Model-Based Design of Embedded Control Systems by means of a Synchronous Intermediate Model

Mouaiad Alras, Paul Caspi, Alain Girault, and Pascal Raymond
Verimag-CNRS and University of Grenoble – France
INRIA Grenoble Rhône-Alpes – France

Abstract

Model-based design (MBD) involves designing a model of a control system, simulating and debugging it with dedicated tools, and finally generating automatically code corresponding to this model. In the domain of embedded systems, it offers the huge advantage of avoiding the time-consuming and error-prone final coding phase. The main issue raised by MBD is the faithfulness of the generated code with respect to the initial model, the latter being defined by the simulation semantics. To bridge the gap between the high-level model and the low-level implementation, we use the synchronous programming language Lustre as an intermediate formal model. Concretely, starting from a high-level model specified in the de-facto standard Simulink, we first generate Lustre code along with some structured "glue code", and then we generate embedded real-time code for the Xenomai RTOS. Thanks to Lustre's clean mathematical semantics, we are able to guarantee the faithfulness of the generated multi-tasked real-time code.

1 Introduction

1.1 Overview

A classical automatic control application consists of a software controller interacting with the physical device/environment via dedicated input and output drivers. Accordingly, engineers use a design environment both to model the physical environment, using continuous domain paradigms (e.g., differential equations), and to design the controller, using sampled discrete time paradigms (e.g., periodic clocks, multi-tasking, priorities). They simulate and debug their application and, once satisfied with it, they implement it over a target architecture consisting of a distributed hardware platform and a real-time operating system (RTOS). One of the most popular such design environment is Simulink, a de-facto standard in industry (automotive, robotics and automation, consumer electronics,...), and our work is based on it.

For lack of adequate tools, past common practice in industry was to implement *manually* the controller, an operation that is both time-consuming and error-prone. Model-based design (MBD) precisely attempts at replacing the manual coding phase by an automatic code generation one. MBD is a very active area of research, both in academia and in private companies. The main assumptions it relies on are that the designer is satisfied with his/her controller (thanks to simulation and testing), that the software paradigms can be implemented on the architecture, and that the libraries of drivers are consistent with the physical device. Some MBD solutions exist today, for instance in avionics with SCADE [4], in automotive with Simulink-RTW [14], and in robotics with Orccad [3, 13].

The core principle of MBD is that the designer wants to obtain code that is *faithful* to his/her design; in other words, what he/she simulates under Simulink must execute exactly in the same manner on the target architecture. As a consequence, MBD must derive as automatically as possible code that is faithful to the design of the controller, and that can be implemented on the target architecture. Of course, since Simulink does not have a denotational semantics, it is impossible to prove formally this faithfulness.

The main issue that stems from the faithfulness requirement is that the semantics of a Simulink model depends on the chosen simulation parameters. For instance, some Simulink models may be accepted if one chooses `discrete-step` simulation, and rejected if one chooses `variable-step`, auto, or multi-threaded simulation. We call those *simulation artifacts*.

This is why we propose to use an *intermediate formal model*, to bridge the gap between the high-level Simulink model (i.e., the automatic control world) and the low-level implementation (i.e., the computer science world). We choose Lustre [10] to express formally the intermediate model, because both Lustre and Simulink are data-flow language that implement the synchronous model of computation, and because Lustre is equipped with a clean mathe-

978-0-7695-3678-1/09 $25.00 © 2009 IEEE
DOI 10.1109/ICESS.2009.36

matical semantics. Our workflow is therefore:

Simulink → Lustre → multiple tasks over a RTOS

Our target RTOS is Xenomai, chosen for usability, maintainability, and portability reasons.

We also generate some structured "glue code", necessary to express features of Simulink that do not exist in Lustre. This is the case, for instance, of periodic clocks and sporadic events.

1.2 Contribution

Our goal is to develop a complete multi-tier MBD tool that goes from a high-level Simulink model to a low-level real-time implementation for the Xenomai RTOS. We involve as well an early conformity analyzing stage.

We extend existing MBD methods also based on Simulink and Lustre [6, 7, 11, 15] to integrate more features of the high-level model. We extend Lustre with specific *meta-operators*, which are used to subsume these features on the one hand, and to incarnate on the other hand the real-time and system-level items of the Xenomai library. We call the extended language *Lustre++*.

In addition to high-level restrictions that prevent unsafe behaviors of Simulink, and in order to produce better code, we propose some confinements and user guidelines in two directions: Grouping tasks to generate the minimum number of RTOS tasks, and using exclusive modes with data sharing to generate more efficient and better structured code.

Our translation is modular and preserves the hierarchical structure of the Simulink model. It is implemented in a prototype tool consisting of two steps. First, our tool translates the Simulink model into our intermediate model Lustre++. Then, this intermediate model can be used in two ways (see Section 4): On the one hand it can be compiled into embedded RTOS code, and on the other hand it can translated into pure Lustre code for the simulation and validation of the behavioral part of the system.

Some existing approaches in industry only handle the mono-processor and mono-task case [14, 15]. Extensions have been proposed for the multi-task case [5, 12], but they are limited to periodic tasks where all the periods are a multiple of the smallest period. The method we present here extends [15] in two directions: on the one hand we handle sporadic tasks and arbitrary periodic tasks, and on the other hand we generate embedded real-time code while [15] stopped at the Lustre level. We use previously developed theoretical results for the inter-task communication protocols [7, 11].

2 The Intermediate Model

2.1 From a Simulink model to a synchronous program

Between the high-level Simulink model and the actual implementation, we use an intermediate model, based on the synchronous programming language Lustre [10]. This translation is hierarchical, that is, the structure of the resulting synchronous model reflects the one of the Simulink model. In order to obtain this structure, we must extract from the Simulink model the following information:

- Subsystems involving features related to asynchronous concepts, such as multi-tasking, periodic clocks, priorities etc. We call this part of the hierarchy the *asynchronous architecture*, and it is intended to be implemented by using features of the target RTOS. This part corresponds to the tree-like structure of the intermediate model.
- Subsystems that can be considered as "functional": they correspond to synchronous tasks, which can be straightforwardly translated into Lustre, and then into classical sequential programs using the Lustre code generator. Those subsystems are considered as atomic in our framwork: they are the leaves of the intermediate model structure.

Figure **??** shows an example of a hierachical Simulink model, together with the corresponding intermediate model structure. In this example, the top-level is made of three concurrent subsystems: the H and G blocks have been identified as "functional" (typically because they only involve simple computations, not detailed here). On the contrary, the F block contains non-purely functional features: a Data Store, and two subsystems triggered by a clock-enable condition: F is then identified as a non-functional node in the intermediate structure. Finally, F is itself made of two systems that are identified as atomic.

Indeed, the distinction between atomic and non-atomic subsystems is not always obvious, this is why we propose guidelines for the design of Simulink models: those guidelines, presented in Sections 2.3 and 2.4 are intended to help the extraction of the intermediate structure. If the Simulink model does not obey the guidelines, we are still able to generate faithful, but less efficient, code.

Since Lustre is notsuited to the expression of the asynchronous features of Simulink (clocks, priorities, and so on), we introduce a set of *meta-operators* to keep track of these informations in the intermediary nodes of the tree structure. In previous solutions, we used to encode thses features with non-structured annotations (e.g., pragmas); we find the meta-operators solution better structured and more elegant.

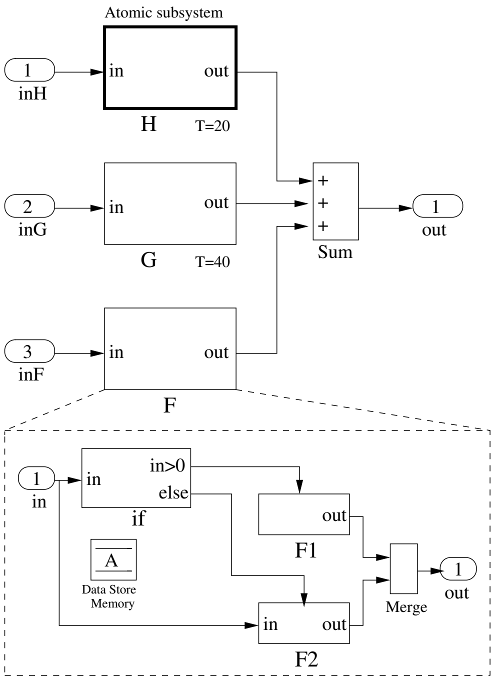

Figure 1. A hierarchical Simulink model.

2.2 Meta-operators

Meta-operators are introduced in order to extend the Lustre language with the most common features one can find in a classical RTOS: triggered tasks, triggers, data buffers. Each meta-operator has a set of static parameters (given between << and >>) for expressing extra-functional information. We call Lustre++ the language extended with meta-operators.

Figure 3 shows the Lustre++ nodes generated for the example of Figure 1. It uses three different meta-operators:

- `clock` has two static parameters called `p` and `h`; it creates a periodic clock of period `p` and phase `h`.
- `condact` has three static parameters called `f`, `c`, and `d`; it invokes the function `f` at each tick of the clock `c`; the parameter `d` is the default value of `f`'s output
- `switch` is a structured collection of `condact` operators, where the clocks are guaranteed to be exclusive.

The meta-operators are designed in order to allow the generation of RTOS code. They can also be translated into pure Lustre code as shown in Figure 4: this program corresponds to the node `main` of the example in Figure 3. The pure Lustre code is not used for embedded code generation because it abstracts away the real-time features. We only use it for testing and model-checking purpose.

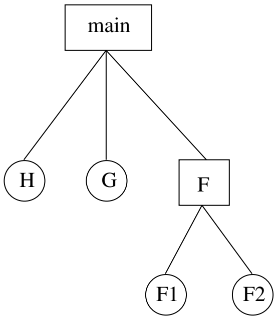

Figure 2. The intermediate model structure corresponding to the Simulink model of Figure 1.

```
node main (inH,inG,inF:int)
returns (out:int);
var
  ck_20, ck_40: alarm;
let
  ck_20 = clock<<20,0>>(_);
  ck_40 = clock<<40,0>>(_);
  out = condact<<H,ck_20,1>>(inH)
       + condact<<G,ck_40,1>>(inG)
       + F(inF);
tel

node F(in:int) returns(out:int);
const
  mem_A: int ref = ref 0;
let
  out = if (in > 0)
    then F1<<mem_A>>(_)
    else F2<<mem_A>>(in);
tel
```

Figure 3. Lustre++ code for the model of Figure 1.

```
node main (inH, inG, inF:int)
returns (out:int);
var
  cnt_20, cnt_40 : int;
  ck_20, ck_40 : bool;
let
  cnt_20 = (0 -> pre(cnt_20) + 1) mod 20;
  ck_20  = ( 0 = cnt_20 );
  cnt_40 = (0 -> pre(cnt_40) + 1) mod 40;
  ck_40  = ( 0 = cnt_40 );

  outH = if(ck_20)
           then current( H(inH when ck_20) )
           else 1 -> pre (outH);
  outG = if(ck_40)
           then current( G(inG when ck_40) )
           else 1 -> pre (outG);
  out = outH + outG + F(inF);
tel
```

Figure 4. Lustre code for the model of Figure 3.

2.3 Atomic subsystem guideline

The default heuristic is to consider that a Simulink subsystem triggered by a clock-enable condition must be implemented by an RTOS task. However, this is not always a good solution, since it may lead to a large number of context switches during the execution that may dramatically slow down the system.

However, we also propose a guideline that allows the user to choose how to regroup computation into a single RTOS task. This is achieved by labeling a block as a Simulink Atomic Subsystem. For instance in Figure 1, the subsystem H is marked as an Atomic Subsystem: this will enforce the code generator to group all hierarchical subsystems contained in H into single RTOS task.

2.4 Exclusive modes guideline

The notion of multi-modes is particularly important in control system design: it corresponds to the case where a state variable is computed according to different control-laws (called the *modes*) depending on the state of the system.

Muli-mode programming is not a built-in paradigm in Simulink, but rather an (expected) consequence of the design. The system F in Figure 1 is a typical example of multi-modes design: F1 and F2 are triggered by exclusive conditions, and their outputs are merged into a single variable “out”.

Recognizing this kind of organization in the simulink model is very important in our approach, since it allows to produce better implementation:

- it is not necessary to protect the access to shared variables, since these accesses are guaranteed to be exclusive,
- instead of generating a system task for each mode, we can produce a single task that dynamically select the right mode.

We propose a guideline that guarantees efficient generated code; a set of n blocks are recognized as *exclusive modes of computation* if:

1. Each block is triggered by an activation condition and the n activation conditions are produced by a single `switch` block.
2. Each output of these n block that is common to at least two blocks must be common to the n blocks and must be combined into a `merge` block.

We illustrate this guideline in the next section by a complete example.

2.5 Example

The example in Figure 5 is based on the controller part of the automotive power train model designed at Ford Motors Company [9, 8] to study the 1-2 gearshift. This Simulink model contains five triggered subsystems (`first`, `second`, `change_of_mind`, `torque_12`, and `inertia_12`) witch all read the same input `T_t` to calculate differently the same outputs `tc1` and `tc2`. Some subsystems need additional inputs and may also be connected internally to other subsystems. For instance, the `inertia_12` subsystem receives its `last_torque` and `last_omega` inputs from `torque_12`.

The `merge` operator is used in Simulink when the same variable is computed by different modes of computation. We restrict the usage of the `merge` operator to exlusive modes of computation (Section 2.4).

Goto/From Tags are used in Simulink as connectors to avoid crossing signal lines . For instance in Figure 5, this is the case of the `reset_inertia` output of block `mode_control` that is connected to the `inertia_12` block. We restrict the use of such tags to be scoped only locally (in other words, Goto/From Tags that span over hierarchy levels are forbidden). Figure 6 shows the Lustre++ code corresponding to this example.

Another way to share variables between subsystems could be achieved by the use of Data Stores. We chose to take into account only local Data Stores declared explicitly using the Memory Block of the Simulink Library (see, e.g, the Data Store A in Figure 1, block F).

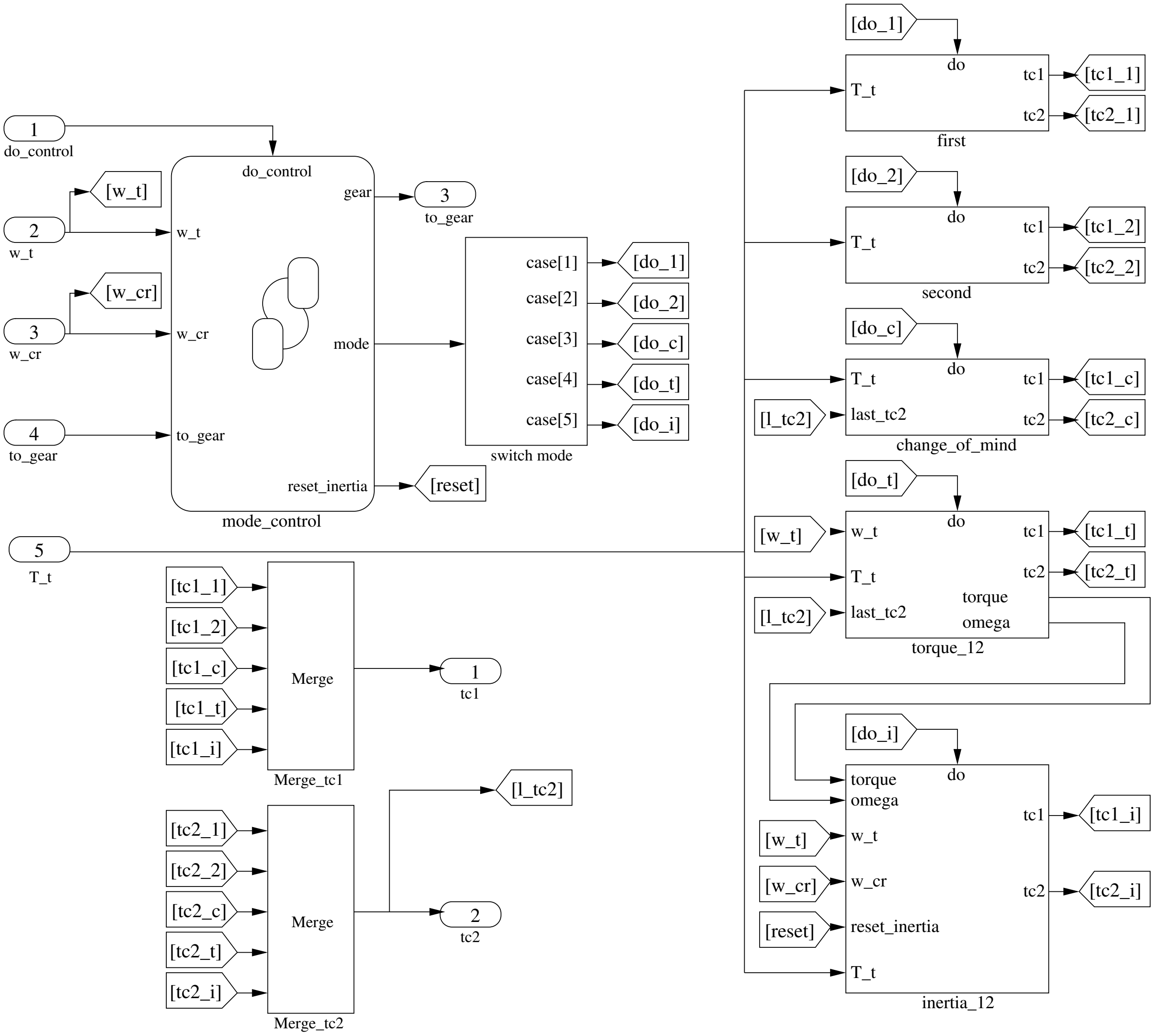

Figure 5. Gear Shift Control Example.

```
type modes_shift = enum {
  do_1, do_2, do_c, do_t, do_i
};

node shift_control(
   i_do_control: bool,i_to_gear: int,
   i_w_t: real, i_w_cr: real, i_T_t: real
   )
returns(o_tc1:real, o_tc2:real, o_gear:int)
var
  t_mode: modes_shift;
  t_reset: bool;
const
  tc2 : real = 0;
  omega : real = 0;
  torque : real = 0;
let
  (o_gear, t_mode, t_reset) =
  condact<<mode_control, in_do_control>>
  (i_omega_t, i_omega_cr, i_to_gear);

(o_tc1, o_tc2)
= switch<<
[
 first,
 seconde,
 change_of_mind<<tc2>>,
 torque_12<<tc2,torque,omega>>,
 inertia_12<<torque,omega>>
],
[
 do_1,
 do_2,
 do_c,
 do_t,
 do_i
]
>>
(w_t, w_cr, reset_inertia, T_t);
tel
```

Figure 6. Lustre++ code for the model of Figure 5.

3 The Xenomai Real-Time Operating System

Several RTOS have been considered: Xenomai[1], RTLinux[2], and PaRTiKle[3]. For usability, maintainability, and portability reasons, we have chosen Xenomai, running on top of RTAI[4]. We use it as a platform for implementing the embedded code generated from the Simulink models. This RTOS provides a considerably useful set of services for the implementation of real-time systems.

In Xenomai, the basic object performing actions is a *task*, a logically complete piece of application code. The Xenomai scheduler ensures that concurrent tasks are run according to some chosen scheduling policy. The two supported scheduling policies are the fixed priority-based FIFO and the round-robin policies.

Any Xenomai task may create any number of watchdog timers, called *alarms*: once the specified delay has elapsed, a user-defined handler is run. An alarm can be either periodic or "one-shot"; in the former case, the real-time kernel automatically reprograms the alarm for the next iteration, according to the user-defined interval value.

To insure the synchronization between tasks, Xenomai offers *condition variables*, which allow tasks to suspend their execution until some predicate on condition variables is satisfied. A condition variable must always be associated with a `mutex` (mutual exclusion section), to avoid race conditions occurring when one task is waiting for a condition variable while another task signals the condition just before the first task waits for it.

For instance, a portion of the Xenomai real-time code generated for the model of Figure 1 is shown in Figure 7. The information required to set the task parameters is assumed to be known from the high level model and given by the user.

The `ck_40` object is created (`rt_alarm_create`) to be triggered every `40` milliseconds and is started (`rt_alarm_start`) without any offset. It will fire the task object `task_G` created (`rt_task_create`) and associated to the corresponding body function `calc_G` which can be seen as a wrapper of `G`'s step function. This portion of code corresponds to the meta-operator `condact<<G,ck_40,1>>(in2)`. The step function is a C code function generated automatically from the corresponding functional Lustre node by the tool `lus2C` (Figure 8).

Xenomai implements the notion of time base, by which timers may be clocked separately according to distinct frequencies given in number of ticks; the duration of a tick is specified by the time base. Such a periodic time base is managed to be compatible with the timing mechanisms of

[1] http://www.xenomai.org

[2] http://www.rtlinuxfree.com

[3] http://www.e-rtl.org/partikle

[4] https://www.rtai.org

```
rt_task task_G;
rt_alarm ck_40;

rt_alarm_create(&ck_40,"ck_40");
rt_alram_start(&ck_40, 0, ns2ticks(40 * TICK));
rt_task_create(&task_G, "task_G", SIZE, PRIO, MODE);

ctx->out1 = 1; //initial value
rt_task_start(&task_G, &calc_G, ctx);

void calc_G(void* ctx){
    while(1){
       rt_alarm_wait(&ck_40);
       ...
       call_step(ctx);
       ...
    }
}
```

Figure 7. Xenomai code for the Simulink model of Figure 1.

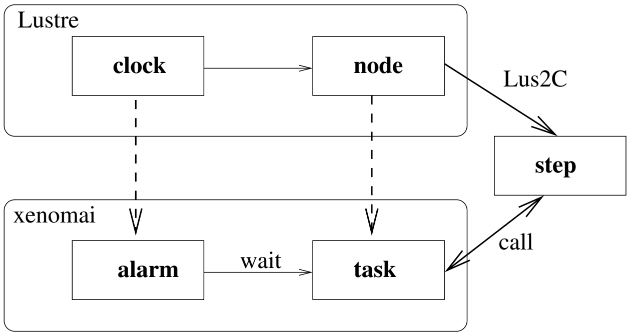

Figure 8. From Lustre to Xenomai example.

simulation in Lustre.

When generating real-time code for automatic control systems, one of the difficult issues concerns blocks that have different periods and that communicate together. Accordingly, there are three cases for a given pair of communicating tasks: high to low priority, high to low priority with unit delay, and low to high priority with unit delay[5] [11]. The difficulty is to guarantee the zero-delay semantics of Simulink. This is achieved thanks to a set of buffers and pointers to these buffers [7]. The principle is that the pointers to the buffers are manipulated upon the arrival of the events triggering the tasks instead of during the execution of the tasks. In that way, the order of arrivals can be memorized and the original Simulink semantics can be preserved [11].

4 Work flow

Our MBD approach is based on a three-phase process. First, Simulink models are parsed thanks to the translator MDL2XML. It extends the existing similar compiler from Sofronis [15] to handle periodic clocks, data stores, atomic subsystems, and exclusive modes of computation.

The MDL2XML translator first involves a filtering and checking stage. Models that contain global data stores or global connectors will be rejected. The output of this stage is a transformed XML file, which is the entry point of the backend code generation.

Second, this XML-encoded Simulink model is translated into an intermediate formal model thanks to the tool XML2LUS. It performs the clock inference, type inference, and conformity analysis. The purpose of this analysis is twofold: to detect combinatorial loops and nondeterministic behaviors (non exclusive data sharing), and to recognize patterns respecting the guidelines in order to generate more efficient code (exclusive modes and tasks grouping).

The XML2LUS tool then generates, for each Simulink block and subsystem, the corresponding Lustre node, and external function code for every “unknown” and “user-defined” block. In addition, glue-code is generated for the meta-operators by retrieving information from the high-level model.

Third, the intermediate model is compiled into actual RTOS code thanks to the tool LUS2XEN. It generates, for each Lustre node, the corresponding real-time task or system call, and for each meta-operator the corresponding real-time object from the native library of Xenomai. Periodic tasks are triggered by alarm objects, sporadic tasks are triggered by event objects, buffers are implemented by condition variables and mutex services. Additional Xenomai glue code is generated to manage the master time base, which serves as the specification basis for delays and timeouts. Moreover, the intermediate model can also be translated into a purely functional model that can be used for formal validation (model-checking, test case generation ...). This is a distinctive feature of our approach and would not be achievable with a direct C code generation such as Simulink-RTW [14].

5 Conclusion and Future Work

We have presented a Model-Based Design (MBD) method for embedded control systems specified in Simulink. Once the model has been simulated and the designer is satisfied with it, we transform it into an intermediate formal model, for which we have chosen the synchronous programming language Lustre. Lustre and Simulink are both synchronous data-flow programming

[5] The low to high priority without unit delay is impossible.

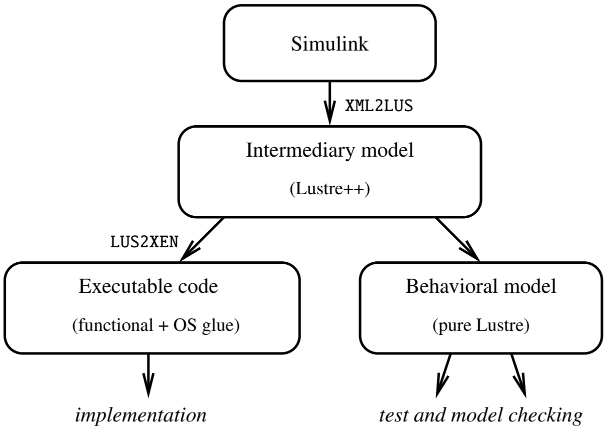

Figure 9. Our work flow with the intermediate model.

languages, so this choice is natural. Besides, Lustre is equipped with a clean mathematical semantics, so our intermediate model allows the designer to use formal validation tools, e.g., model-checkers and test-case generators, to validate his/her Simulink model. More importantly, our intermediate formal model serves as a starting point to generate multi-task real-time code to be executed on the Xenomai RTOS. Thanks to the formal semantics of Lustre, the low-level RTOS code is faithful to the high-level Simulink model. This faithfulness is the central feature of our MBD method.

Our intermediate formal model consists of two parts: on the one hand classical Lustre nodes that encode the purely functional part of the Simulink model, and on the other hand ad-hoc meta-operators that encode the asynchronous features of the Simulink model, such as the periodic clocks, the priorities, and so on. The meta-operators are structured hierarchically to reflect the hierarchical structure of the Simulink model. During the real-time code generation phase, the functional part (i.e., the classical Lustre nodes) are translated into RTOS tasks whose C code is directly obtained via the usual Lustre to C compiler. Concerning the meta-operators, they are translated into RTOS code thanks to the services offered by Xenomai (i.e., tasks, alarms, condition variables, mutexes, and so on).

Even if we don't address, in this paper, classical real-time problems (WCET analysis, scheduling feasibility, etc...), we are aware of these orthogonal problems. Faithful translation can be guaranteed only under the condition that all real-time constraints are satisfied [2, 1].

In future work, we will generalize our MBD method to a wider class of meta-operators to cover more RTOS sevices, and we will enhance the intermediate model with fault tolerance and distribution annotations.

References

[1] A. N. Audsley, A. Burns, M. Richardson, and K. Tindell. Applying New Scheduling Theory to Static Priority Preemptive Scheduling. *Software Engineering Journal*, pages 284–292, 1993.

[2] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. Hard Real-Time Scheduling: The Deadline Monotonic Approach. In *Proceedings 8th IEEE Workshop on Real-Time Operating Systems and Software*, Atalanta, 1991.

[3] J.-J. Borrelly, E. Coste-Manière, B. Espiau, K. Kapellos, R. Pissard-Gibollet, D. Simon, and N. Turro. The Orccad architecture. *Int. J. on Robotic Research*, 17(4):338–359, Apr. 1998.

[4] D. Brière, D. Ribot, D. Pilaud, and J.-L. Camus. Methods and specifications tools for Airbus on-board systems. In *Avionics Conference and Exhibition*, London, UK, Dec. 1994. ERA Technology.

[5] J.-L. Camus, P. Vincent, O. Graff, and S. Poussard. A verifiable architecture for multi-task, multi-rate synchronous software. In *International Conference on Embedded Real-Time Software, ERTS'08*, Paris, France, Jan. 2008.

[6] P. Caspi, A. Curic, A. Maignan, C. Sofronis, S. Tripakis, and P. Niebert. From Simulink to Scade/Lustre to TTA: A layered approach for distributed embedded applications. In *LCTES'03*, pages 153–162. ACM, New-York, June 2003.

[7] P. Caspi, N. Scaife, C. Sofronis, and S. Tripakis. Semantics-preserving multitask implementation of synchronous programs. *ACM Trans. Embedd. Comput. Syst.*, 7(2), Feb. 2008.

[8] Ford Motor Company. *Structured Analysis Using Matlab/Simulink/Stateflow - Modeling Style Guidelines*, 1999.

[9] Ford Research Laboratory. *Hybrid Models for Automotive Powertrain System - Revisiting a Vision*, 2000.

[10] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, Sept. 1991.

[11] N. Scaife and P. Caspi. Integrating model-based design and preemptive scheduling in mixed time- and event-triggered systems. In *Euromicro Conference on Real-Time Systems, ECRTS'04*, pages 119–126, Catania, Italy, June 2004. Verimag research report TR-2004-12.

[12] D. Simon and F. Benattar. Design of real-time periodic control systems through synchronisation and fixed priorities. *Int. J. on of Systems Science*, 36(2):57–76, Feb. 2005.

[13] D. Simon and A. Girault. Synchronous programming of automatic control applications using Orccad and Esterel. In *IEEE Conference on Decision and Control, CDC'01*, Orlando (FL), USA, Dec. 2001. Invited Session.

[14] The MathWorks, Inc. *Automatic Code Generation Technology For Embedded Control Applications*, Nov. 2002.

[15] S. Tripakis, C. Sofronis, P. Caspi, and A. Curic. Translating discrete-time simulink to Lustre. *ACM Trans. Embedd. Comput. Syst.*, 4(4):779–818, Nov. 2005.