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

Connecting Real-Time Calculus to the Synchronous Programming Language Lustre

Karine Altisen and Matthieu Moy

Verimag Research Report n^o TR-2009-14

September 2009

Connecting Real-Time Calculus to the Synchronous Programming Language Lustre

Karine Altisen and Matthieu Moy

Verimag

September 2009

Abstract

Keywords: Real Time Calculus, computational models, lustre, nbac, abstract interpretation, generator, synchronous observer

Reviewers: Florence Maraninchi

Notes:

How to cite this report:

@techreport { verimag-TR-2009-14,
title = { Connecting Real-Time Calculus to the Synchronous Programming Language Lustre},
authors = { Karine Altisen and Matthieu Moy},
institution = { Verimag Research Report },
number = {TR-2009-14},
year = { 2009},
note = { }
}

1 Introduction

The Verimag laboratory has been studying various kinds of embedded systems models for more than 20 years. A *model* can either be further refined to an implementation, or be used to validate a specification and evaluate the performances of a system before it is actually built. In the latter case, we call the approach *virtual prototyping*. Verimag has already applied such an approach to the various levels of abstraction that are used to model systems-on-a-chip [Cor08] and sensor networks [Sam08]. In both cases, the approach is fully based on the *executability* of models.

In this document, we report on a work done to bridge the gap between computational (or executable) models, and analytical models. This work has been initiated by discussions with Lothar Thiele's group at ETHZ.

1.1 Computational vs Analytical Models

One of the main benefits of executability is that it allows comparing the model with an implementation, and does not restrict a priori the amount of details that can be put in the model. This helps getting confidence in the faithfulness of the model.

For instance, in the context of sensor networks, such *computational* models allow to mimic the behavior of the hardware parts, with respect to energy consumption. The energy cost of a message sent from one node and reaching the sink, is computed according to the detailed energy modes of the radio components, driven by the detailed protocols, and connected to the application code. Conversely, in more abstract models for sensor networks, a fixed energy cost is associated with each message reaching the sink node, regardless of what happens when it is transmitted (collisions, retransmissions, etc.). The latter model is easier to analyze, but when the activity of the network has a high spatial and temporal variability, the two models start diverging. The model that mimics the behavior of the hardware is more likely to be faithful, than a model where the cost associated with a message is an average value.

In the context of systems-on-a-chip, early timing performance evaluation may be needed for the dimensioning of the system. A typical source of inefficiency in such systems is what happens on the bus. In order to be relevant for performance evaluation, a model of the bus has to show at least an abstraction of the arbitration policy, and this is usually done with a computational model, in methods like transaction-level modeling.

On the other hand, detailed computational models can be slow to simulate, are hardly compositional (in the sense that an execution of a individual components are usually not sufficient to get results for the complete system), and obviously, are limited to a finite number of executions. Worst-cases and best-cases can only be estimated, not computed exactly.

If detailed computational models can be considered as one end of the full range of models, *analytical models* are at the other end. Their respective advantages and drawbacks can be pictured as shown by Figure 1. The interesting and feasible models are placed on a diagonal (non analyzable and unprecise models are useless, whereas very detailed and easily analyzable models are hopeless). On this diagonal, an important point is the limit between models that have *states*, and models that have not. Detailed computational models clearly have states. Abstract analytical models like the Real-Time Calculus do not (see more details below).

In a lot of embedded systems we would like to model and analyze, the notion of state is important, because it allows to model various *modes* of the system. For instance, a multimedia system-on-a-chip will rely on a component called the *power manager*, which can turn some other components on and off. A precise estimation of the energy consumption has to take this behavior into account, otherwise it is too abstract.

One of the main reasons to try and bridge the gap between analytical and computational models is the hope to find models in which we can express such mode behaviors, while retaining at least part of the efficient analysis of pure analytical models.

1.2 Modular Performance Analysis and Real-Time Calculus

The MPA-RTC [TCN00] approach provides a framework for compositional analysis of performances such as end-to-end delays and buffer sizes. It is able to compose different types of processes, scheduling policies

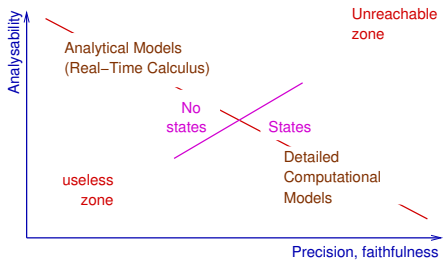

Fig. 1. Computational vs Analytical Models

and resource allocations. Each component has an analytic specification which computes upper and lower bounds on the execution times, given upper and lower bounds on the number of events to be computed and resources available, in any time interval.

The components are described at a very high level of abstraction. The abstraction on the behaviors of the inputs and outputs is the notion of *arrival curve*. An arrival curve is a pair of functions $\alpha^l(\Delta)$, $\alpha^u(\Delta)$ that give, respectively, the minimum and the maximum number of events in any time interval of length Δ. This abstraction does not allow to consider *sequences* of different abstract behaviors, since it is based on some information valid for all intervals of a given length. This also implies that no notion of *states* is allowed in RTC components, because an RTC component is an arrival curve transformer.

When using such abstractions for components that have several intrinsic modes, the results obtained are particularly coarse. For example, a component with an initial mode and then a stationary behavior cannot be described without unreasonable approximation.

1.3 Related works: connecting MPA-RTC to other formalisms

To overcome these drawbacks, an approach is to individually analyze some state-based or new component of an MPA framework with tools that support it and to re-inject the results into the MPA framework. This implies making the interfaces between those tools and the RTC analysis compatible.

An RTC component is specified as a transformer of input arrival curves into output arrival curves. Depending on the component, the RTC analysis then computes the output curves, given the input ones. To replace the RTC analysis with other tools that support state-based components (fig. 2), we thus have first to make the interfaces compatible: the tools should be able to read and produce arrival and service curves. Second, they should compute the same kind of results.

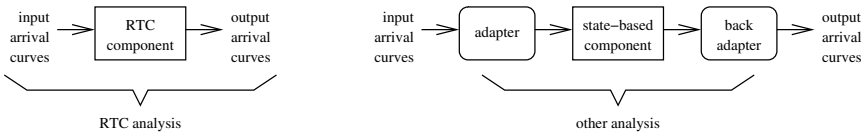

Fig. 2. From RTC analysis to other analysis

This approach has already been used successfully to connect RTC to various other formalisms in the past: ETHZ proposed a connection to a simulation model to allow testing methods where formal methods are not applicable, or using an actual embedded system [KT06] and a connection to Event-count automata (ECA) [PCTT07]. Independently, Uppsala university developed a tool called CATS [KMY07,Mok07,DAR07], which uses a connection from RTC to timed-automata, and allows the use of the Uppaal [LPY97] model-checking tool for timed automata to compute the arrival curves.

More recently, some work in the Verimag laboratory by Yanhong Liu augmented the CATS approach with the notion of granularity [LAM09], and another approach for the connection to timed automata has

been proposed by Kai Lampka from ETHZ [Lam08]. This report uses a comparable approach, but uses discrete-time, and synchronous languages. These three works are part of the collaboration between Verimag and ETHZ within the Combest project.

1.4 Structure of the report

The next section presents some basic MPA-RTC notions; section 3 provides the basics of the synchronous language Lustre that we are using, and the analysis that are possible on this language. Section 4 presents our contribution in detail.

2 Definition of The MPA-RTC Framework

The framework of Modular Performance Analysis for Real Time Calculus provides a compositional abstraction for the modeling and the analysis of real time distributed systems: it mainly allows to compute best and worst case timing properties on the behavior of the system. It describes a network of connected components; the basic components (RTC component) are triggered by streams of events and fed by streams of available resources; those streams are characterized by arrival curves and service curves.

The concrete view of an RTC stream is characterized by a cumulative function R : Time $\rightarrow \mathbb{N}$, where Time is the domain of time (positive real numbers or positive integers usually). $R(t)$ represents the cumulative number of events in the stream from date 0 to date t. The abstract view of the same RTC stream is given by a pair of arrival curves such that:

$$\forall t, \forall \Delta \quad \alpha^l(\Delta) \leq R(t+\Delta) - R(t) \leq \alpha^u(\Delta)$$

This means that in any sliding windows of length Δ, the number of events is upper and lower bounded by the upper α^u and lower α^u arrival curves (this is illustrated in Figure 3). Upper and lower service curves $\beta^u(\Delta)$ and $\beta^l(\Delta)$ are similarly defined, but give the amount of computing resource available during windows of time of length Δ.

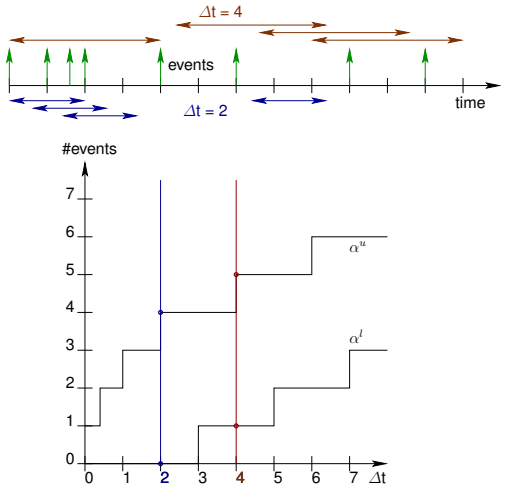

Fig. 3. Arrival curves and event streams

An RTC component is a transformer of such curves. In the concrete view, it correlates input and output cumulative functions, depending on the behavior of the component. The abstract view provides an analytic correlation between input and output arrival and service curves. This is illustrated in Figure 4, where α^u, α^l represent the arrival curves for the input events, β^u, β^l represent the arrival curves for the processing units (i.e. available resource). $\alpha^{u'}$, $\alpha^{l'}$ give the output rate of events and $\beta^{u'}$, $\beta^{l'}$ give the remaining resource

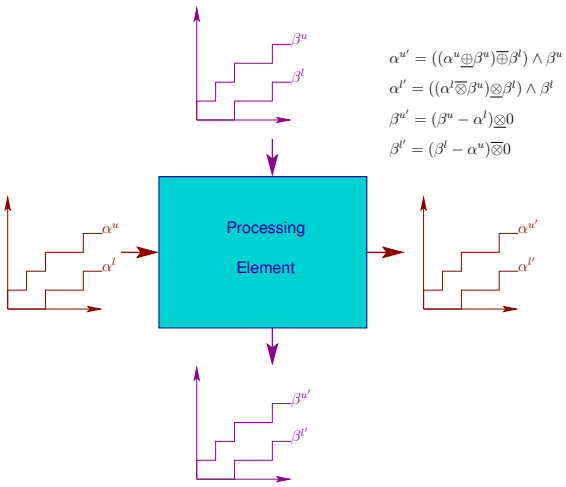

Fig. 4. RTC Component

(that can be used for example as the service curve of another component sharing the same resource). The formulas are given for reference but will not be detailed here.

The performance of a given system modeled as MPA network is obtained by computing input and output arrival and service curves for the whole.

As the method uses analytic models and computations, it provides exact solutions (worst case and best case bounds) and it usually scales up for large systems. But it is not easily adaptable to new components: a new analytic study has to be done for each new component or way to connect them. Furthermore there are strong limitations on the types of components that can be handled: no state dependent components can be modeled.

3 Using Lustre Programs and Abstract Interpretation

In this work, we propose to write the state-based component with the programming language Lustre [BCH+85] and to perform the analysis using the abstract interpretation tool nbac [Jea03].

3.1 Lustre programs

Lustre is a synchronous data-flow programming language. It is simple and formally defined .It is well suited to program (state-based) embedded system or abstractions of it. Lustre programs can be tested, verified against a property (see [HLR92]). Furthermore (and overall), it also allows to program the adapter (see fig. 2) and the back adapter/wrapper with same language, by using the techniques of the so-called synchronous observers [HLR93]. We will discuss this further on paragraph 4.

The idea behind Lustre is to 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 `x` is $1, 5, 12, 42, \ldots$, then the stream of values for `y` is $3, 7, 14, 44, \ldots$. Of course, it is also possible to refer to past values of a variable, by using the operator `pre` which means "previous value of". 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".

For example, the following program implements a counter:

```
node counter(x: bool) returns (y: int)
let
  y = 0 -> (if x then pre(y) + 1 else pre(y));
tel
```

This simple Lustre program states that at the first instant, y is equal to 0, and afterwards, when x is true, the new value of y is equal to its previous value plus one, otherwise, y keeps its previous value. In short, y gives the number of times x has been true since the origin of time.

3.2 Abstract Interpretation of Lustre Programs

Abstract interpretation [CC77] is a common verification technique that can deal in particular with unbounded data-types such as integers. The abstract interpretation theory not only allows checking invariant on a program, but can also "discover" invariants (for example, give the interval of possible values for a variable at a point of a program). Abstract interpretation is based on the notion of abstract domain, which is a way to over-approximate the set of reachable states of the program. It allows a trade-off between performance and precision by choosing the abstract domain appropriately.

In our case, dealing with numeric variables is a must-have: our approach is based on synchronous observers which are Lustre programs representing arrival curves, and therefore containing numerical variables to count the number of occurrences of an event in a time window. These programs can be handled by abstract interpretation tools: we use the tool nbac [Jea03].

4 Detailed Proposition

Since our goal is to represent arrival curves with Lustre programs, where an event can only arrive at a clock tick, we restrict ourselves to the discrete time model. Discrete-time arrival curves are less expressive than continuous time ones, but a continuous-time arrival curve can be approximated with an arbitrarily small error by choosing the appropriate time granularity.

Our state-based component is now a program written with arbitrary Lustre. The component takes as input (resp. output) a flow of integers, representing the number of event occurrences at the current clock tick that the component receives (resp. emits).

For example, the Lustre component representing a component with an infinite processing capacity would be:

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

and a shaper (limited, constant, processing capacity, with a buffer to manage the overload) can be modeled this way:

```
node queue_transformer (in_seq: int; max_speed: int)
                returns (out_seq: int)
var
  backlog: int; work: int; empty_queue: bool;
let
  -- things to do 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 <= max_speed);

  -- amount of work accumulated in the past.
  backlog = if (empty_queue) then 0 else work - out_seq;

  out_seq = if (empty_queue) then work else max_speed;
tel
```

We now focus on the adapters and on the analysis part: our Lustre program is parameterized by a stream of events as inputs and outputs another stream of events, while the MPA framework provides arrival curves

for input, and expects arrival curves as output. The adapters will allow converting arrival curves into sets of possible sequences of events, and the other way around.

4.1 Synchronous Observers

We chose to model the adapter and the back adapter with the same synchronous observers technique, written in Lustre [HLR93]. Here comes a short summary on what we need about Lustre observers.

Generally speaking, a Lustre observer models a *safety property* P a Lustre program has to verify. This property may depend on both the inputs and the outputs of the program.

The properties we are dealing with are, as it is often the case, safety properties in the form "Given an input stream satisfying a precondition, the system outputs a stream satisfying a property". On the other hand, Lustre allows us to express properties in the form "at each point in time, given an input satisfying a precondition, the system's output will satisfy a property". In other words, we can express: always$\big(P(t) \Rightarrow Q(t)\big)$, and we want to prove: always$\big(P(t)\big) \Rightarrow$ always$\big(Q(t)\big)$ which are not equivalent in general. However, the two formulations become equivalent if the following three conditions are verified [HLR93]:

Permanent failure: if the precondition becomes false, it has to remain false forever.

Causality: the precondition is *causal*, that is, for any finite sequence $F = i_0, i_1, \ldots, i_t$ for which the precondition is true, there exists an infinite sequence prolonging F such that the precondition is always true.

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

The first point is easy to work around: once the property becomes false, we keep it false forever. In Lustre, if `ok` is the property to be transformed, the transformation is as simple as:

```
always_ok = ok and (true -> pre(always_ok)).
```

The second can be solved in general with "non-blocking restriction" [HLR93]. In the particular case of observers for arrival curves, we can guarantee the causality with a pre-processing of the arrival curves themselves.

The third gives us a constraint on the way to write the generators: it should not use oracles (input variables) to model non-determinism.

With the 3 conditions satisfied, if the observer outputs a "false" at time t, it means there's an execution for which the precondition has not been falsified up to t and which falsifies the condition. This is equivalent to saying that the property has a counter-example.

4.2 Adapter: from arrival curves to streams of events

We show how to program an adapter as a deterministic Lustre observer. Given a pair of (lower and upper) arrival curves, the adapter provides a way to generate an event stream conforming to the arrival curves. There are two ways to achieve this result: a generator, that will produce such stream, or an observer, that will read an arbitrary stream and check its conformance. We choose the second approach, and therefore generate a piece of Lustre code that checks the conformance of an event stream.

As mentioned above, we restrict ourselves to discrete-time, discrete-event, and finite time window. Let N be the biggest time window that appear in the arrival curve. The observer has to check the number of events in any window smaller or equal to T. We implement this by checking, at each instant t, that the number of events received in the time windows $[t-1, t]$, $[t-2, t]$, $\ldots$, $[t-N, t]$ are between their lower and upper bounds.

The trick here to implement this verification with a reasonable number of variables (N counters) is to re-use the counters from on instant to the next: the number of events in the time window $[x, t+1]$ is the number of events in the time window $[x, t]$ plus the number of events received at time $t+1$, so the event counter for a window of size s at time t can be obtained by adding the number of events at the current instant to the counter for a window of size $s-1$ at the previous instant.

An observer is generated by replacing `mi` and `Mi` by the values of $\alpha^l(i)$ and $\alpha^u(i)$.

```
-- deterministic observer, with 3 counters
-- (one for each size of interval)
node AC_det (i: int) returns (OK: bool);
  count1, count2, count3: int;
let
  count1 = i;
  count2 = i->(pre(count1) + i);
  count3 = i->(pre(count2) + i);
  OK =  m1 <= count1 and count1 <= M1
    and m2 <= count2 and count2 <= M2
    and m3 <= count3 and count3 <= M3
    and (true -> pre(OK)); -- never become true again
                           -- after being false once.
tel
```

The actual program is actually slightly more complex: at instants $t < N$, some time windows would "cross" the origin of time, so their corresponding counter should actually be ignored. A real observer is actually as follows:

```
node input_observer  (i: int) returns (ok: bool)
var
   time: int;
   ok_now: bool;
   c1: int;
   c2: int;
   c3: int;
let
   time = 0 -> pre(time) + 1;
   c1 = i;
   c2 = i -> pre(c1) + i;
   c3 = i -> pre(c2) + i;
   ok =                                  (m1 <= c1 and c1 <= M1)
    and (if time < 1 then true else (m2 <= c2 and c2 <= M2))
    and (if time < 2 then true else (m3 <= c3 and c3 <= M3))
    and (true -> pre(ok));
tel
```

It trivially satisfies the "permanent failure" and "determinism" conditions above, but is not necessarily causal. In other words, it is possible that at some point in time, the lower bounds forces a number of event greater that l while the upper bound forces this number to be lower that u with $u < l$. But this causality problem actually comes from the way the curves are given. Indeed, given a pair of curves, we can get an equivalent pair of curves that yields a causal observer. The transformation is done by a few iterations on the curves, each iteration being $O(n^2)$ in complexity. In other words, we can get rid of the causality problem once and for all by a pre-processing on the curves.

4.3 How to compute output arrival curves?

Given the observer above, we can express the question "for any input sequence complying with (α^u, α^l), will the system output a sequence complying with $(\alpha^{u'}, \alpha^{l'})$?", and ask an abstract interpretation tool to prove it. On the other hand, our goal is to actually *compute* $(\alpha^{u'}, \alpha^{l'})$ (or at least, a conservative approximation of it).

The tool we are using for the proofs, nbac, 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.

For example, to compute $\alpha^u(4)$, we start with the hypothesis $\alpha^u(4) = 1$, 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 $\alpha^u(4) = 2$, and then $4, 8, 16, \ldots$ 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 (where n is the value to be computed) instead of launching an invariant-discovering tool just once, but this has not been a problem in practice.

4.4 The ac2lus Toolbox

The whole approach is implemented in a simple C++ toolbox. The curves are manipulated using arrays of positive integers.

The algorithms implemented are:

- Parser/pretty printer for a textual format for arrival curves,
- Observers generation,
- Binary search calling the tool nbac directly from within C++,
- Sub-additive and "causalification" closure.
- Visualization with GNUplot,

As an extension to plain arrival curves, we allow the value "-1" to mean "no constraint for this time window", which allow manipulating "sparse" arrival curves (they still consume memory, but the observer generated is simpler).

When normalizing the curves before generating the observer, our algorithm notices when a point of the curve can be recomputed from others, and replaces these points with "-1" values. When the last points are "-1", the tool trim the curves to the smallest prefix needed. This allows working with a minimal set of variables in the observers.

5 Conclusion

5.1 Summary

In this task of the Combest project, we are currently exploring an approach similar to the connection of RTC to timed automata [Lam08], using Lustre to model the components. Our approach is fully based on the notion of synchronous observers to verify properties.

The connection to the RTC framework is done with a Lustre observer on the inputs of the system, and another on the output of the system. The well-formedness of observers depends on some properties which we can ensure on the analytic descriptions. The components are also expressed as Lustre programs taking streams of integers as inputs and outputs; this allows the modeling of complex behaviors, including state-based behaviors which are clearly needed to model embedded systems, especially energy-aware systems.

Once the system is modeled and the synchronous observers are set up, one can use all the available verification techniques for synchronous systems. We are using abstract interpretation which can deal nicely with integers (which appear naturally in our programs, whose main task is to count events), and can scale up better than model checking if one accepts to loose some precision. As the tool we are using for the proofs was designed to prove properties, not to discover invariants, 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.

We developed an implementation of the complete framework for the connection from RTC to Lustre, and back from the Lustre world to RTC. This implements the pre-processing on arrival curves, the observer generation algorithm, and a binary search algorithm calling the external tool nbac.

5.2 Comparison with [Lam08]

As mentioned in the introduction, this work comes from a recent collaboration between Verimag/SG and ETHZ. It has been motivated by the limitations of the MPA-RTC framework which leads to similar piece of works in their team and ours. The main differences with their task (A) are:

- use discrete-time instead of continuous time (both within the model and in the analytic description of the event streams);

– use of abstract-interpretation instead of model-checking to compute the arrival curves;
– use of a programming language (Lustre) instead of an automata-based formalism.

This work is another attempt to bridge between the RTC frameworks and state-based formalisms and a particular attempt towards a programming language.

5.3 Perspectives

The work presented here is only at its preliminary stage. A lot is still to be done to be applicable to actual systems.

First, the approach has only been tested on very small examples of transformers. We plan to write larger and more realistic examples to test our tool, and compare the results with other approaches. By "result", we mean here both the precision, and the performances of the proof engine. The precision can easily be better than pure RTC-MPA, and we have good hope to scale better than the timed-automata based approaches, but this claim has to be verified with a benchmark. We plan to write a common benchmark to compare our work with Lustre, Y. Liu's work [LAM09] and K. Lampka's [Lam08].

Then, the approach itself can be improved. One worthy experiment would be to try different proof engines. Remaining in the domain of abstract interpretation tools, we plan to compare ASPIC [Gon07] and nbac on the same programs. ASPIC is a tool recently developed in Verimag. It is known to be faster and more precise than nbac on some programs. We also want to try a version of nbac able to deal with the "octagon" abstract domain instead of polyhedra as we are doing now, and to experiment with invariant discovery in nbac to replace our binary search algorithm.

We presented one way to implement an observer for a pair of curves. This algorithm has the advantage of being general, and can check the conformance of an event stream for any arbitrary finite and discrete RTC curves. But in some specific situations, different approaches may give better results. We plan in particular to:

– adapt the algorithm presented in [Lam08] to the case of discrete-time. This could considerably reduce the number of numerical variables to use for piecewise linear curves;
– implement a non-deterministic observer. We have seen that the observer for the input stream has to be deterministic, but it would be possible to make the output stream's observer non-deterministic. Instead of checking the number of events for *all* time window, the observer could verify this number for *any* time window, choosing the start and the size of the window in a non-deterministic way.

Also, our approach currently uses an observer for the input stream. It would be interesting to allow using a non-deterministic generator instead. The difference is that an observer checks the conformance of a stream, while the observer would generate a stream which would be correct by construction. This may change the performance of the abstract interpretation tool, and would also allow the use of a testing environment instead of formal proofs. Testing wouldn't give strong upper and lower bounds, but would allow a reasonable estimation in cases where formal proofs do not work.

Ideally, our toolbox should implement all these strategies, and propose them as option to the user.

In the long run, we would like to introduce state-based behaviors not only in the components, but also in the abstractions used as interfaces for the components. In other words, we believe that arrival curves are not universal enough to express all the behaviors that a real system exhibits, and plan working on alternatives to arrival curves, while keeping the main ideas of this work, namely: compositionality, use of abstract interpretation.

Also, while this work Lustre allows describing state-based behavior, this is only a first step towards our initial goal, which is the estimation of energy consumption. With state-based behaviors in MPA-RTC, we can model the timed behavior of energy-aware systems, but still not the energy itself.

References

BCH+85. J-L. Bergerand, P. Caspi, N. Halbwachs, D. Pilaud, and E. Pilaud. Outline of a real time data-flow language. In *Real Time Systems Symposium*, pages 33–42, San Diego, September 1985. 3

CC77. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL'77*, Los Angeles, January 1977. 3.2

Cor08. Jérôme Cornet. *Separation of Functional and Non-Functional Aspects in Transactional Level Models of Systems-on-Chip*. PhD thesis, Grenoble Institute of Technology, april 2008. 1

DAR07. Uppsala University DARTS, IT Dept. Cats tool, 2007. `http://www.timestool.com/cats`. 1.3

Gon07. Laure Gonnord. *Acclration abstraite pour l'amlioration de la prcision en Analyse des Relations Linaires*. Thse de doctorat, Universit Joseph Fourier, Grenoble, October 2007. 5.3

HLR92. N. Halbwachs, F. Lagnier, and C. Ratel. Programming and verifying critical systems by means of the synchronous data-flow programming language LUSTRE. *IEEE Transactions on Software Engineering, Special Issue on the Specification and Analysis of Real-Time Systems*, pages 785–793, September 1992. 3.1

HLR93. N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous observers and the verification of reactive systems. In M. Nivat, C. Rattray, T. Rus, and G. Scollo, editors, *Third Int. Conf. on Algebraic Methodology and Software Technology, AMAST'93*, pages 83–96, Twente, June 1993. Workshops in Computing, Springer Verlag. 3.1, 4.1

Jea03. B. Jeannet. Dynamic partitioning in linear relation analysis. application to the verification of reactive systems. *Formal Methods in System Design*, 23(1):5–37, July 2003. 3, 3.2

KMY07. Pavel Krcal, Leonid Mokrushin, and Wang Yi. A tool for compositional analysis of timed systems by abstraction (extended abstract). In Einar Broch Johnsen, Olaf Owe, and Gerardo Schneider, editors, *Proc. of NWPT'07, the 19th Nordic Workshop on Programming Theory, O slo, Oct. 10-12*, 2007. 1.3

KT06. Simon Künzli and Lothar Thiele. Generating event traces based on arrival curves. In *13th GI/ITG Conference on Measurement, Modeling, and Evaluation of Computer and Communication Systems (MMB)*, pages 81—-98. VDE Verlag, March 2006. 1.3

Lam08. Kai Lampka. Combining computational and analytic model descriptions for evaluating embedded real-time systems. Technical report, ETHZ, 2008. Appendix for the Combest report. 1.3, 5.1, 5.2, 5.3

LAM09. Yanhong Liu, Karine Altisen, and Matthieu Moy. Granularity-based interfacing between rtc and timed automata performance models. Technical report, Verimag, Centre Équation, 38610 Gières, August 2009. 1.3, 5.3

LPY97. K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997. 1.3

Mok07. Leonid Mokrushin. Compositional analysis of timed systems by abstraction. PowerPoint Slides, 2007. 1.3

PCTT07. Linh T. X. Phan, Samarjit Chakraborty, P. S. Thiagarajan, and Lothar Thiele. Composing functional and state-based performance models for analyzing heterogeneous real-time systems. *Real-Time Systems Symposium, IEEE International*, 0:343–352, 2007. 1.3

Sam08. Ludovic Samper. *Modlisations et Analyses de Rseaux de Capteurs*. PhD thesis, INPG, Grenoble, France, April 2008. 1

TCN00. Lothar Thiele, Samarjit Chakraborty, and Martin Naedele. Real-time calculus for scheduling hard real-time systems. In *International Symposium on Circuits and Systems ISCAS 2000*, volume 4, pages 101–104, Geneva, Switzerland, March 2000. 1.2