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

Pablo Rauzy

Ens, dept info

13 septembre 2010

The Verimag Lab

The Synchronous Team at Verimag

Presentation

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.

Activities

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.

Programming Critical Systems

The Need for Verification

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

Speed and Robustness

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.

Lustre

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

Definition

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.

Definition

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.

Definition

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.

Definition

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.

- The Lustre compiler generate a finite state automata encoded in a *host language* (C), this is the step function.
- The programs then run as an infinite loop "read-inputs, compute, update-variables, write-outputs".

Current Issues

Electronic Era

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.

Static Scheduling

As of now, Lustre allow concurrent programming in an entirely deterministic way, using a static scheduling of tasks computed at compile time.

Dynamic Preemptive Scheduling

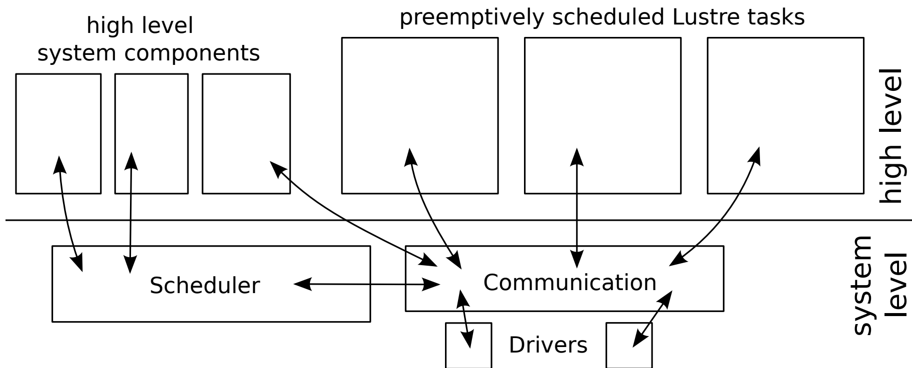
Definition

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*.

The Need for a System Layer

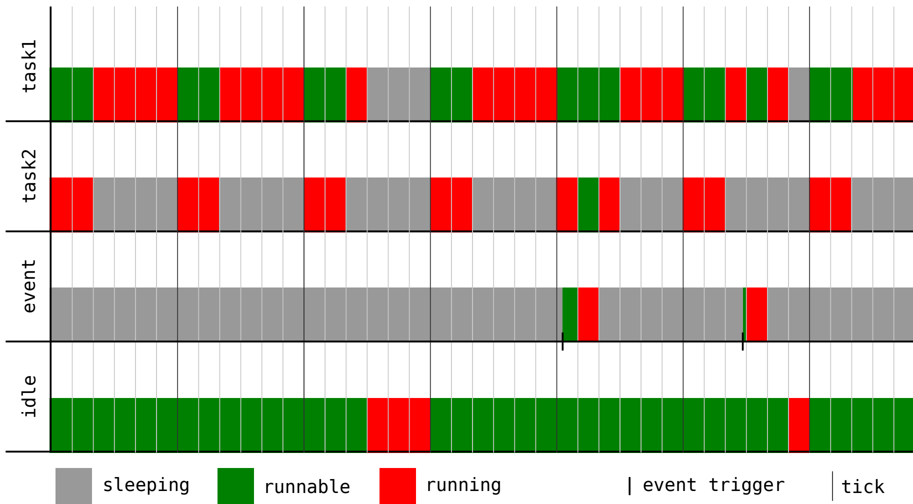
Preemptive scheduling thus require an external agent to make the context switches when necessary, hence we need a low-level layer between our programs and the hardware responsible for that.

The Goal of my Internship



Dynamic Preemptive Scheduling

An example

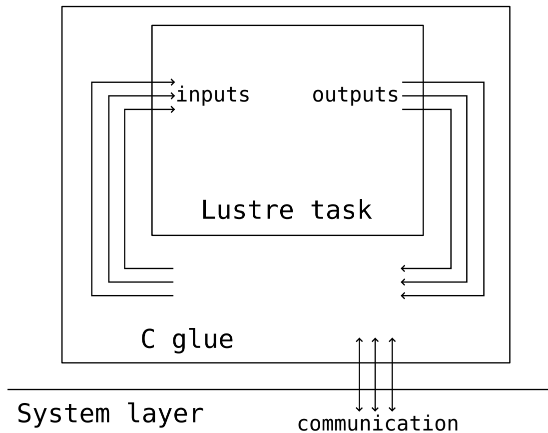


Development Platform: VirtualBox

Why?

- Deployment for testing is really quick.
- Easily connected to a debugger.

A Lustre Process



Context switching

- It consists in saving the state of a process before switching to another, to be able to restore it later.
- This operation is very low level so it has to be coded in assembly language.
- We need to call the scheduler at a very high frequency.
- Tests using a simple round robin scheduler and “fake” tasks.

Preemptive Scheduling of Periodic Tasks

- Check if some processes start a new period.
- Mark them as runnable and add them into a max-heap which uses process priorities as keys.
- Context switch if needed.
- When a task has finished a clycle it calls the scheduler to be put in the sleepings list and removed from the runnables heap.
- It is required to have an idle tasks.

Sporadic Tasks

- Managed like to periodic tasks, but with an interruption trigger.
- Allow to have only one kind of process to take care of for verification purpose.

Inter-Process Communication

Issues and Requirements

- Imposed by Lustre working loop: "read-inputs, compute, update-variables, write-outputs".
- Shared variables used for communications must not change during a cycle to preserve the synchronous and equationnal semantic of Lustre programs.

Dynamic Buffer Protocol

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*" discussing the subject and proposing a solution.

Dynamic Buffer Protocol

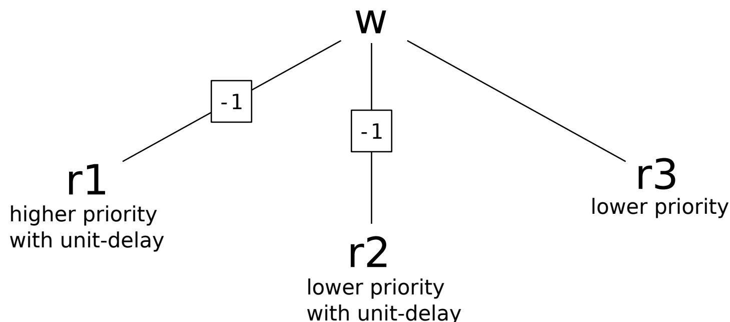
Presentation

DBP is an inter-task communication protocol that is semantics-preserving and memory-optimal (In the worst case, DBP requires at most $N + 2$ buffers for each writer, where N is the number of readers for this writer). 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.

Why?

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.

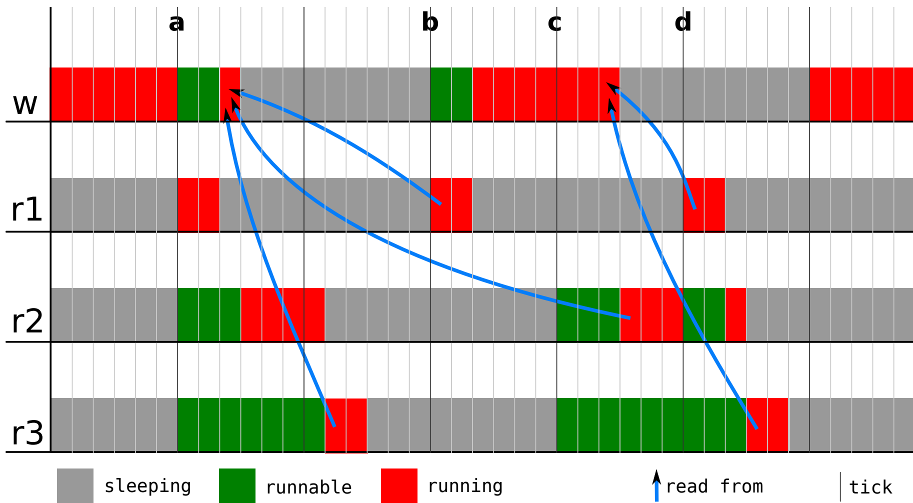
Dynamic Buffer Protocol
An Example



Dynamic Buffer Protocol

- 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.
- The reader task $r1$, which has a higher priority than w, maintain a pointer **`p1`**, which is set to **`previous`** when $r1$ becomes runnable, and read from **`B[p1]`**.
- 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).

Dynamic Buffer Protocol and Preemptive Scheduling



Writing System Components in Lustre

Why Not the Whole System?

- Ideally we would like to have everything coded in Lustre.
- It's not possible: `context_switch` is a non-returning call, it doesn't fit in Lustre "read-inputs, compute, update-variables, write-outputs" loop.
- So we want as much of the scheduler logic as possible in Lustre, and keep the very low-level basic operations in C.

Which Components?

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.

Periodic Tasks Clock in Lustre

```
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
```

Selection of the Next Running Task in Lustre

```
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
```

Selection of the Next Running Task in Lustre

get_next

```
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
```

Selection of the Next Running Task in Lustre

get_last_true

```
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
```

Model Checking

Definition

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.

The `Lesar` Tool

- `Lesar` explore the graph of all the possible states of the automata representing the Lustre program to check if a boolean property is always true or not.
- The method used works only with boolean variables and thus integers and operation on integers have to be simulated using boolean.
- The complexity of the verification is exponential in the number of states of the automata (state explosion problem).

The Lesar Tool

- `Lesar` explore the graph of all the possible states of the automata representing the Lustre program to check if a boolean property is always true or not.
- The method used works only with boolean variables and thus integers and operation on integers have to be simulated using boolean.
- The complexity of the verification is exponential in the number of states of the automata (state explosion problem).

Model Checking for Lustre

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.

Verification for my Project

Verifying the Tasks and Test the System

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.

Conclusion and Perspectives

- Automatic generation toolchain.
- Remaining verification work.
- Vertical communication.