Formal modelling framework of data acquisition modules using a synchronous approach for timing analysis

Lionel Morel *, Jean-Philippe Babau **
Belgacem Ben-Hedia ***

* CITI / INSA-Lyon
Université de Lyon
F-69621 Villeurbanne Cedex - France
Email: lionel.morel@insa-lyon.fr
** UBO, LISyC, UEB
F-29238 Brest Cedex 3, France
Email: jean-philippe.babau@univ-brest.fr
*** Procon Labs, Paris-Guyancourt, France
Email: bbhe dia@gmail.com

Abstract: In the field of process control, data acquisition modules (such as device drivers) require special care in their design, because they usually stand as bottlenecks between hardware devices and control applications. In particular timing constraints on occurrences of data are often given based on intuition and empirical experience. The work presented here intends to provide a formal model to characterize timing properties such as input data delay. As an illustration, a model of a simple data acquisition module is presented. We show how the formal model can be exploited to establish bounds of delay.

Keywords: Data acquisition, formal validation, embedded systems, synchronous modelling.

1. INTRODUCTION

Process control systems (see figure 1) are found in various embedded and industrial settings. The main role of the control application is to safely control a physical process, taking some decision in response to Input flows representing consecutive views of the physical process. Physical process independently evolves from control application. Thus, to be safe, a Control System need to satisfy critical QoS constraints on Input flows (data freshness, maximal data lost rate), Output flows (frequency) and related to Input and Output flows (realtime deadlines). Due to these QoS constraints, it is necessary to establish precisely guarantees of the different parts of such systems, particularly for QoS performances. In this work, we mainly focus on the data acquisition part and on formal evaluation of data transfer delays as QoS performances.

The most common techniques to formally evaluate maximum delays are based on tasks and message scheduling analysis (for general introductions see e.g. Klein et al. (1993); Liu and Layland (1973); Migge et al. (2003)). These techniques consider simple architectures, generally static, and may give unrealistic bounds. In order to limit this problem, many works are using techniques based on exhaustive modelling of temporal behaviors such as communicating timed automata or hybrid automata (see in particular Belarbi et al. (2004); Waszniowski and Hanzlíček (2008)). These works make it possible to obtain general and realistic information about timing (dynamic) behaviors of complex architectures. Following these approaches,
validate infinite systems. However, the language is proof-oriented and seems too powerful for evaluating timing properties on any Lustre program. We thus define a component model based on Lustre but with adequate restrictions to allow the evaluation of timing properties. Finally, Lustre is used as a pivot language and our approach is not limited to its syntax.

This paper proposes a formal framework for modelling and validating timing properties of data acquisition modules, and more particularly their software part. This framework is based on an encoding of a data acquisition module's components using the synchronous paradigm (see Benveniste et al. (2003), Caspi et al. (1987)). A component model based on synchronous semantics is first proposed. Then, a generic modelling architecture for data acquisition modules is described. Properly equipped with timing information (each data occurrence is stamped with the date of its emission), these models are then checked for propagation delay evaluation.

The rest of the paper is organized as follows. The next section presents data acquisition modules and the Lustre synchronous language. Section 3 presents the proposed synchronous component model. In section 4, an example is presented and the validation of timing properties using the proposed modelling approach is discussed. Section 5 discusses related works while section 6 concludes and gives further directions for this work.

2. CONTEXT

2.1 Data acquisition systems

Generally speaking a data acquisition system is composed of:

- A Sensor that can convert information characterizing the physical environment (temperature, speed, etc.) into a digital information;
- A HW Interface (or hardware driver) that gets the data coming from the physical device and stores them into registers that are later accessible to the device driver;
- A Data Acquisition Software (or software driver), which is a dedicated piece of the system (usually integrated in the operating system) meant as an abstraction of the hardware for easing its manipulation by the application software.

The goal of the data acquisition software part is to route information (data) between different data producers (sensors and their HW interface) to data consumers (the control application), Even if control might follow both directions, data flows always in the same direction: from producers to consumers.

For this study, a simplifying assumption is that data occurrences are never merged or decomposed (no data fusion or data extraction) inside the data acquisition software, but are organized (filtered, transformed and interpreted) both in time and space and, at the end, delivered to the control application. The data acquisition software acts as a modifier of data flow values and of data flow QoS. To conclude, we make the following assumptions on the acquisition part, considering that a flow is an infinite sequence of occurrences:

- for one output flow, including all its occurrences, there is one and only one corresponding input flow (that can contain its occurrences);
- for one output occurrence, there is one and only one corresponding input occurrence;
- different output flows (resp. occurrences) are related to different input flows (resp. occurrences).

The Lustre-based approach proposed here can cover both hardware and software parts. In the sequel, we will therefore talk about Data acquisition modules.

2.2 QoS criteria

Several factors can be used to give timing characterization of acquisition software. These generally include:

- Period (or frequency) of occurrence of an event or a data. In particular, one can express period of reaction of a sensor, and thus information arrival laws, such as those defined in Martin et al. (2005). By extension, one can also describe the activation period of the control application or some part of it.
- Delays include communication or routing delays (between sensors and applications), computation delays, communication or routing delays between the controller and the actuators. Here, we are mainly interested in the first category, i.e. communication delays.
- Jitter reflects variations of timing property, so several types of jitter exist (see Bate et al. (2003)).
- Transmission (or routing) errors that mainly lead to information loss, increase in the number and length of delays or even errors of interpretation in the control process (see Wittenmark et al. (1995)).

In the next section, a modelling framework is presented. It is based on the synchronous language paradigm, in particular, that of Lustre.

2.3 The synchronous language Lustre

The synchronous approach (see Benveniste et al. (2003)) has been developed since the 80s for the description of reactive embedded systems. It is based on the synchronous hypothesis that stipulates that 1) communications between different components of a system can be considered instantaneous from the components’ environment point of view; and 2) the reaction of a component to the arrival of new input values is instantaneous.

The language Lustre proposed by Caspi et al. (1987) is a data-flow language. Every variable or expression \( X \) represents an infinite flow of values \( X_0, \ldots, X_i, \ldots \), where \( X_i \) is the value of \( X \) at the \( i \)-th instant in time (with \( i \in [0, \infty) \)). A unique global clock serves as a time reference. Sampling of flows can be built on top of that thanks to the existence of a special abs value. The language is equational: each variable is defined by exactly one equation, and equations are organized to form components, which can in turn be used to define other components.

The formal semantics of the language allows for the application of formal verification techniques, mainly Boolean
model-checking (see Halbwachs et al. (1992)) and abstract interpretation for the verification of numerical properties (see Jeannet (2003)). Considering the type of properties we wish to validate (mainly bounds on delays), we will be mainly using the nbac tool, presented in Jeannet (2003).

Its clean and simple semantics, along with the efficiency of associated verification techniques has made of Lustre a success in highly-critical application fields. It has been used for the development of control applications in nuclear power plants, avionics, etc.

3. LUSTRE-BASED MODELLING FRAMEWORK

This section presents the framework for modelling data acquisition modules. It is based on Lustre, with several constraints on the form of components (types of inputs/outputs control events, possible manipulation of data) and on the way components are connected together.

3.1 Flows and occurrences

A data acquisition module conveys data flows from producers (or Sources) to consumers (or Sinks). To formally characterize flows, we give now some definitions and assumptions on data flows under study.

A data is a flow of occurrences, i.e. an infinite sequence of occurrences, each conveying a value. For a flow \( f \), an occurrence \( i \) is characterized by a value \( (fvi) \), the date it is generated by the producer \( (fti) \) and its production order number \( (fsi) \).

For the Source, \( fsi \) is initialized to 0 and is incremented each time a new occurrence is created. \( fvi \) is evaluated via a simulation function representing process evolution (it may be the result of an external function call), and \( fti \) corresponds to the current time when the occurrence is emitted by the Source. In our modelling setting, this time is given by the global clock.

When produced, a flow is consumed by others components, producing other internal or external flows. Except for a Source, a data flow is always computed from one and only one existing flow (no merge and no creation). Moreover, an occurrence is computed from one and only one occurrence (there is no history and no average values). So, we can consider that consumed and produced flows convey the same information, even if the values are different (filter, adaptation, interpretation). The creation time and the production order numbers for an occurrence of a flow \( g \) is the same than for an occurrence of the corresponding used flow (flow \( f \) consumed by the same component). A behavior \( g = op(f) \) implies:

\[
\begin{align*}
gvi &= \text{operation}(fvi); \\
\text{gtj} &= fti; \\
\text{gsi} &= fsi;
\end{align*}
\]

The operation has a pre-condition stating that no value \( fvk \) other than \( fvi \) should be used, and a post-condition stating that \( gvi \) is not a constant.

Each output occurrence computed from the same \( fsi \) input occurrence represents the same information. From now on, the term occurrence is used to refer to a specific \( fsi \) occurrence.

3.2 Components

Following the proposed framework, a component (figure 2) has exactly one input data flow and one output data flow. The generalization will be discussed in section 6. If complex composite components (receiving (resp. sending) data on several inputs (resp. outputs) flows) are necessary, they have to be explicitly decomposed in atomic ones.

In order to decide how to treat its input data and when to emit output data, a component has a number of events and output events (also called control signals). Note that the synchronous setting this work is based on imposes that all flows evolve on a common global clock (with possible absent values at some instants).

Inputs/Outputs The first input event isAvailable is received by the component whenever a new input value is available on the dataIn flow. The giveOut input is received when the component emits a new value on the dataOut flow. The getIn output is emitted by the component when it wants to receive a new value on the dataIn flow. outAvailable is emitted when a new value is made available on the dataOut flow by the component. Paragraph 3.3 shows how these signals can be used (and are actually sufficient) to encode a wide range of common communication mechanisms.

Beside these four signals, a component can also have a number of other input events, noted eventIn on the figure.
These can be used to model activations of transmission of a data occurrence by the component, e.g. periodic, aperiodic or sporadic.

**Behavior** An automaton describes the behavior of a component regarding new occurrences. The component contains a local memory flow (that can be a register or a memory area). This memory (named Reg in figure 2) is used to save a certain number of occurrences following a certain policy. Except for a Source or a Sink, a component does not generate new occurrence of its own and can only save, transmit or forget an occurrence:

- *save* an occurrence it receives on its input data flow in the local memory using a certain policy (FIFO, LIFO, etc);
- *transmit* an occurrence to another component, i.e. emitting it on the dataOut flow;
- *forget* an input occurrence. It can either not take into account a new occurrence it receives on its input flow, or deciding to overwrite an occurrence in its local memory.

It is important to note that, *save* and *forget* are exclusive operations, and that *transmit* is optional. In the same way, transmitting can be triggered by an *eventIn* input.

A component's behavior also includes operations on values *fvi*. As previously said, dependency between different occurrences carried on the same flow are forbidden. E.g. a component cannot use the last two occurrences received on its dataIn flow to compute the next occurrence of its dataOut flow.

Operations are abstracted by their execution time, because: 1) they are independant from occurrences’ life cycle (no deletion of occurrences); and 2) we are only interested in timing properties.

After introducing components, we now present how they can communicate with each other.

### 3.3 Communication Between Components

The control events of a component are intended to model classic communication means between components.

Concerning reception of new input occurrences, a component can be:

- *waiting*: it is indefinitely waiting for the arrival of a new occurrence;
- on *request*: it first sends a reading request, by sending a value on the *getIn* output signal. It then waits for the corresponding occurrence, identified by a value received on the *inAvailable* signal.

The rhythm with which it sends the request can be determined as periodic, sporadic or in bursts.

A component can send an occurrence using one of the following two modes:

- *pull*: it waits for a reading request from one or several other components;
- *push*: the component sends one or several occurrences. It can follow simple sending rules, or a multiple send rule (periodically, sporadically or in bursts).

For both *pull* and *push*, one can specify some treatment time for answering these requests. *Pull* and *request* modes on one side, and *push* and *waiting* on the other, are interdependent: a connection between two components is always such that the one producing data is in *pull* mode (resp. *push*) and the other in *request* (resp. *waiting*).

These different modes are easily encoded as state-machines that rely on events both internal and external to the component.

### 3.4 Architectural considerations

Following the modelling framework, a data acquisition module is organized as follows. First, it is necessarily built with exactly one *Source* component and exactly one *Sink* component that represents the control application. In between the *Sink* and *Source* components, one can use as many intermediate components as needed. So, it exists three kind of components: *Source, Intermediate* and *Sink*.

The *Source* component originates all data occurrences in the model. It can only control its output *dataOut* using its *giveOut* and *outAvailable* control signals. It has no *dataIn*, *inAvailable*, *getIn* ports.

The *Sink* component has a reversely comparable configuration. It can only control the way it reads input occurrences on its *dataIn* flow. This is done with the *getIn* and *inAvailable* control flows and has no *dataOut*, *outAvailable*, *giveOut* ports.

All components are connected so that data flows in only one direction. Also, there are no "loop" dependency between components.

Like constraints expressed earlier for components internal behavior, these last two are applied to restrain the space of possible values for the delay, and the complexity of computing this delay.

These constraints are given to enable evaluation of delays of information, and more precisely of occurrences. The framework enforces architectural constraints and occurrences life cycle constraints.

Some architectural constraints are ensured by construction and others by checking properties on the model. Providing an editor tool to build correct-by-construction data acquisition models, and checking adequacy with the framework are both part of future work.

Concerning behavioral constraints, an occurrence is generated once by its *Source* and we assume that it is possible to follow its life cycle in the system (tracing data through copy and communication). To provide timing delays characteristics, we assume that after its creation, it exists a moment when an occurrence disappears (no more copy in the system). If not, the delay would be infinite for the occurrence. Checking this bounded life-time property is complex. At the moment, if exhaustive simulation provides delays, we can conclude *a posteriori* that the system is correct under delay evaluation. If not, we cannot conclude because of possible combinatorial explosion of simulation.

These constraints are sufficient for computing delay of information in the module we build using these components. However they are not necessary, and part of future work
(see section 6) consists in relaxing them to fit more realistic applications.

3.5 Formal Definitions

The following definitions follow the synchronous semantics of Lustre. It is however simplified with the considerations previously mentioned.

All variables manipulated by components are flows of data.

**Definition 1. (Flow).** A data-flow \( X \) is an infinite sequence of values \( x_0, x_1, ..., x_i, ... \) where \( x_i \) is the value taken by \( X \) at the \( i \)-th instant of execution.

**Components** More precisely, as we model delay, a time-stamp is associated to every data that is put into the system. These timestamps are values emitted by the global clock of the program.

**Definition 2. (Time-stamped flows).** A stamped flow \( (X, T, S) \) is a data-flow whose values are tuples of the form \( (v_i, t_i, s_i) \). \( v_i \) is the value of \( X \) at the \( i \)-th instant of execution. \( s_i \) is the production order number and acts as an identifier of the corresponding data occurrence. Finally, \( t_i \) is the time at which this occurrence has been initially produced in the program.

**Definition 3. (Component).** A component in a data acquisition module is a tuple:

\[
C = (\text{dataIn}, \text{dataOut}, \text{ctrlIn}, \text{ctrlOut}, \text{localCtrl}, A, \text{Reg})
\]

where:

- \( \text{dataIn} \) (resp. \( \text{dataOut} \)) is the input (resp. output) time-stamped flows;
- \( \text{ctrlIn} = \{\text{inAvailable}, \text{giveOut}, \text{eventIn}\} \) and \( \text{ctrlOut} = \{\text{outAvailable}, \text{getIn}\} \) are input and output control flows;
- \( \text{localCtrl} = \text{regControl} \cup \text{processingControl} \) is a set of local control events (see below);
- \( \text{Reg} \) is a memory block, that is controlled with the \( \text{regControl} \) events;
- \( A \) is a mealy-machine defined as a tuple \((S, S_0, \Sigma, \Gamma, T, \mathcal{G})\), where:
  - \( S \) is the set of state and \( S_0 \in S \) is a unique initial state.
  - \( T = S \times \Sigma \rightarrow S \) is the transition function.
  - \( \mathcal{G} = S \times \Sigma \rightarrow \Gamma \) is an output function.
- \( \Sigma = \text{CtrlOut} \cup \text{processingControl} \). This means that the control is triggered by both external control events and local control events. \( \text{processingControl} \) are essentially meant to describe treatment times, i.e. they are defined as the timeout of some internal counter based on the global clock. Then \( \text{eventIn} \) signals can be used e.g. to define activation periods.

\( \Gamma = \text{CtrlOut} \cup \text{regControl} \). This means that the automaton of a component controls exactly two things. First, through \( \text{regControl} = \{\text{save, forget, transmit}\} \) it controls the \( \text{Reg} \) component, i.e. it has an ability to decide whether a data occurrence is saved, forgotten or transmitted. It also controls the \( \text{CtrlOut} \) events emitted towards other component. These serve for asking for new input values, signalling new output values or propagating local control decisions (e.g. timeout of the activation period).

\[
\text{Reg(dataIn, save, transmit, forget)} \rightarrow (\text{dataOut})
\]

\[
\text{var : mem;}
\]

\[
\text{if(save)\{mem = dataIn;\}}
\]

\[
\text{if(transmit)\{dataOut = mem;\}}
\]

\[
\text{if(forget)\{dataOut = null;\}}
\]

Fig. 3. The Reg memory component.

The \( \text{Reg} \) memory (see figure 3) is a simple combinatorial component (it includes no state of its own) which can be described by the following semi-algorithm:

- The forget and save actions are not compatible. Here, we decide that this constraint (exclusion of forget and save) should be verified on the control automaton of the component.

Note also that the time-stamp associate to every occurrence of every data flow in the model is never modified by a component.

The next paragraph describes how two components can be composed. This composition can be used recursively to build more complex data acquisition models.

**Composition** Two components can be composed by connecting the one’s input data flow output to the other’s output data flow, along with connecting control signals from each other together.

The composition is intrinsically synchronous. First, whenever a component receives one new input value, the corresponding output value is instantly available from the environment point of view. This means that the reaction time can be neglected. Second, whenever a new output value is produced by a component, it is immediately available to any component that is declared to use it as input (synchronous broadcast). This means that communication is considered instantaneous.

**Definition 4. (Synchronous Composition).** Given two components \((i \in [1, 2])\)

\[
C_i = (\text{dataIn}_i, \text{dataOut}_i, \text{ctrlIn}_i, \text{ctrlOut}_i, \text{localCtrl}_i, A_i, \text{Reg}_i)
\]

The composition \(C_1 \times C_2\) is defined as a new component \(C_{12}\) such that:

- \(\text{dataIn}_{12} = \text{dataIn}_1\)
- \(\text{dataIn}_{12} = \text{dataIn}_2\)
- \(\text{ctrlIn}_{12} = \text{ctrlIn}_1\)
- \(\text{ctrlOut}_{12} = \text{ctrlOut}_2\)
- \(\text{localCtrl}_{12} = \text{localCtrl}_1 \cup \text{localCtrl}_2\)
- \(A_{12} = A_1 \parallel A_2\) (synchronous product)
- \(\text{Reg}_{12} = \text{Reg}_{12} \cup \text{Reg}_{12}\) (both memories are updated independently)

We can now build complex data acquisition models. As shown in definition 2, each data flow is tagged with timestamps. The following paragraph shows how one can use these timestamps to compute delays of data occurrences.

**Computing delays** The form of the data acquisition module under study is given in figure 4.

Every data occurrence that traverses the program is a tuple \((v, t, s)\), i.e. a data value tagged with a time-stamp.
4.1 Model

A number of components are connected, with data flowing from the Source (left), through the Inter1 and Inter2 towards a Sink component not drawn.

We are interested in the delay of information between the input of the Source, i.e., when an occurrence is actually entering the data acquisition system, and the output of Inter2.

All three components contain a local Reg memory capable of holding exactly one occurrence. They are also driven by some global Activation Controller, used to model activation periods. To allow that, each component has one extra input signal act.

We describe below several configurations of modes for the three components we consider.

Observing transmission delays As it was described in section 3.5, every data flow in the model is a tuple \((v, t, s)\) of value, time of emission, identifier of occurrence. In order to observe data delay, we add an observer component. This takes as input the global time reference and the output data from the Sink component. It computes the delay exactly as described in 3.5.

4.2 Validation

As mentioned earlier, Lustre comes with a set of formal validation tools\(^1\). These include simulators, debuggers, Boolean model-checkers and abstract interpretation tools.

Several configurations have been implemented very simply from the original model, by changing the modes of each components and the associated frequencies of emission/reception.

push/waiting and push/waiting A first configuration is with the three components in push mode for their outputs and Inter1 and Inter2 in waiting mode for their inputs. Source, Inter1 are both configured to write periodically their output data with a respective period of \(fS, fI1\) and \(fI2\) defined by the Activation Controller.

(1) A trivial case for these periods \((fS = fI1 = fI2)\) gives a delay of exactly 0. The three components are perfectly synchronized and no data occurrence is lost along the data acquisition module.

(2) If we fasten the Source (by changing \(fS\) to, e.g., 1, meaning that the component sends a new data at each instant), the delay is still kept to 0, but now data is lost on the way. This means that data does not always go from Source to Inter2, and that when it actually does, it does so instantly. This situation (with \(fI1 = fI2 = 5\)) is depicted by the chronograph of fig. 6. dataOutI2_s is the number of the occurrence, and we can clearly see that, e.g., occurrences 16, 17, 18 and 19 are lost.

By varying these different factors, interesting situations can be observed.

(4) If the Source and Inter1 component have the same emission rate, but are out of phase, the delay is constant (because of identical emission rates) but different from 0.

pull/request and push/waiting In a second configuration, modes are changed between the Source and Inter1 components only. This communication is now in pull/request mode. Interesting parameters are then \(fI1\_in, fI1\_out\) and \(fI2\_out\), respectively the frequency at which Inter1 requests a new input from Source, the frequency at which Inter1 produces a new output, and the frequency at which Inter2 produces a new output occurrence.

(5) Using this configuration, figures 7 and 8 show delay results for \(fI1\_out=5\). \(fI2\_out\) varies from 8 to 9 between the two charts, while \(fI1\_in\) varies on the horizontal axis of each chart. Each dot represents one simulation. An interesting situation arises when \(fI1\_in = fI2\_out\). In that case, we can see that the interaction between the request and pull modes is not instantaneous. The Source component actually takes one instant to prepare its transmission in response to the request from Inter1. By that time, the emission order from Inter1 is already passed and the occurrence transmitted by Inter1 will wait for another period of Inter2 to come out of it.

Other illustrating configurations include computation times. Whenever a component receives a new input occurrence, it takes a finite time to process it. Different cases can be designed for taking this time into account.

(6) Suppose an action is reactivated by the component for each new data occurrence it receives (or requests). If the computation time of this action is greater than the output sending or input reading period (depending on the mode the component is configured with), then simulation shows no data transmission. The delay can be supposed to be infinite.

\(^1\) see http://www-verimag.imag.fr/SYNCHRONE/
Fig. 5. An example of data acquisition module modelled in Lustre. The dashed arrows represent observed information. \( \text{ActS}, \text{ActI1} \) and \( \text{ActI2} \) are complementary inputs events to components.

Fig. 6. Example (2): no delay is observed, but some occurrences are lost.

Once possible values are identified by simulation, the abstract interpretation tool Nbac by Jeannet (2003) can be used for proving the uniqueness of these values. Example (6) is proved correct by Nbac. This can also be used to generate counter example traces (in our case, behaviors that lead to values of delay that were not foreseen by simulation), as described in Gaucher et al. (2003).

5. RELATED WORKS

A large number of works are related to the performance evaluation of device drivers used in large information storage equipments. These approaches are generally based on execution observation techniques (temporal trace analysis) as in Cota-Robles and Held (1999), or on simulation of models as in Bate et al. (2003). The main objective of these works is to evaluate delay. But the techniques used are based on statistical or probabilistic analysis. The obtained results are only average delays. Therefore the maximum delay can not be evaluated, which is crucial for real time systems dedicated to process control.

The practical approach we adopt here has already been applied in Ben Hédia et al. (2005), but using an asynchronous modelling framework. The main difference is that here we propose a formal model of data acquisition systems that confirms simulation results.

Modelling asynchronous systems using a synchronous approach is not a new idea (see Halbwachs and Baghdadi (2002)). More recent work, including Jahier et al. (2007); Jahier et al. (2009) have pushed this idea further, by proposing libraries of operating systems asynchronous components (semaphores, schedulers, threads, etc.). Where these works concentrate on capabilities of Lustre for modelling complex asynchronous systems, we try to show how Lustre can be used to model systems on which we can verify delay properties. These two approaches are complementary.

6. CONCLUSION AND FUTURE WORKS

This work provides a formal and precise Lustre-based framework for analysis of timing properties in data acquisition module. Then formal verification tools may be used to evaluate and validate timing properties, especially propagation delay constraints.

Several complementary works can be foreseen from here. First, we are working on automating simulation and making the generation of possible delays more efficient. The automatic verification also shows limits, that should be explored, independently from the model.

A second continuation to this work consists in relaxing constraints put on components and considered architectures. The form of data that are exchanged (value, date, identifier) directly gives us the possibility to relax constraints on modifications of occurrences. One could allow values to be changed, while keeping a trace of the identifiers of occurrences involved in the computation of a given occurrence. One then needs to study how this impacts the way to compute the new date (and thus delay) of such occurrences.

This techniques lets also foresee relaxing constraints on components mixing different input flows to form one output flow. This opens the way to more complex systems.

Once this model is relaxed, the approach can be applied to more general applications. A first example, a more realistic mode of a device driver, inspired from existing RTOS-based implementations. Other interesting applications include synchronous models of real-time operating services, such as proposed, for example, by Jahier et al. (2007) or Jahier et al. (2009). In the same way, we are working on providing a link with well-established design languages, such as AADL and its behavioral specification Annex, to be translated into the proposed component model.
Fig. 7. $f_{I_1\text{ out}} = 5$ and $f_{I_2\text{ out}} = 8$.

Among other possible future works, one can apply the same modelling and verification techniques on other properties for real-time systems: jitter, message loss. Finally, this study should be completed with simulation based on implementations of models on RTOS-based platforms. This will allow us to reinforce modelling assumptions.

REFERENCES


Hallwachs, N. and Baghdadi, S. (2002). Synchronous modelling of asynchronous systems. In EMSOFT’02:


