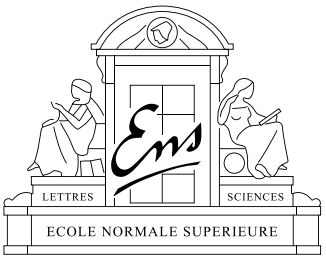
Research Internship Report

with the Synchron team at the Verimag lab, during June and July 2010,

supervised by Christophe Rippert, Karin Altisen and Kévin Marquet.

A formal approach to the development of system services in embedded systems: from model to implementation.

Pablo Rauzy



Computer Science Department — École Normale Supérieure

September 19, 2010

Abstract

Our goal is to enable preemptive multitasking in critical embedded systems programmed with the formally defined Lustre programming language, in order to get rid of the constraints imposed by static scheduling, which is still in use at this day. We aim to do so by introducing a tiny system layer between the hardware and the software, which would also be written as much as possible in Lustre to allow global validation of the system using formal verification methods.

Contents

Thanks

First, I would like to thank Christophe Rippert, Karin Altisen and Kévin Marquet, for providing such a great internship subject, advising me so well and being available during my internship and even after.

Second, I'd like to give special thanks to Damien Vergnaud, who took care of many internship-related administrative tasks for us.

During my internship I met with Florent Capelli and Martin Bodin, students at the ENS Lyon who were also doing an internship at Verimag and with who we sympathized, I'd like to thank them for all the good moments and funny discussion we had. I'd like to make the same kind of thank to Nicolas Berthier, PhD student at Verimag and also LaTeX-guru. I'd also want to thank Pascal Raymond for being this much available for me and all of my "how"s and "why"s about Lustre language and tools.

Finally I'd like to thank every person I met at Verimag, either at lunch, during the coffee break, or even around the *Libération* or *Le Canard Enchaîné* crosswords... I really enjoyed meeting them and discussing with all of them.

1 Introduction

1.1 The Synchronous team at Verimag

The "Synchronous" team of the Verimag laboratory proposed more than 20 years ago the formally defined synchronous language Lustre, for the development of critical control software. Since then, Lustre has evolved and is now used in the industry with-in the SCADE tool, provided by Esterel Technologies. Notable users are Airbus, Schneider Electric, and Eurocopter, for instance.

During the last decade, the activities of the group have been extended outside the strict scope of synchronous languages and control systems, to cover most aspects of embedded system design, implementation and validation. This work has mostly been done with the development of Lustre.

1.2 Critical systems and synchronous languages

Software development targeting critical embedded systems requires the use of reliable methods, based on formal models allowing automatic validation of programs.

In order to reach maximum speed and to avoid having non-verified parts, these software usually runs on the bare metal, without any layer between the hardware and the software.

The followings are some of the properties programming languages used in critical system development may have.

To be *formally defined* for a programming language means that its semantic is defined and imposed by the definition of the language, this ensure the possibility of reasoning about the execution of programs. Example of formally defined languages are SPARK and Lustre.

A programming language is said *declarative* if it tries to express the logic of a computation without explicitly describing its control flow but rather expressing correlations between states of the system. The attempt is to minimize or eliminate side effects by describing what the program should accomplish, rather than describing how to go about accomplishing it. Example of declarative languages are Prolog, CSS, XSLT and Lustre.

A *synchronous* language is one of which the execution of the program is punctuated by a clock: every variables values are computed "simultaneously" at each clock tick. This is an advantage for programming reactive systems, which are

often interrupted and must respond quickly. Example of synchronous languages are Esterel, Signal and Lustre.

The *dataflow* approach models programs as directed graphs of the data flowing between operations. Example of dataflow languages are Simulink, Lucid, Verilog, Max/Msp and Lustre.

In Lustre, the control flow is obtained by solving constraints imposed by equations expressing transitions between the system states (declarative), these transitions are computed "simultaneously" for all variables (synchronous), and programs can be seen as automata that we can study in order to make verifications about certain kind of properties (dataflow).

> "Lustre is a formally defined, declarative, and synchronous dataflow programming language, for programming real-time systems."

But even with those properties, Lustre currently has some limitations that we would like to overcome...

1.3 Issues and goals

With the development of technologies, electronic components and embedded systems spread everywhere. This create the need to run multiple programs on the same chip, in order to avoid multiplication of physical components and the growing complexity of their interconnection.

As of now, Lustre allow concurrent programming in an entirely deterministic way, using a static scheduling of tasks computed at compile time. This implies several limitations: for instance this way of multitasking is not preemptive which means that it's not able to handle sporadic tasks such as unpredictable events or emergency procedures whose execution time is unknown during the system conception. Furthermore, these kind of features are commonly needed in real-time systems.

Those constatations advocate for the introduction of some minimalist system services to handle the implementation of additional features, such as dynamic preemptive scheduling, while still being able to run verification and guarantee the semantics and the validity of programs.

All this led to the purpose of my intership: the design and implementation of the said "tiny system layer".

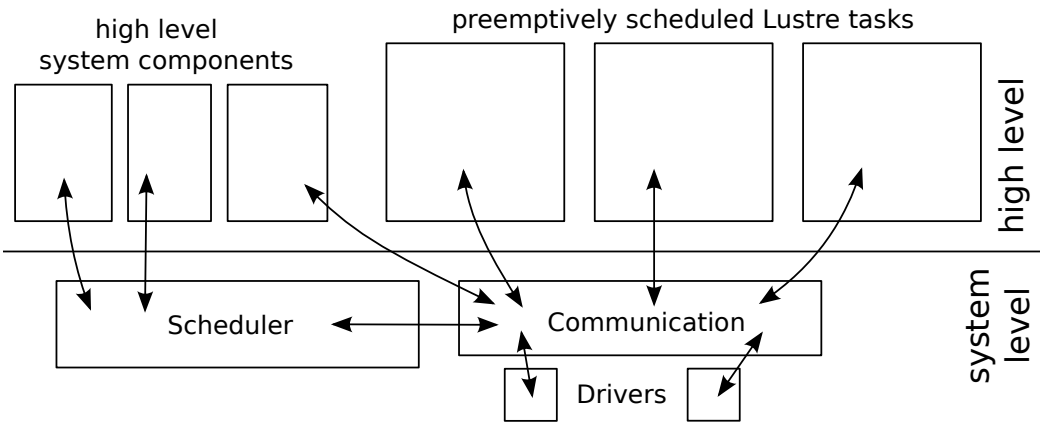


Figure 1: Schematic views of what the overall system should look like.

2 Propositions

Before going any further, lets study Lustre from a closer point of view.

2.1 The Lustre programming language

One of the important notions in Lustre is the time. Any variable and expression denotes a *flow*, which is a pair of a possibly infinite sequence of values of a given type, and a clock representing a sequence of times.

Any program, or piece of program, written in Lustre as a cyclic behavior, and that cycle defines a sequence of times which is called the *basic clock* of the program.

Programs are organized in **`node`**s which contain equations (in the mathematical sense). These equations contain variables of either primitive types (bool, integer, real and tuples) or user-defined types, and use a set of primitive operators (boolean: **`and`**, **`or`**, **`not`**, **`xor`** ; relational: **`=`**, **`<`**, **`<=`**, **`>`**, **`>=`** ; arithmetic: **`+`**, **`-`**, **`*`**, **`/`**, **`div`**, **`mod`** ; conditional: **`if then else`**).

In addition to those, there are four other operators:

- **`pre`** which act as a memory. Think of a physical register but for any type of value. The sequence of values of the **`pre E`** flow is $(nil, e_1, e_2, ...)$ when $(e_1, e_2, ...)$ is the sequence of values of the flow **`E`**.
- **`->`** ("followed by") is a binary operator that given two flows **`E`** $(e_1, e_2, ...)$ and **`F`** $(f_1, f_2, ...)$ gives a flow **`E -> F`** $(e_1, f_2, f_3, ...)$.

- **when** is used to create slower clock based on a boolean flow. **E when B** is the flow whose clock tick when **B** is $true$ and whose values are those of **E** at these times.
- **current** is used to get the current value (the value computed at the last clock tick of the flow) of a flow with a slower clock.

This table clarify the behavior of the last two operators:

E	1	2	3	4	5	6	7	8
B	$false$	$false$	$true$	$false$	$true$	$false$	$false$	$true$
A = E when B			3		5			8
current A	nil	nil	3	3	5	5	5	8

The code 1 is an example of Lustre code.

Code 1:

```
node counter (init, incr: int; reset: bool)
  returns (n: int);
let
  n = init -> if reset then init else pre(n) + incr;
tel

node dummy (restart: bool)
  returns (d: int);
var a: int;
let
  a = 0 -> pre(a) + 1;
  d = counter(pre(a), d, restart);
tel
```

In addition to that, Lustre also has support for arrays and recursive nodes as syntactic sugar. The compiler expands arrays into as many variables as they have elements and unfolds recursive nodes into regular nodes. As a consequence, arrays size and index must be known at compile time, as well as the parameters controlling the recursion.

Arrays are typed by $type\hat{}size$, and recursion is controlled by the **with** operator, which acts like **if** but the condition must be computable at compile time. An example of recursive node is presented in code 2.

Code 2:

```
node get_array_item (const size: int; array: int^size; index: int)
  returns (item: int);
let
  item = if index = 0 then array[0]
         else with size = 1 then -1
```

```
            else get_array_item(size - 1, array[1 .. size - 1],
                                index - 1);
tel
```

Besides equations, Lustre programs may contain *assertions* which are boolean expressions that should always be $true$. Their primary use is optimization informations for the compiler, but they are also used by verification tools.

Lustre compilation process is certified to be valid, which means that the development process of the compiler obeys certain rules.

Lustre compiler generate a single finite state automata, that it encodes in the C programming language for efficiency. For the same reason the generated code runs on bare metal, without any layer in between.

The general form of the compiled code is as follow :

```
forever do:
  read inputs
  computations
  update memories
  emit outputs
```

Where `computations` is a transition to the next state in the automata, which means that a step of each equations of the program is solved.

2.2 The need for a system layer and propositions

With the development of technologies, electronic components and embedded systems spread everywhere. Moreover, everywhere real-time systems are in use, we need, or at least want, them to do increasingly many things. It is not uncommon for a newly designed car to have more than a hundred processing units in the basic model.

This obviously create the need to run multiple programs on the same chip, in order to avoid multiplication of physical components and the growing complexity of their interconnection.

Currently, Lustre allow concurrent programming in an entirely deterministic way, using a *static scheduling* of tasks at compile time, which is moreover done manually.

This causes several limitations, this way of multitasking can't handle events or emergency procedures. A real-time system could need to able to trigger emergency break at any time for instance. This kind of need is very common as we can imagine in critical embedded systems, but it is still needed to have a

program running on a separate chip dedicated to this task. In our quest to run as many thing as possible on a single chip this is a strong curb.

In order to solve this issue, we need dynamic and preemptive multitasking.

Preemption, by opposition to *cooperation*, is the act of temporarily interrupting a task without requiring its cooperation, and with the intention of resuming the task at a later time. Such a change is known as a *context switch*.

Preemptive scheduling thus require an external agent to make the context switches when necessary, hence we need a low-level layer between our programs and the hardware responsible for that.

Once the idea is here, the first thought is to just use the Linux kernel as many embedded devices do. The problem is that in our context, critical real-time system, we need reliable, formally validated software.

Plus, we have one more specificity here: classic scheduler as we can found in the Linux kernel are designed for endless process that just need to share CPU time. But what we have is *periodic processes*, this means they have to be run every x seconds or microseconds, and that their execution time is known to be less than their period.

To summarize, our goal at the scheduling level is to be able to do dynamic preemptive scheduling as shown in figure 2 (page 10). Also, this implies the question of how to manage communication between processes while scheduling knowing that we must preserve the semantics of Lustre programs.

The proposition is then: let's make a low-level and as small as possible system layer which use verified Lustre components as much as possible.

3 Designing a preemptive kernel

3.1 Choosing the right platform

Before starting the development, the first step is to choose the platform to be used. Since the project is to run a system on bare metal, there is two options for the development phase. These two options use the same approach for the organizations of the code, this means that porting the code from one to another won't be something complicated.

First option, use some external hardware. Either a "real" computer or a microchip like a Arduino or a Lego NXT brick. The advantage of this is that the development takes place in real situation, and components like motors and sen-

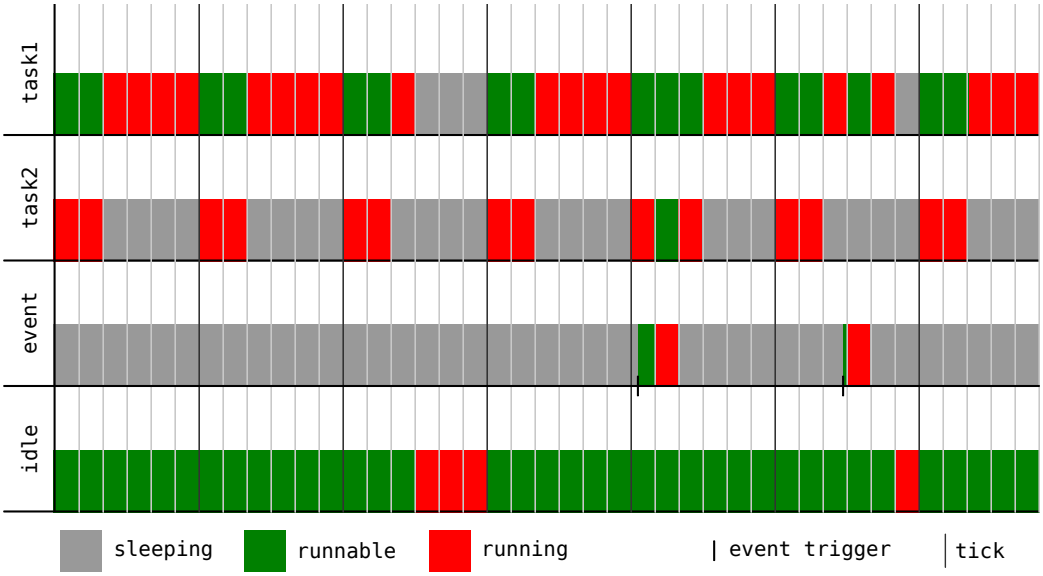


Figure 2: *task1* is long and have a low priority, its period is 3 ; *task2* is quick and have a higher priority, its period is 1 ; *event* is very quick and have the higher priority so when it is triggered the scheduler launch it at the next tick.

sors can easily be added to the system so the tests can go further than simple printing on a screen.

Second option, use a virtual platform. Either a processor emulator or a virtualization software like VirtualBox. The advantage here is that the deployment on the platform for testing is really quick and that we can connect it a debugger, which is almost vital to the development process, through a virtual serial port.

The choice has been to use both options for their positive sides: the development will be done in VirtualBox until it's sufficiently advanced to be worth porting and continued on external hardware, the Lego NXT.

3.2 Lustre process

We first need to define what is a process and more specifically a Lustre process, since this is what we want to schedule. A process consist essentially of an identifier, a name (convenient for debugging), a period, a priority, a current state, a container for its context, and a pointer to the function representing its tasks (written in Lustre). The C structure representing a process is viewable in code 3.

Code 3:

```
enum process_state_t {
  PROC_STATE_NULL, PROC_STATE_SLEEPING,
  PROC_STATE_RUNNABLE, PROC_STATE_RUNNING
};

struct kernel_context_t {
  int esp; /* stack pointer */
  unsigned int stack[STACK_SIZE];
};

struct process_t {
  int pid;
  char name[PROC_NAME_MAXLEN];
  time_t period;
  int priority;
  process_state_t state;
  kernel_context_t kc;
  void (*task_function)(void);
};
```

Processes context data are in a separate structure because we need to pass pointer to them to the function which makes the context switch.

The `task_function` is a pointer to a C function in the process glue which for ever repeat the cycle "read-inputs, compute, update-variables, write-outputs", where "compute, update-variables" is a call to the step function generated by the Lustre compilation.

3.3 Context switching

The very first step that need to be acheived is to switch context. As said before this is an operation performed by the scheduler which consist of saving the state of the currently running process and restoring the state of another process to be run.

A process "state" or *context* consists of the values in the registers and flags of the processor, including the stack pointer and the instructions counter.

This operation is very low level so it has to be coded in assembly language. The principle is quite simple: push the state on the stack and save the stack pointer, then restore the stack pointer of the process to be run and pop its state from the stack. When this function return, the return address is the value of the restored instructions counter so the process to be run restart immediately as if it had never been interrupted.

A very simple round robin scheduler was used in order to test the context

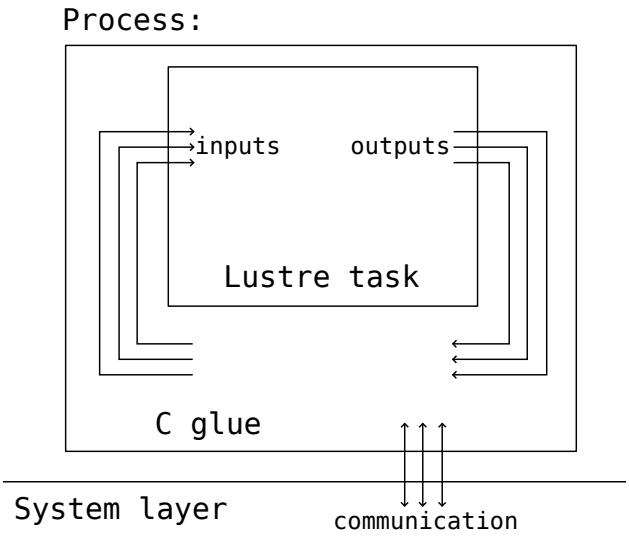


Figure 3: Schema of a Lustre process. The glue is responsible for communication with the system (scheduling, inter-process communication...)

switch: it alternatively runs two tasks and is called from the function handling the interruption used by the timer.

Scheduling is a more complex activity than mere context switching, and now that the context switch is working, it's time to start the development of the scheduler.

3.4 Preemptive scheduling of periodic tasks

We need to preemptively schedule a set of tasks which each have a period and a priority. To achieve that our scheduler is triggered at a high frequency for a very short time in which it check if some processes start a new period, if so it mark them as runnable and add them into a max-heap which uses process priorities as keys. Then it compares the priority of the currently running process with the process pointed to by the root of the runnables heap. If the later is higher then the scheduler switch the context.

When a task has finished a step (or cycle) it calls the scheduler to be put in the sleepings list and removed from the runnables heap, until a new period starts.

It is required to have an idle tasks with the lowest priority and which is always runnable.

3.5 Sporadic tasks

Implementation of sporadic tasks is based on processor interruptions, and the handler of the interruption call the scheduler. Then, there is two choices for what the scheduler should do.

The scheduler could either be made to immediately context switch to the sporadic tasks, or to manage the sporadic task as a regular task by adding it to the runnable heap, except with a higher priority, so it's the task selected at the next scheduler round. Considering that the scheduler is triggered highly frequently enough to see the launch of the sporadic task as almost instantaneous is reasonable. Plus doing it this way has the advantage to follow the same model as periodic tasks, which could be useful to make verification, or to implement communication between those two types of tasks.

At the level of concrete implementation, in addition to the new interrupt handler, sporadic tasks only required to add a `trigger` field which correspond to a letter on the keyboard in the `process_t` structure to be able to have multiple event in VirtualBox since we don't have access to any other input peripheral than the keyboard.

Figure 2 (page 10) is a chronogram of an example of execution with two periodic tasks and one sporadic tasks.

3.6 Inter-process communications

Inter-process communications is very important in critical real-time systems and should not be hazardous. The synchronous paradigm of Lustre programs oblige us to take a special care of this aspect in our system layer.

3.6.1 Issues and requirements

The requirements are imposed by Lustre working loop: "read-inputs, compute, update-variables, write-outputs", shared variables used for communications must not change during a step of the loop in order to preserve the synchronous and equationnal semantic of Lustre program's code. This is simple when there's only one Lustre program running, but when multiple programs run in different process that are preemptively scheduled, it become far less trivial. However Paul Caspi, Norman Scaife, Christos Sofronis and Stavros Tripakis worked on this particular problem and released a paper titled "*Semantics-Preserving Multi-Task*

Implementation of Synchronous Programs"[1] discussing the subject and proposing a solution.

3.6.2 The Dynamic Buffer Protocol

DBP (Dynamic Buffer Protocol) is an inter-task communication protocol that is semantics-preserving and memory-optimal. DBP guarantees semantical preservation under all possible triggering patterns of the synchronous program: thus it is applicable not only to time-triggered, but also event-triggered applications. It is based on the use of intermediate buffers and manipulations of write-to/read-from pointers to these buffers: these manipulations happen upon arrivals, rather than executions of tasks, which is a distinguishing feature of DBP. Moreover, DBP is memory-optimal in the sense that it uses as few buffers as needed, for any given triggering pattern. In the worst case, DBP requires at most N + 2 buffers for each writer, where N is the number of readers for this writer.

The way is has been implemented here is a communication module at the system level, which is just a storage of double buffers with accessors. Buffers memory is allocated statically on the stack since everything about shared variables is known at compile time, in particular their type, and thus their size. A shared variable is written by a task and read by one or many other tasks, each reader-task have it's own buffer for the shared variable. It's the glue of Lustre process which use the accessors of the communication module and its task inputs and outputs to make processes communicate according to the DBP.

Then it's all a question of when to read, write and swap buffers. This depends on how the variable is shared. The shared variable can be read by a lower priority task or by a higher one, and in the first case, the variable can be read through a unit-delay (register) or directly, in the latter case, there's always a unit-delay.

The figure 5 (page 15) is an example of tasks running and communicating through shared variable as in the graph of figure 4 (page 15), which show all three possible cases.

The writer task, w, maintain a double buffer **`B[0,1]`** with **`current`** and **`previous`** pointers. Initially, **`previous = current = 0`**. When it runs, w writes to **`B[current]`**. When it becomes runnable the two pointers are swapped (moment **b** in figure 5).

It is important to note that this is done when the task becomes runnable, not running, which is not the same for instance if a task with higher priority is still running or becomes runnable at the same time. This is why the DBP can't be

[1] ACM Transactions on Embedded Computing Systems, Vol. V, No. N, July 2007

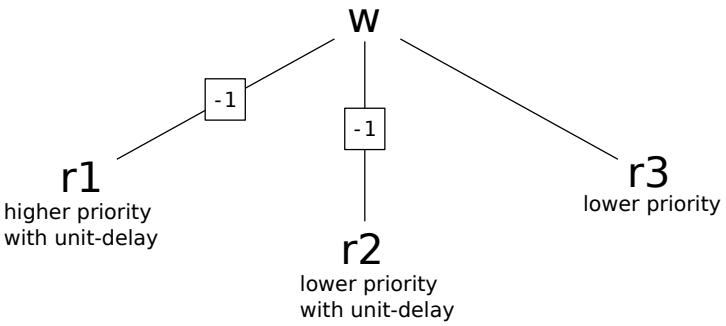


Figure 4: w is the writer task ; $r1$ is a reader task with a higher priority than w and a unit-delay ; $r2$ is a reader task with a lower priority than w and a unit-delay ; $r3$ is a reader task with a lower priority than w.

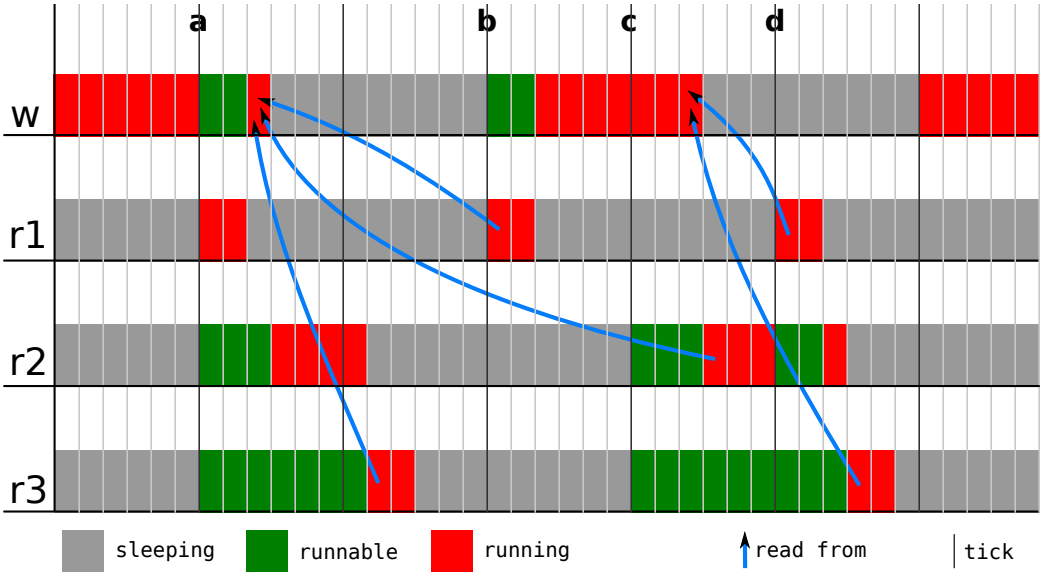


Figure 5: An example chronogram with DBP.

managed by the tasks alone and has to be managed at system level.

The reader task $r1$, which has a higher priority than w, maintain a pointer **p1**, which is set to **previous** when $r1$ becomes runnable (moment **b** and **d** in figure 5), and read from **B[p1]**. Since the inputs and outputs of Lustre tasks are used to pass shared variables around, this is done silently and the pointer is implicit: when $r1$ becomes runnable, **B[previous]** is copied as one of $r1$ inputs.

For the reader tasks $r2$ and $r3$, which have lower priorities than w, the protocol says that the w tasks maintains their buffer (that's the N of the N + 2 buffers announced in the description of the protocol), but the implementation uses the inputs and outputs of Lustre tasks so those additional buffers aren't needed. In fact they exists but they are distributed in the tasks, as input variables: when $r2$ becomes runnable (moment **b** in figure 5), **B[previous]** is copied as one of its inputs, and when $r3$ becomes runnable (moment **a** and **c** in figure 5), the same happens with **B[current]**.

Using the Dynamic Buffer Protocol ensure that we don't change the semantics of the tasks when adding inter-process communication to preemptively scheduled tasks. This, of course, is important for verification purpose.

4 Writing system components in Lustre

4.1 Which components?

Ideally we would like to have everything coded in Lustre, but it won't be possible for several reasons. One of which is that the call to the context switching function doesn't return by definition, so it can't have a valid semantic in Lustre. A not ending call can't fit in Lustre "read-inputs, compute, update-variables, write-outputs" loop.

So what we want is to have as much of the scheduler logic as possible in Lustre, and keep the very low-level basic operations in C. Apart from context switching, the scheduler has to select the next running task, and to maintain the periodic tasks clock. These are the two parts of the scheduler that we are going to write in Lustre.

4.2 Periodic tasks clock in Lustre

What we want here is a Lustre node that takes an array of periods as input and output an array of boolean indicating if a period begins or not.

This is a trivial task in classical programming languages, but the absence of real arrays and the dataflow approach in Lustre force us to think differently. The end result (viewable in code 4) is quite simple but not straightforward for someone who is not familiar with the dataflow paradigms and the Lustre arrays, remember that they are unfolded at compile time.

Code 4:

```
node periods_clock (periods: int^n) returns (activation: bool^n);
var deadlines: int^n;
let
  deadlines = 0^n -> if pre(deadlines) = 0^n
                     then periods - 1^n
                     else pre(deadlines) - 1^n;
  activation = (deadlines = 0^n);
tel
```

Here is how it works: **`activation[i]`** is $true$ only if **`deadlines[i]`**'s value is 0. **`deadlines[i]`** is a flow which initial value is 0 (so when the program start all the periods starts), and then is updated conditionally at each step. If the value of **`deadlines[i]`** at the previous step was 0, then a new period started at the previous step, so it's value is **`periods[i] - 1`**. Else the value of **`deadlines[i]`** is decremented by 1. This is done synchronously for all tasks.

4.3 Selection of the next running task in Lustre

The low-level implementation of this feature is quite simple: it simply uses a max-heap of runnable processes indexed by their priorities. In Lustre we can't do it this way, once again the dataflow approach put things into a new perspective.

So we need to write the selection of the runnable process with the higher priority as a system of equations.

We have access, as inputs, to the activation boolean array computed in code 4, to the priorities array and a boolean array indicating for each task if it has finished since last step.

Code 5:

```
node select_task (priorities: int^n; activation, finished: bool^n)
  returns (cur, next: int);
var
  runnable: bool^n;
let
  runnable = activation or (pre(runnable) and not finished);
  cur = pre(next);
  next = get_next(runnable, priorities);
tel
```

A process is runnable if it has just been activated or if it was runnable at the previous step and didn't finish in between (code 5). Then knowing priorities and runnable tasks we can compute which process we have to run next (code 6). The currently running process is also returned so the low-level avoid context switching if not needed (which is most of the time).

Code 6:

```
node get_next (runnable: bool^n; priorities: int^n)
  returns (next: int);
var
  max: int^(n+1);
  highest: int;
let
  max[0] = -1;
  max[1 .. n] =
    if runnable[0 .. n - 1] and
       (priorities[0 .. n - 1] > max[0 .. n - 1])
    then priorities[0 .. n - 1]
    else max[0 .. n - 1];
  highest = max[n];
  next = get_last_true(n, runnable and
                          (priorities = highest^n));
tel
```

The **get_next** node in code 6 make a clever use of the way Lustre unfold arrays to simulate an operation equivalent to the folding of a list on an array. Once unfolded, the code says that for $i = 0$ to $n - 1$, **max[i+1]** value is the priority of process i if it is runnable and its priority is greater than **max[i]**. In every other case, the value of **max[i+1]** is **max[i]** (**max[0]** is initialized to -1 which is known to be less than the smallest priority, which is 0 for the idle process).

When solving this system of equations the higher priority of all runnable process is in **max[n]**, which we can access because **n** is the number of tasks, a constant known at compile time. Then the only thing left to do is to get which process is runnable and has the priority that has just been computed. This is done by constructing a boolean array where a cell at index i is $true$ only if the process i is runnable and have the priority we're looking for. Then we have to look for the index of this cell and that's our selected process. Given Lustre arrays support, this is done with a recursive node (once again we can do that because the depth of the recursion, n, is known at compile time) in code 7.

Code 7:

```
node get_last_true (const size: int; array: bool^size)
  returns (index: int);
let
```

```
  index = if array[size - 1] then size - 1
          else with size = 1 then 0
               else get_last_true(size - 1, array[0 .. size - 2]);
tel
```

The **get_last_true** node walk through a boolean array backward looking for a $true$ value and then return its index in the array. In our usage their should be only one $true$ value in the array, but in case not, we arbitrarily choose the last one. In case no $true$ value is encountered in the array, we still return 0 because in our case it represents the idle process which is always runnable.

This end the redevelopment in Lustre of this part of the scheduler. And now that we have a working system with as little low-level code as possible, we can start to move our interest into verification.

5 Model checking

Lustre programs can be verified using different approaches of model checking, depending on what want to prove correct.

Model checking is a technique for automatically verifying correctness properties of finite-state systems. This is done by testing automatically whether a modelisation of a system meets its specification.

5.1 Boolean property checking

One can use the Lesar tool to verify that a certain boolean property is always true. This is done by explorating every reachable state of the automata produced by the Lustre compilation and see if the property is actually true in every possible case.

The problem with this approach is that it only works with boolean property, and that the complexity of the verification is exponential in the number of states in the automata. This means that emulating integers with boolean tuples won't work without a small upperbound on those numbers. This is known as the state explosion problem.

5.2 Program verification

Lustre programs are self-descriptive, thanks to the declarative aspect and the dataflow approach. This means that the verification tools bundled with Lustre can most of the time operate on Lustre programs themselves. When they can't because the program rely on external tools or have a part that is not programmed using Lustre, a model of the program that have the same behavior can be coded in Lustre and then the overall program can be verified.

The advantage of this approach is that the whole program is formally verified and is then certified to work as expected.

In the case of our project there's clearly a need for modelisation since the system part in not programmed in Lustre but in C. The problem is that this part is coded in C specifically because it if not feasible with Lustre so an exact modelisation might not be possible. However, it is still possible to verify Lustre components individually and to extensively tests the system parts.

Since only part of the scheduler are coded in Lustre only those part can be formally verified. This means that the scheduler can't be verified as it is but that it is rather an instance with specific tasks that we are able to verify. This is not a problem since for each Lustre critical real-time software using our system architecture we know almost everything about the tasks at compile time and verification can be done on each particular project individually, certifying their own scheduling before they're actually in production use.

As we jut said, the low level system layer can't be verified for now. Efforts are made in this direction, for instance a Xavier Leroy team at INRIA is working on *Compcert*, a verified C compiler which has ARM (widely used in embedded devices) among it's target architectures. This, again, is why we need this layer as thin as possible.

6 Conclusion and perspectives

6.1 Automatic generation toolchain

Providing an automatic generation toolchain is the ultimate goal. The aim is to be able to build complete kernel usable in embedded system from a set of Lustre tasks plus a configuration file describing their periods, their priorities and which of the inputs and outputs variables are shared and with which other tasks.

The configuration could also specify a target architecture, either x86 (as in

VirtualBox), or specific a ARM model for embedded devices.

6.2 Remaining verification work

Even if I was able to start verifying some properties of the scheduler with the help of Pascal RAYMOND, the two months I spent at Verimag were too short to make all the necessary verification work. Ideally a test suite should be written for the system layer, at least until we can have the same kind of verification for C as we have for Lustre, and a framework for validation of the whole system should also be written. It would allow rapid verification by using the framework to adapt the verification to each specific instance of the system (each set of tasks, architecture etc.) and could be integrated in the automatic generation toolchain.

6.3 Remaining work on the Lego NXT

Now that we have a working and sufficiently advanced system running in VirtualBox we can start porting it to the Lego NXT brick. I didn't have much time to spend on that either, but since I left Kévin MARQUET has made some significant progress in this direction. This will allow further testing and the development of "vertical" communication.

The only "vertical" communication we have for now is between the scheduler and its high level components, but since it's the scheduler which call them all the communication is made through inputs and outputs of Lustre tasks. What we aim to develop is the communication between scheduled Lustre tasks and low-level components such as drivers. This communication would use external functions (in Lustre it is possible to declare external functions that are written in the host language, which is C) and should also follow the Dynamic Buffer Protocol so the update of the data is consistent with Lustre execution loop.

6.4 Conclusion

During this two months at Verimag, I learned a lot of interesting stuff while working on my subject, but also many things about the everyday life in a computer science research lab. Now I can say with even more conviction that the job I want to do is actually computer science researcher.

The project has progressed well during these two months: the dynamic preemptive scheduling works well, the inter-process communication also, the system

layer is already rather small... but there's still many things to do left for future interns!