timeout. In this experiment, the timeout has been reached when trying to prove the point $\alpha^u_{out}(9) \stackrel{?}{=} 27$, hence we know that $\alpha^u_{out}(9) = 28$ is an upper-bound, but the tool did not prove that $\alpha^u_{out}(9) = 27$ was not (in this case, 28 is actually the best bound, and `kind` proves it if we increase the timeout to 20). The points $\alpha^l_{out}(\delta) = 0$ are provably the best points for $\delta \leq 4$, but the next points are approximate (and indeed, disappointing since the tool could not prove anything here).

Another option is to run the analysis in a modular way: first, analyze `power_aware_1`, get the output arrival curve, and use this arrival curve as input for the local analysis of `load_depend_gpc`. This approach gives different results:

α^u_{out} = 0, 6, 10, 14, 18, 22, 26, 28, 80
α^l_{out} = 0, 0, 0, 0, 0, 1, 4, 5, 0

The results are exact bounds given the input arrival curve up to $\delta = 7$, but the arrival curve is indeed an abstraction of the real possible behaviors possible as the input of the second module, which explains the less precise results on α^u_{out}. Starting from $\delta = 8$, the models given to the prover are too big and most of the proofs fail, hence the grossly approximated results $\alpha^u_{out}(8) = 80$ and $\alpha^l_{out}(8) = 0$.

Performance-wise, the first points of the curves are computed much faster when using the modular method: computing the first 5 points takes 8 seconds modularly, and 34 seconds when done with the global approach. The complexity grows linearly with the number of components of the system. Regarding points of the curves corresponding to a bigger δ, the situation is different: the complexity of the Lustre program to prove is dominated by the complexity of the observers.

The approach is flexible and allows multiple ways to analyze the same system. Obviously, since each analysis give a valid bound, the min/max of all analysis is also valid and may give better bounds. For the example, combining both results, and applying subadditive closure, we get:

α^u_{out}= 0, 5, 9, 13, 17, 21, 24, 25, 27, 28, 30
α^l_{out}= 0, 0, 0, 0, 0, 1, 4, 5, 5, 5, 5

4.4 Evaluation and Comparison

Apart from obtaining output arrival curves, the above experiments show that: the tool provides counter-examples which enable subtle understanding of the system; quantitative or logical properties can easily be expressed and directly analyzed with the tool; many strategies of the tool can be tried to obtain the tightest results.

Compared to [2], the closest approach to our work, component can be easily written and extended. For example, the fixed-priority scheduler is just 2 lines of Lustre, while the model with Timed Automata is far less trivial. Both tools use exact model-checking (we use abstract interpretation through `nbac` as a fall-back when `kind` fails). When the analysis succeeds, they both give the same result. But on both sides, the proof may be too long, and may have to be stopped with a timeout, leading to imprecision. The bottleneck is therefore the performance.

To the best of our knowledge, there exist no comprehensive benchmark for the problem we are solving, hence, we can compare only with a few examples, usually chosen to perform well with the tool they benchmark.

We tried the case study of [2] on `ac2lus`, but it showed bad results: we are able to compute a few points of the curves, but not enough to get relevant results, and the long-term rate `ac2lus` computes is over-approximated by a factor 2. Our interpretation of these bad results is that our model uses large timing constants (we wrote the model with a timing granularity of 1ms), and the number of variables we use is proportional to the order of magnitude of numerical constants. Uppaal, used by [2], uses an efficient symbolic encoding of time (based on zones), and doesn't suffer from this.

On the other hand, our experiments with timed automata and RTC [4] showed that the performance of the analysis was proportional to the order of magnitude of counters (our understanding is that counters are managed enumeratively by Uppaal). `ac2lus` and the underlying model-checkers are far less sensitive to this. For example, we tried multiplying all the constants of the model (and arrival curves) of the system of section 4.3 by 20 and re-run the analysis, which took 44min instead of 6min11 (dividing the result by 20 yields the same result as the initial one). The performance degradation is here less than linear with respect to the size of counters (this is partly due to the number of iterations of the binary search). With an enumerative algorithm, we would pay a factor 20 in four places: the generator, each of the two components, and the observer.

5 Conclusion

We presented `ac2lus`, a new combination of the RTC framework with the Lustre language. It allows to model the system to be analyzed (or parts of it) as Lustre components with inputs specified with arrival curves; the tool can compute output arrival curves using modular analysis or global analysis and it can also evaluate some quantitative property on any variable of the components. As other RTC interfacing techniques [2,7,3], this results in a considerable increase of the expressiveness of the framework, essential to model power-managed components, while keeping the modular aspect of RTC. Nevertheless, compared to them, `ac2lus` complements the state of the art by allowing a wider variety of modern methods to be used, namely Abstract Interpretation and SMT-solving. As a side effet, we believe that the use of a synchronous programming language makes easy for the user to write models, and allows a lot of variants of the adapters to fine-tune the performances.

In the future, we plan to try other verification tools as alternatives to `kind` and `nbac`. The tool `aspic` [13] sounds promising. It adds acceleration techniques to abstract interpretation, which should apply to the Lustre code we generate for adapters. Also, `aspic` does not only prove properties, but it is also able to *discover* invariants. This could allow us to get rid of the binary search algorithm, and launch the tool only once to compute a value.

An interesting application of `ac2lus` would be to consider the counter-examples provided by the proof engines for the last points which fails during the binary search. These counter-example exhibit an execution for which the computed bounds (α^u, α^l) are actually reached. These executions can be interesting for diagnosis, and could also be used to generate test-cases for the actual system, since they exhibit corner-case behaviors, hardly reproducible randomly or manually.

Our long term goal is to be able to actually handle energy: handling state-based behavior in RTC allows proving timing properties on energy-aware systems, but still gives no information on energy consumption.

Finally, we are looking for alternatives to RTC's arrival curves to model abstraction of event streams. After introducing state-based behaviors in the components, we believe the next, logical step is to introduce the same in the interfaces between components.

References

1. L. Thiele, S. Chakraborty, and M. Naedele, "Real-time calculus for scheduling hard real-time systems," in *ISCAS*, 2000. 1, 1

2. K. Lampka, S. Perathoner, and L. Thiele, "Analytic real-time analysis and timed automata: A hybrid method for analyzing embedded real-time systems," in *EMSOFT*, 2009. 1, 1, 1, 3.1, 3.4, 3.4, 4.4, 5
3. "Cats tool," 2007, `http://www.timestool.com/cats`. 1, 1, 5
4. K. Altisen, Y. Liu, and M. Moy, "Performance evaluation of components using a granularity-based interface between real-time calculus and timed automata," in *QAPL*, 2010. 1, 1, 3.4, 4.4
5. J.-L. Bergerand, P. Caspi, N. Halbwachs, D. Pilaud, and E. Pilaud, "Outline of a real time data-flow language," in *RTSS*, 1985. 1, 2
6. S. Künzli and L. Thiele, "Generating event traces based on arrival curves," in *MMB*, 2006. 1
7. L. T. Phan, S. Chakraborty, P. Thiagarajan, and L. Thiele, "Composing functional and state-based performance models for analyzing heterogeneous real-time systems," in *RTSS*, 2007. 1, 3.4, 5
8. B. Jeannet, "Dynamic partitioning in linear relation analysis. application to the verification of reactive systems," *Formal Methods in System Design*, 2003. 1, 2, 3.3
9. G. Hagen and C. Tinelli, "Scaling up the formal verification of Lustre programs with SMT-based techniques," in *FMCAD*, 2008. 1, 2, 3.3
10. L. Morel, J.-P. Babau, and B. Ben-Hedia, "Formal modelling framework of data acquisition modules using a synchronous approach for timing analysis," in *WRTP/RTS*, 2009. 1
11. N. Halbwachs, F. Lagnier, and C. Ratel, "Programming and verifying critical systems by means of the synchronous data-flow programming language LUSTRE," *Transactions on Software Engineering*, 1992. 2
12. N. Halbwachs, F. Lagnier, and P. Raymond, "Synchronous observers and the verification of reactive systems," in *AMAST*, 1993. 2, 3.1, 3.2
13. L. Gonnord and N. Halbwachs, "Combining widening and acceleration in linear relation analysis," *Lecture Notes in Computer Science*, 2006. 3, 5
14. J.-Y. Le Boudec and P. Thiran, *Network Calculus*. Springer Verlag, 2001. 3.1
15. K. Altisen and M. Moy, "Arrival curves for real-time calculus: the causality problem and its solutions," in *TACAS*, March 2010. 3.2, 3.4
16. B. Jonsson, S. Perathoner, L. Thiele, and W. Yi, "Cyclic dependencies in modular performance analysis," in *EMSOFT*, 2008. 4.3