From Simulink[1] to SCADE[2]/Lustre to TTA: a layered approach for distributed embedded applications

Paul Caspi, Adrian Curic, Aude Maignan, Christos Sofronis, Stavros Tripakis [3]
Peter Niebert[4]

Abstract

We present a layered end-to-end approach for the design and implementation of embedded software on a distributed platform. The approach comprises a high-level modeling and simulation layer (Simulink), a middle-level programming and validation layer (SCADE/Lustre) and a low-level execution layer (TTA). We provide algorithms and tools to pass from one layer to the next: a translator from Simulink to SCADE/Lustre, a set of real-time and code-distribution extension to SCADE/Lustre, and implementation tools for decomposing a SCADE/Lustre program into tasks, scheduling the tasks as a multi-period multi-processor scheduling problem and distributing the tasks on the execution platform along with the necessary "glue" code.

1 Introduction

Designing safety-critical control systems requires a seamless cooperation of tools at several levels — modeling and design tools at the control level, development tools at the software level and implementation tools at the platform level. When systems are distributed, the choice of the platform is even more important and the implementation tools must be chosen accordingly. A tool-box achieving such a cooperation would allow important savings in design and development time as well as safety increases and cost effectiveness.

In the course of several European IST projects — SafeAir, Next-TTA and Rise, such a goal has been progressively approached and partially prototyped. This paper reports the achievements up to now. The developments were based on the following choice of tools at the different levels: Simulink at the control design level, SCADE/Lustre at the software design level and TTA at the distributed platform level. Why such a choice?

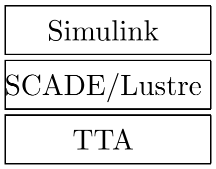

Figure 1: A design approach in three layers.

Simulink: The choice of Simulink is very natural, since it is considered a *de-facto* standard in control design, in domains such as automotive or avionics.

SCADE/Lustre: SCADE (Safety Critical Application Development Environment) is a tool-suite based on the *synchronous* paradigm and the Lustre [7] language. Its graphical modeling environment is endowed with a DO178B-level-A automatic code generator which makes it able to be used in highest criticality applications. Besides, a simulator and model checkers come along with the tool as plug-ins. It has been used in important European avionic projects (Airbus A340-600, A380, Eurocopter) and is also becoming a de-facto standard in this field.

TTA: TTA (Time Triggered Architecture) [12] supports distributed implementations built upon a synchronous bus delivering to every computing unit a global fault-tolerant clock. It is currently used in a number of automotive and avionics applications. Furthermore, it ideally matches the synchronous paradigm and can be seen as well adapted to our framework.

Although SCADE/Lustre can be seen as a strict subset of Simulink (the discrete-time part) and a number of code generators for Simulink exist (e.g., Real-Time Workshop, dSpace), we still believe it is important to add the extra layer (SCADE/Lustre) between Simulink and TTA. One simple reason is that important companies already use SCADE/Lustre in their tool-chain. The level-A qualified code generator is also a crucial aspect that makes certification considerably

[1]Matlab and Simulink are Registered Trademarks of The MathWorks, Inc., Natick, MA.

[2]SCADE and Simulink Gateway are Registered Trademarks of Esterel Technologies, SA, Parc Euclide - 8, rue Blaise Pascal, 78996 Elancourt cedex, France.

[3]VERIMAG, Centre Equation, 2, avenue de Vignate, 38610 Gières, France. Web: www-verimag.imag.fr. This work has been supported in part by the European IST project "Next TTA" under project No IST-2001-32111.

[4]Laboratoire d'Informatique Fondamentale (LIF), CMI, 39 rue Joliot-Curie, F-13453 Marseille Cedex 13, www.lif.univ-mrs.fr.

easier. Another reason is that powerful analysis tools such as model-checkers and test generators are available for SCADE/Lustre, but not for Simulink. Finally, Simulink was initially conceived as a simulation tool, whereas SCADE/Lustre was initially conceived as a programming tool. These different origins become apparent when we examine the (weak) typing features of Simulink, its lack of modularity sometimes, and its multiplicity of semantics (depending on user-controlled "switches" such as the simulation method). On the contrary, SCADE/Lustre was designed from the very beginning as a programming language and has the above features. Therefore, it can serve as a reliable middle layer which filters-out Simulink ambiguities and enforces a strict programming discipline, much needed in safety critical applications.

In the rest of the paper we describe the work done at each of the three layers. First, the translation of Simulink to SCADE/Lustre. Second, extensions to SCADE/Lustre for specifying code distribution and real-time constraints. Third, implementation of SCADE/Lustre on the distributed time-triggered platform TTA. Related work is discussed at the end.

2 Overview of the three layers

For completeness of the paper, we briefly describe Simulink, SCADE/Lustre and TTA.

2.1 A short description of Simulink

Simulink is a module of Matlab for modeling data flow transfer laws. The Simulink notation and interface are close to the control engineering culture and knowledge. The user does not need any particular knowledge of software engineering. The control laws are designed with mathematical tools. The validation is made through frequency analysis and simulation. For more details, the reader can look at the MathWorks web site (www.mathworks.com). Most important features of Simulink, which also affect the translation to SCADE/Lustre, are described in Section 3.

2.2 A short description of SCADE/Lustre

SCADE is a graphical environment commercialized by Esterel Technologies. It is based on the synchronous language Lustre. A Lustre program essentially defines a set of equations:

$$\begin{aligned} x_1 &= f_1(x_1,...,x_n,u_1,...,u_m) \\ x_2 &= f_2(x_1,...,x_n,u_1,...,u_m) \\ &\vdots \end{aligned}$$

where x_i are *internal* or *output* variables, and u_i are *input* variables. The variables in Lustre denote *flows*. A flow is a pair (v, τ), where v is an infinite sequence of *values* and τ is an infinite sequence of *instants*. A value has a *type*. All values of v have the same type, which is the type of the flow. Basic types in Lustre are boolean, integer and real. Composite types are defined by tuples of variables of the same type (e.g., (x, y), where x and y are integers). An instant is a natural number. If $\tau = 0, 2, 4, \cdots$, then the understanding is that the flow is "alive" (and, therefore, its value needs to be computed) only on the even instants. The sequence τ can be equivalently represented as a boolean flow, $b_0, b_1, \cdots$, with the understanding that τ is the sequence of indices i such that $b_i = \mathit{true}$.

The functions f_i are made up of usual arithmetic operations, control-flow operators (e.g., `if then else`), plus a few more operators, namely, `pre`, `->`, `when` and `current`.

`pre` is used to give *memory* (*state*) to a program. More precisely, `pre`(x) defines a flow y, such that the value of y at instant i is equal to the value of x at instant $i - 1$ (for the first instant, the value is undefined).

`->` initializes a flow. If $z = x$`->`y, then the value of z is equal to the value of x at the first instant, and equal to the value of y there-after. The operator `->` is typically used to initialize a flow the value of which is undefined at the first instant (e.g., a flow obtained by `pre`). For example, a counter of instants is defined by the equation $x = 0$`->`(`pre`$(x) + 1$), instead of $x =$ `pre`$(x) + 1$, which leaves it undefined.

`when` is used to *sample* a flow, creating a flow which lives less frequently than the original flow: x `when` b, where x is a flow and b is a boolean flow which lives at the same instants as x, defines a flow y which lives only when b is *true*, and has the same value as x on those instants.

`current` is used to extend the life of a sampled flow y, to the instants of the flow which originally gave birth to y, by the usual *sample-and-hold* rule: `current`(y), where y is a flow sampled from x, is a flow x' which leaves on the same instants as x, has the value of y on the instants when y is alive, and keeps its previous value during the instants when y is not alive.

Structure is given to a Lustre program by declaring and calling Lustre nodes, in much the same way as, say, C functions are declared and called. Here is an example of node declaration in Lustre:

```
node A(b: bool; i: int; x: real)
returns (y: real);
  var j: int;
      z: real;
  let
      j = if b then 0 else i;
```

```
    z = B(j, x);
    y = if b then pre(z) else C(z);
tel.
```

A is a node taking as inputs a boolean flow b, an integer flow i and a real flow x and returning a real flow y. A uses internal flows j and z (with usual scope rules). The body of A is declared between the `let` and `tel` keywords. A calls node B to compute z and node C to compute y (conditionally). B and C are declared elsewhere.

Clock calculus: Given a flow $x = (v, \tau)$, τ is called the *clock* of x, and is denoted $\texttt{clock}(x)$. The clock calculus is a typing mechanism which ensures that the Lustre program has a well defined meaning. For example, we cannot add two variables x and y, unless they have the same clock (i.e., they are alive at the same instants): otherwise, what would the result of $x + y$ be on an instant where x is alive and y is not alive?

All input variables have the same clock, which is called the *basic* clock, denoted *basic*, and given by the sequence of instants $0, 1, 2, \cdots$ or equivalently the boolean flow $\mathit{true}, \mathit{true}, \mathit{true}, \cdots$.

A simplified version of the clock calculus of Lustre is shown in Table 1. By convention, $\texttt{clock}(\mathit{basic}) = \mathit{basic}$. If the constraints on clocks are not satisfied, then the program is rejected by the Lustre compiler.

Partial order: A Lustre program defines a *partial order* on x_i (denoted by $\rightarrow$), expressing variable dependencies at each instant: if $x_i \rightarrow x_j$, then x_j depends on x_i, that is, in order to compute the value of x_j at a given instant, we need the value of x_i at that instant. The partial order is well-defined, since the compiler ensures that there are no cyclic dependencies on the variables x_i: all cycles must contain at least one `pre` operator, which means that the dependency is not on the same instant, but on previous instants.

2.3 A short description of TTA

TTA [12] (Time Triggered Architecture) is a distributed, synchronous, fault-tolerant architecture. It includes a set of computers (TTA *nodes*) connected through a *bus*. Each TTA node is equipped with a network card implementing the *time triggered protocol* [13] (TTP). TTP provides a number of services to the nodes, including *clock synchronization*, *group membership* and *faulty node isolation*.

The programs running on each TTA node use the TTP controller to communicate with programs running on other nodes. Communication is *time triggered*. This means that there is an *a-priori* global schedule specifying which TTA node will transmit which message at what time. This schedule, called *message description list* (MEDL), is constructed off-line and loaded on each TTP controller before operation starts. The MEDL ensures that no two TTA nodes transmit at the same time (given the correct functioning of the clock synchronization protocol). Therefore, no on-line arbitration is required (contrary, for example, to the CAN bus).

The TTP controller and the CPU of the TTA node are linked through the computer-network interface (CNI): this is essentially a shared memory where the contents of each message are stored. The programs running on a TTA node read/write on the CNI independently of the TTP controller, which reads and writes according to the MEDL (i.e., when it is time for this node to send/receive a message, the TTP controller will read/write the corresponding part of the CNI).

The MEDL describes the operation of the bus in the global, common time axis, which is produced by the clock synchronization protocol. Time is divided into *cycles, rounds* and *slots* (Figure 2). Cycles are repeated as long as the system runs, in exactly the same way. Each cycle contains a number of rounds and each round a number of slots. Rounds have the same duration, whereas slots within a round may have different durations. Each slot is assigned to a TTA node, meaning that (only) this node transmits during that slot. Within a slot, a node transmits a *frame*, which contains one or more messages. The messages are broadcasted, meaning every other node can read them. The difference between rounds is that the frames of a given slot need not be the same among different rounds of a cycle. For example, if slot 1 is assigned to node A, A may transmit frame X in slot 1 of round 1 and frame Y in slot 1 of round 2. However, operation among different cycles is identical (i.e., A will transmit X in slot 1 of round 1 of cycle 1, of cycle 2, of cycle 3, and so on).

3 Simulink to SCADE/Lustre

In our approach, we start with a Simulink model, consisting of two parts: a discrete-time part describing the controller and a continuous-time (or perhaps discrete-time) part containing the environment in which the controller is supposed to operate. Modeling both the controller and the environment is of course essential for studying the properties of the controller by simulation. Once the designer is happy with the results, the implementation of the controller can start. The first step in our approach is to translate the controller part of the Simulink model to SCADE/Lustre.

In this section we describe this translation. We begin by pointing out the main differences of the two lan-

expression e	`clock`(e)	constraints	comments
input x	*basic*		
$x+y$	`clock`(x)	`clock`(x) = `clock`(y)	similarly for $-$, `->`, `if then else`, etc
`pre`(x)	`clock`(x)		
x `when` b	b	`clock`(x) = `clock`(b), b boolean	
`current`(x)	`clock`(`clock`(x))		

Table 1: Clock calculus of Lustre.

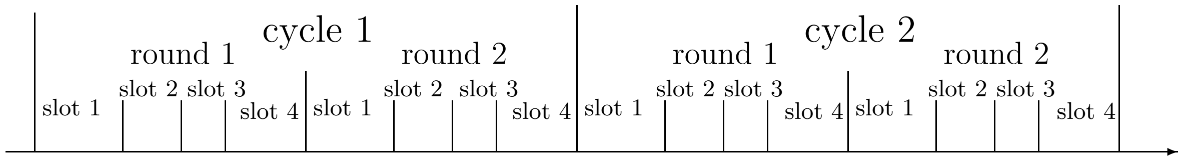

Figure 2: Operation of a TTA bus in time.

guages and which subset of Simulink can be handled by our translation. Then we present the main principles of the translation and discuss two implementations.

We first fix some terminology, to be used in the sequel. For Simulink, we will use the term *block* for a basic block (e.g., an adder, a discrete filter, a transfer function, etc) and the term *subsystem* for a composite (a set of blocks or subsystems linked by signals). The term *system* is used for the root subsystem. For SCADE/Lustre, we will use the term *operator* for a basic operator (e.g., +, `pre`, etc) and the term *node* for a composite.

3.1 Simulink and SCADE/Lustre

Both Simulink and SCADE/Lustre allow the representation of signals and systems, more precisely, *multi-periodic sampled* systems. The two languages share strong similarities, such as a data-flow model,[1] similar abstraction mechanisms (basic and composite components) and graphical description. However, there are several differences:

(1) SCADE/Lustre has a discrete-time semantics, whereas Simulink has a continuous-time semantics. It is important to note that even the "discrete-time library" Simulink blocks produce piece-wise constant continuous-time signals[2].

(2) SCADE/Lustre has a unique, precise semantics. The semantics of Simulink depends on the choice of a simulation method. For instance, some models are accepted if one chooses variable-step simulation and rejected if one chooses fixed-step, "auto", or "multi-threaded" simulation.

[1]The foundations of data-flow models were laid by Kahn [10]. Various such models are studied in [14].

[2]Thus, in general, it is possible to feed the output of a continuous-time block into the input of a discrete-time block and vice-versa.

(3) SCADE/Lustre is a strongly-typed system with explicit type set on each flow. In Simulink, explicit types are not mandatory. A type-checking mechanism exists in Simulink (some models are rejected due to type errors) but, as with the execution semantics, it can be modified by the user by setting some "flags".

(4) SCADE/Lustre is modular in certain aspects, whereas Simulink is not: for instance, a Simulink model may contain implicit inputs (the sampling periods of a system and its sub-systems, which are not always inherited).

3.2 Translation goals and limitations

Given the above differences, the goals and limitations of our translation are the following:

(1) We only translate a discrete-time, non-ambiguous part of Simulink. In particular, we do not translate blocks of the continuous-time library, S-functions, or Matlab functions. The Simulink model to be translated is assumed to be (part of) the controller embedded in a larger model (including the environment).

(2) The translation is faithful only with respect to the following simulation method: "solver: fixed-step, discrete" and "mode: auto".

(3) The SCADE/Lustre program must be run at the time period the Simulink model was simulated. Thus, an outcome of the translation must be the period at which the SCADE/Lustre program *shall* be run (i.e., the period of the basic clock). To know the period at which the Simulink model was simulated, we assume that for every external input of the model to be translated the sampling time is explicitly specified.

(4) We assume that the Simulink model to be trans-

lated has the "*Boolean logic signals*" flag on[3]. Then, a requirement on the translator is to perform exactly the same type inference as Simulink. In particular, every model that is accepted by Simulink must also be accepted by the translator and vice versa.

(5) For reasons of traceability, the translation must preserve the hierarchy of the Simulink model as much as possible.

3.3 Overall translation scheme

First, *type inference* and *clock inference* are performed on the Simulink model. The two steps are independent. Then, the translation per-se is performed hierarchically (bottom-up). We now elaborate on each of these steps.

3.4 Type inference

There are three basic types in SCADE/Lustre: *bool*, *int* and *real*. Each flow has a declared type and operations between different types are not allowed: for example, we cannot add an *int* with a *real*[4]. In Simulink, types need not be explicitly declared. However, Simulink does have typing rules: some models are rejected because of type errors. The objective of the type inference step is to find the type of each Simulink signal, which will then be used as the type of the corresponding SCADE/Lustre flow.

Simulink provides the following *data types*: *boolean*, *double*, *single*, *int8*, *uint8*, *int16*, *uint16*, *int32*, *uint32*. Informally, the type system of Simulink can be described as follows. By default, all signals are *double*, except when: either the user explicitly sets the type of a signal to another type (e.g., by a *Data Type Converter* block or by an expression such as *single(23.4)*); or a signal is used in a block which demands another type (e.g., all inputs and outputs of *Logical Operator* blocks are *boolean*).

We can formalize the above type system as follows. First, denote by $SimT$ the set of all Simulink types and let $SimNum = SimT - \{boolean\}$. Then, every Simulink block has a (polymorphic) type: some examples are shown in Figure 3. The type of a Simulink subsystem (or the root system) A is defined given the types of the subsystems or blocks composing A, using a standard function composition rule. Type inference is done using a standard fix-point computation on the lattice shown in Figure 4.

Once the inference of Simulink types is performed, these types are mapped to SCADE/Lustre types as follows: *boolean* is mapped to *bool*; *int8*, *uint8*, *int16*, *uint16*, *int32* and *uint32* are mapped to *int*; *single* and *double* are mapped to *real*.

[3]This flag yields a stricter type checking in Simulink, for instance, logical blocks accept only boolean inputs.

[4]Predefined *casting* operators *int2real* or *real2int* can be used if necessary.

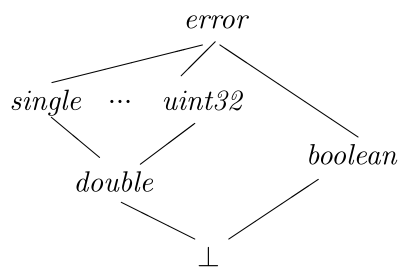

Figure 4: The type lattice of Simulink.

3.5 Clock inference

As mentioned above, a SCADE/Lustre program has a unique basic clock. "Slower" clocks are obtained through the basic clock using the `when` operator. The clock of every signal in SCADE/Lustre is implicitly calculated by the compiler, which ensures that operations involve only flows of the same clock[5]. Thus, clocks can be seen as extra typing information.

Discrete-time Simulink signals may also contain timing information, called "*sample time*", consisting of a *period* and an initial *phase*. The sample time of a signal specifies when the signal is updated. A signal x with period π and initial phase θ is updated only at times $k\pi + \theta$, for $k = 0, 1, 2, ...$, that is, it remains constant during the intervals $[k\pi+\theta, (k+1)\pi+\theta)$. Sample times can be set in input signals and discrete-time blocks and they also serve as an extra type system in Simulink: some models are rejected because of timing errors.

Another timing mechanism of Simulink is by means of "*triggers*". Only subsystems (not basic blocks) can be triggered. A subsystem can be triggered by a signal x (of any type) in three ways, namely, "*rising, falling*" or "*either*", which specify the moment the trigger occurs w.r.t. the direction with which x "crosses" zero (with boolean *true* identified with 1 and false with 0). The sample time of blocks and subsystems inside a triggered subsystem cannot be set by the user: it is "*inherited*" from the sample time T of the triggering signal. The sample times of the input signals must be all equal to T. The sample time of all outputs is also T. Thus, in the example shown in Figure 5, the sample times of s, x_1, x_2 and y are all equal.

In what concerns triggered subsystems, Simulink is as modular as Lustre, where a node B called inside a node A cannot construct a "faster" clock than the basic clock of A (i.e., the clock of its first input). However, Simulink allows the sample time of a subsystem B embedded into a subsystem A to be anything. For

[5]Since checking whether two boolean flows are equal is generally undecidable, clock checking in SCADE/Lustre is syntactic.

$$
\begin{aligned}
\mathit{Constant}_\alpha &: \alpha, \alpha \in \mathit{SimNum} && (1)\\
\mathit{Adder} &: \alpha \times \cdots \times \alpha \rightarrow \alpha, \alpha \in \mathit{SimNum} && (2)\\
\mathit{Relation} &: \alpha \times \alpha \rightarrow \mathit{boolean}, \alpha \in \mathit{SimNum} && (3)\\
\mathit{Logical\ Operator} &: \mathit{boolean} \times \cdots \times \mathit{boolean} \rightarrow \mathit{boolean} && (4)\\
\mathit{Discrete\ Transfer\ Function} &: \mathit{double} \rightarrow \mathit{double} && (5)\\
\mathit{Data\ Type\ Converter}_\alpha &: \beta \rightarrow \alpha, \alpha, \beta \in \mathit{SimT} && (6)
\end{aligned}
$$

Figure 3: Types of some Simulink blocks.

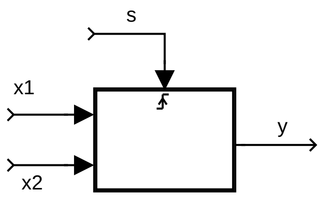

Figure 5: A triggered subsystem.

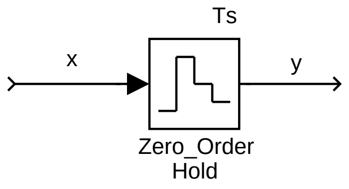

Figure 6: A Zero-order Hold block modifying the period of its input.

instance, the period of A can be 2 and the period of B 1. We consider this a non-modular feature of Simulink.

The objective of clock inference is to compute the period and phase of each Simulink signal, block and subsystem, and use this information when creating the corresponding Lustre flows and nodes and when defining the period at which the SCADE/Lustre program must be run. Due to lack of space, we only give some examples on how this is done. The details will appear in a subsequent paper.

Consider the Simulink model of Figure 6 and assume that the period of input x is 1 and that the period set to the *Zero-order Hold* block is 2.[6] Then, the output y has period 2 and in the generated SCADE/Lustre program, it will be defined as $y = x$ `when` $b_{1/2}$, where $b_{1/2}$ is the boolean flow *true false true false* $\cdots$. Now, if 1 is the smallest period in the entire Simulink model, this will also be the period at which the generated SCADE/Lustre program must be run. In the SCADE/Lustre program, `clock`$(x) = \mathit{basic}$ and `clock`$(y) = b_{1/2}$.

As another example, consider a subsystem A with two inputs x_1 and x_2, with periods π_1 and π_2, respectively. If these are the only periods in the Simulink model, then the period of the SCADE/Lustre program must be the greatest common divisor (GCD) of π_1 and π_2. This will also be the period of the outputs of A.

[6]Unless otherwise mentioned, we assume that phases are 0.

3.6 Hierarchical translation

Logically, a Simulink model is organized as a tree, where the children of a subsystem are the subsystems (or blocks) directly embedded in it. The translation is performed following this hierarchy in a bottom-up fashion (i.e., starting from the basic blocks). For traceability, naming conventions are used, such as suffixing by an index or using the name path along the tree.

Simple basic Simulink blocks (e.g., adders, multipliers, the $\frac{1}{z}$ transfer function) are translated into basic SCADE/Lustre operators. For example, an adder is simply translated into $+$ and $\frac{1}{z}$ is the `pre` operator in SCADE/Lustre.

More complex Simulink blocks (e.g., discrete filters) are translated into SCADE/Lustre nodes. For example, the transfer function $\frac{z+2}{z^2+3z+1}$ is translated into the SCADE/Lustre code:

```
node Transfer_Function_3(E: real)
returns(S: real);
var Em_1, Em_2, Sm_1, Sm_2: real;
let
   S = 1.0*Em_1+2.0*Em_2-3.0*Sm_1-1.0*Sm_2 ;
   Em_1 = 0.0 -> pre(E) ;
   Em_2 = 0.0 -> pre(Em_1) ;
   Sm_1 = 0.0 -> pre(S) ;
   Sm_2 = 0.0 -> pre(Sm_1) ;
tel.
```

A Simulink subsystem S is translated into a SCADE/Lustre node N, possibly containing calls to

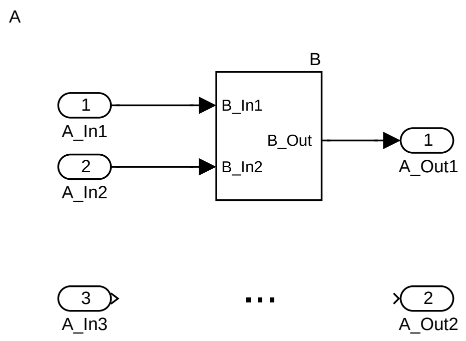

Figure 7: Simulink system A with subsystem B.

other nodes. For example, consider the Simulink system A shown in Figure 7. A contains subsystem B. The Lustre code generated for this example is as shown below.

```
node A(A_in1,A_in2,A_in3 :real)
returns (A_out1,A_out2 :real);
let
   A_out1 = B(A_in1 , A_in2);
   A_out2 = ...
tel.

node B(B_in1,B_in2: real)
returns (B_out);
let
   ...
tel.
```

3.7 Tools

The above ideas have been implemented in two tools: (1) a commercial tool, called SimulinkGateway, by Esterel Technologies; (2) an academic prototype, called Sim2Lus, developed at Verimag. Sim2Lus is implemented in Java. It translates Simulink to an intermediate XML format and then to Lustre.

The tool has been applied in the context of the European IST project "Next TTA" on a case study provided by Audi, on a Simulink model of a warning processing system, part of a larger system actually used in Audi cars. The Simulink model has 5 layers, 20 subsystems and 113 blocks in total. The resulting Lustre program is 718 lines long and is generated in a few seconds. The Lustre code was verified using the model-checker Lesar [17], against a set of properties provided by Audi. Some of the properties were found false. A more detailed report on the case study (modulo confidentiality constraints) will be provided elsewhere.

4 SCADE/Lustre extensions

The set of extensions we propose aim at relating the SCADE/Lustre program with its implementation on TTA. This is necessary, since the SCADE/Lustre program has a logical-time semantics, whereas the implementation operates in real-time. The extensions allow the user to express what it means for an implementation to be correct and to direct the compiler to some extent. Thus, they facilitate both analysis (checking whether an implementation is correct) and synthesis (automatically building correct implementations).

The extensions do not change the high-level (logical-time) semantics of Lustre. To ensure backward-compatibility, they are provided mainly as annotations (*pragmas*) which can be taken into account or ignored, depending on the version of the compiler used. The extensions follow the *declarative* style of Lustre.

4.1 Extensions

A set of *code distribution* primitives allow the user to specify which parts of the Lustre program are assigned to which TTA node. A set of *timing assumption* primitives allow the user to specify known facts about the implementation, such as what is the period of an external clock, what is worst-case execution time (WCET) of a code block or the transmission time of a message. A set of *timing requirement* primitives allow the user to specify properties that the implementation must satisfy, such as *relative deadlines* of the form "from the moment input x is read until the moment output y is written, at most k time units elapse". We give some examples of primitives and their usage in what follows.

4.1.1 Code distribution: The annotation $\texttt{location} = P$, where P is the name of a node in the distributed platform, is used to specify that a particular code block must be executed on P (at every instant). For example,

$$\begin{array}{ll} x = f(y) & (\texttt{location} = P) \\ y = g(z) & (\texttt{location} = Q) \end{array}$$

says that x must be computed on P and y on Q. Note that this implies that y must be transmitted from Q to P, before computation of x can start.

At the moment, we assume that `location` annotations are placed only in the body of the main node of a SCADE/Lustre program. They can annotate any kind of equation, including calls to other nodes, and they imply that the entire computation must be performed in the specified location. Location annotations are also placed on declarations of input variables, to specify on which location the input variable is sampled.

4.1.2 Basic clock period and predefined periodic clocks: The annotation

$$(\texttt{hyp}) \quad \texttt{basic\_period} = p$$

declares that the period of the basic clock is p time units. This is a timing assumption. Time units are implicit, but they have to be consistent throughout all declarations of timing assumptions.

The primitive $\texttt{periodic\_cl}(k, p)$, where k, p are integer constants, $k \geq 1$ and $p \geq 0$, defines a sub-clock of the basic clock, with (logical-time) period k and initial phase p. That is, $\texttt{periodic\_cl}(k, p)$ is a boolean flow which is false for the first p instants and then becomes true once every k instants. For example,

$$\texttt{periodic\_cl}(2, 0) = \mathit{true}, \mathit{false}, \mathit{true}, \mathit{false}, \cdots$$

Note that such flows can already be defined in Lustre (e.g., see the definition of `clock1over2` in Section 3). However, the `periodic_cl` primitive is more than just a shorthand. It helps the compiler identify the different periods in a *multi-periodic* application, and to use this information for scheduling (see Section 5).

4.1.3 Execution times: The primitive `exec_time` is used to specify assumptions on the execution times of Lustre operators. It can be used either on a basic operator (e.g., $+, -, *$) or a composite operator (called a Lustre node). If A is an operator, the declaration

$$(\texttt{hyp}) \quad \texttt{exec\_time}(A) \ \texttt{in} \ [l, u]$$

states that the execution of A (assuming no interruptions) takes at least l and at most u time units.

It should be noted that we assume that lower and upper bounds on execution times are *given*, that is, we do not compute them. Computing such bounds is a challenging issue by itself. Techniques such as, for instance, those of [11] can be used for this purpose.

4.1.4 Deadlines: Deadlines are timing requirements. They are specified using the primitive `date`. The expression $\texttt{date}(x)$, where x is a flow, denotes the time the value of x becomes *available* in a given instant, during program execution. An input (respectively, internal, output) variable becomes available when it is read (respectively, computed, written). Constants are available at the beginning of an instant. Then, assuming x is an input and y an output variable, the declaration

$$(\texttt{req}) \quad \texttt{date}(y) - \texttt{date}(x) \leq 5$$

states that y should be written at most 5 time units after x is read.

Execution priorities: The primitive `after` is used to add dependencies between Lustre expressions. This is useful when the programmer wants to enforce execution priorities between readings, computations or writings. The expression

$$\mathit{exp}_1 \ \texttt{after} \ \mathit{exp}_2$$

needs both exp_1 and exp_2 to be computed and outputs the value of exp_1. For example, in the program

```
x = f(u);
y = g(v after x);
```

the variable y will be computed after x.

A dependency between expressions which are not alive at the same instants does not make sense. Therefore, the arguments of `after` must have the same clock. This is checked at compile-time in the same way as for an arithmetical expression like $x + y$. As usual, the dependencies induced by `after` are also checked by the compiler for absence of cycles.

4.2 Analysis

The purpose of analysis is to check whether an implementation is correct. Correctness, in our case, means respecting the timing requirements, as well as the standard logical constraints (e.g., respecting the partial order defined by data dependencies). Of course, it is very hard (if not impossible) to check whether the actual implementation is correct: this would imply checking every possible execution and for each execution, observing the availability times of variables and checking that the timing requirements are met.

Instead, correctness can be checked on a *model* of the implementation. Such a model can be obtained by merging three parts: (1) an "untimed" part which models the execution of the compiled code, (2) a "timed" part which models the timing assumptions, and (3) a (timed) part which models the environment. Since parts (1) and (3) generally deal with infinite-domain data (e.g., integers), we need finite-state abstractions. Tools that perform such abstractions and build finitary models are already available as part of the Lustre tool suite. For instance, Lesar [17] generates a finite-state model of a Lustre program by considering only the boolean variables, and then uses BDD-based techniques for model-checking. One of the goals of our project is to integrate Lesar with the tool Kronos [5], which is a model-checker of *timed automata* [1]. A similar approach has been successfully taken in the project Taxys [4], for checking timing requirements on Esterel [2] programs.

4.3 Synthesis

Analysis is currently possible only for uni-processor applications, since the current Lustre compiler produces uni-processor implementations (tools like Lesar assume such an implementation in building their models). Even if multi-processor compilation and modeling were available, the compiler has many choices when generating code. To see this, consider a trivial example. The Lustre program

$$x = f(u)$$
$$y = g(w)$$

admits two possible implementations:

implementation 1	implementation 2
$x := f(u);$	$y := g(w);$
$y := g(w);$	$x := f(u);$

Which one should the compiler choose? Currently the choice is done arbitrarily. However, it greatly affects the properties of the implementation, in particular with respect to the timing constraints. In the above example, if the time to execute f is long and there is a short deadline associated with y, implementation 2 is clearly preferable. The objective of synthesis is to aid the compiler in choosing correct implementations. In our case, synthesizing correct implementations of Lustre programs on TTA can be reduced to a *multi-period, multi-processor scheduling* problem, as is shown in Section 5.

5 SCADE/Lustre to TTA

For implementation onto TTA, we start with a SCADE/Lustre program extended with the additional information presented in the previous section. In particular, we assume, for the time being, that the allocation of SCADE/Lustre nodes to TTA nodes is given, so that the compiler does not have to decide this part. In Section 5.1, we present the general compilation scheme. An important part of this scheme is the scheduler, which solves a multi-period, multi-processor scheduling problem, taking into account the particular constraints imposed by the TTA bus. Our scheduling techniques are presented in Section 5.2.

5.1 Compilation scheme

The classical compilation of a Lustre node produces a single *step-function* with an execution context, which passes inputs and outputs and stores the program state. At every "tick" of the basic clock the inputs are written into the context, the step-function is called, and the outputs are written out. The compilation into one monolithic step-procedure is not suitable for the TTA architecture. The solution is to generate several Lustre modules which will run on different TTA processors and will exchange messages via the TTA bus. Special "glue" module will be generated to coordinate and interface different modules running on the same processor. The compilation scheme is shown in Figure 8.

In a first step the Analyzer (after performing type checking, clock checking, etc) builds the syntax tree and global partial order. Then, it partitions the partial order to obtain a coarser graph. This is done according to boundaries defined by the annotations of Section 4 (for example, an event associated with a deadline is a good point for partitioning). The nodes of this coarser graph represent *tasks* for the scheduling problem. Some tasks correspond to computations (i.e., the execution of the C code generated for the corresponding piece of Lustre code), others to input readings or output writings, and others to the exchange of messages between TTA nodes. The tasks are related with a precedence relation (direct output of the partial order). They have variable execution times (according to the `exec_time` annotations) and different periods (computed by diving the period of the basic clock according to predefined clocks). Their start and end times are constrained by relative deadlines (according to `date` constraints). This gives rise to a scheduling problem, which is given by the Analyzer to the Scheduler.

The Scheduler solves the scheduling problem (or declares that no solution exists). A solution consists in a bus schedule (MEDL) plus a set of schedules, one for each TTA node. A TTA node schedule is time-triggered: task execution is triggered either by the ticks of TTA round counter or by a finer TTA clock, counting *macro-ticks* within a round. If no solution is found, it may be that the partition in tasks is too coarse. In this case, a finer partition is found and the process is repeated.

Based on the Scheduler results, the Integrator assembles the Lustre code corresponding to adjacent tasks in the schedule and constructs a set of Lustre modules. It also automatically generates the "glue code" interfacing these modules. The following simple example illustrates the process of compilation on one processor.

```
(hyp) basic_period = 10
(hyp) exec_time(A) in [3,5]
(hyp) exec_time(B) in [7,8]
(hyp) exec_time(B1) in [2,3]
(hyp) exec_time(B2) in [3,4]

(req) date(x) - date(u) <= 6

x = A(u);
y = B(u when periodic_cl(2,0), 0.0 -> pre(y));
```

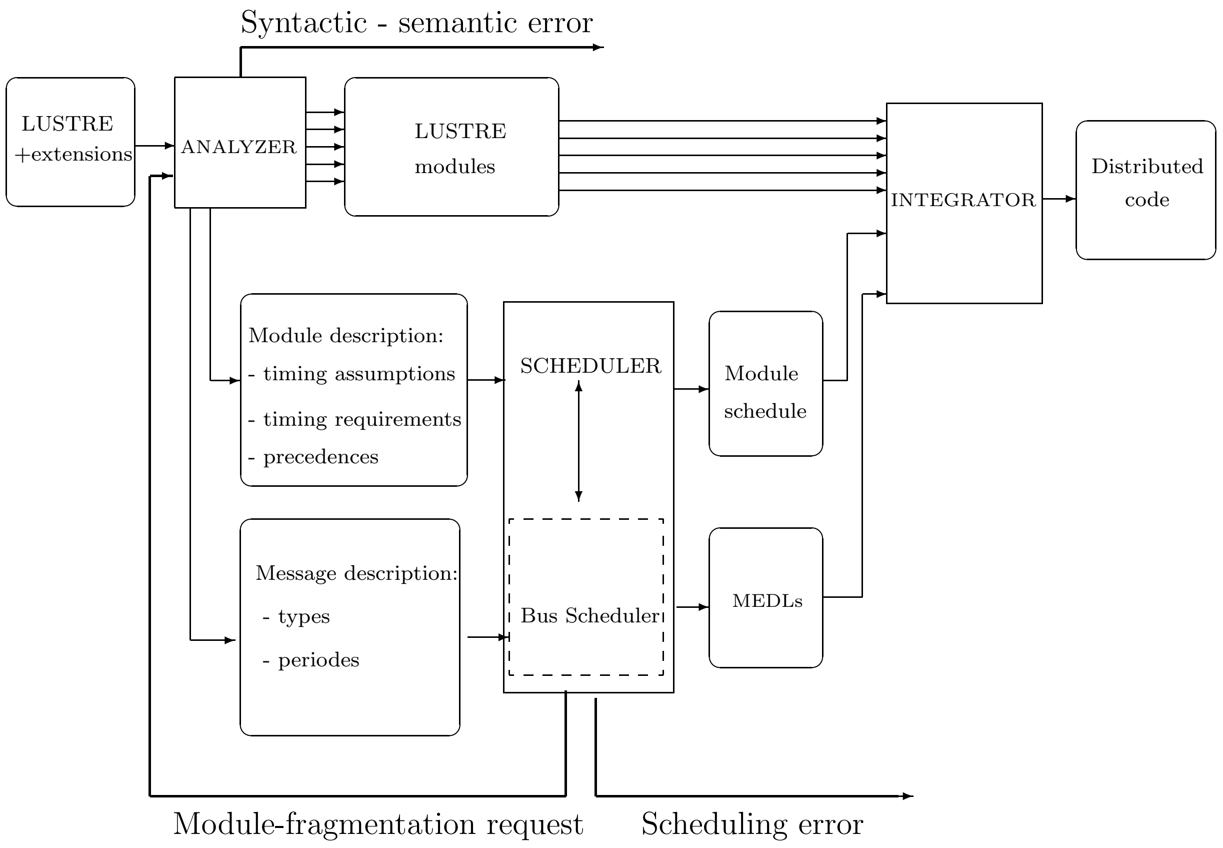

Figure 8: The compilation scheme

In the program, node A works at the basic clock, with period 10. Node B works at a slower clock (defined by `periodic_cl`$(2,0)$) with period twice the basic period. We assume that B calls two sub-nodes, B_1 and B_2, with execution times as shown above. Even though nodes A, B are independent, the deadline requirement implies that node A must be executed before B, so that x is produced at most 6 time units after u becomes available. However, the entire cycle (10 units) is not long enough to contain both A and B (max execution time: $5+8=13$). Thus, as a result of the Scheduler, B is "split" in two modules, executing B_1 and B_2, respectively, as shown in Figure 9.

The final step is to integrate the two Lustre modules by generating the glue code. The glue code is itself written in Lustre, as shown below. z is used as an intermediate variable to carry the result from B_1 to B_2. The values of z are meaningful only at odd instants $(1, 3, ...)$ while the values of y are meaningful only at even instants. It is assumed that an instant corresponds to the beginning of a TTA round.

```
node glue(u: real) returns (x, y: real) ;
var z: real;
let
  oddtick = periodic_cl(2,0) ;
  x = A(u) ;
  z = if oddtick then B1(u, 0.0 -> pre(y))
                 else 0.0 -> pre(z) ;
  y = if oddtick then 0.0 -> pre(y)
                else B2(u, pre(z)) ;
tel.
```

5.2 Scheduling

We now turn to the scheduling problem. The input to the Scheduler is a set of tasks. Each task i is allocated to a processor, loc_i, and has a variable execution time in $[l_i, u_i]$, where l_i, u_i are non-negative constants. Tasks have precedences (partial order). Preemption is not allowed. Each task is also associated with a period and the objective is to find a schedule such that a set of relative deadlines is satisfied.

Tasks model computation units and messages. A special processor models the TTA bus. The problem is not a standard scheduling problem, because of the relative deadlines and the bus periodicity constraints.

In what follows, we first define the problem formally. Then, we discuss the solution technique for the simple case where the execution times are fixed, there is only one processor and a single period, and the TTA bus constraints are not taken into account. We then show how the technique extends to the general case.

5.2.1 Definition of the scheduling problem:

Let s, e be the vectors of start and finish times of tasks. Thus, s_i is the start time of task i, e_j is the finish time of task j, and so on. We will denote by $t_i = e_i - s_i$ the duration of task i, and t the corresponding vector for all tasks.

The scheduling problem is formally defined using a set of constraints. There are three types of constraints:

- Execution time constraints $\mathcal{E}(t)$: a conjunction of inequalities $l_i \le t_i \le u_i$, for each i.
- Precedence, mutual-exclusion and TTA bus constraints $\mathcal{C}(s,t)$:
  - Precedence: if $i \to j$ in the partial order then $e_j \le s_i$ in the set $\mathcal{C}(s,t)$.
  - Mutual-exclusion: if $loc(i) = loc(j)$ then $e_j \le s_i \vee e_i \le s_j$ in the set $\mathcal{C}(s,t)$.
  - TTA bus: these constraints can also be expressed in a linear form with disjunctions, however, they are more involved, thus, we do not attempt to present them here.

  Note that $\mathcal{C}(s,t)$ is not convex, because of the disjunctions in the mutual-exclusion constraints.
- User requirements $\mathcal{R}(s,t)$. This is a conjunction of relative deadlines of the form $\alpha_i - \alpha_j \le d_{ij}$, where $\alpha_i = e_i$ or $\alpha_i = s_i$ and d_{ij} is a constant and $i \neq j$.

The problem is to find a scheduling *strategy*, expressed as a set of constraints $\mathcal{S}(s,t)$, such that

$$\begin{aligned}&\forall s,t:\\&\Big(\mathcal{S}(s,t) \Rightarrow \mathcal{C}(s,t)\Big)\wedge\\&\Big(\mathcal{E}(t) \Rightarrow \mathcal{S}_{/t}(t)\Big)\wedge\\&\Big(\mathcal{S}(s,t)\cap\mathcal{E}(t) \Rightarrow \mathcal{R}(s,t)\Big),\end{aligned} \tag{7}$$

where $\mathcal{S}_{/t}(t) \equiv \exists s.\mathcal{S}(s,t)$. The first conjunct says that the strategy should respect the dependency and mutual-exclusion constraints. The second conjunct says that the strategy should not try to restrain the execution times. The third conjunct says that the user requirements must be met.

The strategies can be classified as *static or dynamic*. In static strategies, all decisions are taken off-line, while in dynamic strategies, decisions can be modified at run-time (e.g., when a task finishes, we can re-consider which task to execute next). Dynamic strategies are not implementable in TTA. Static strategies can be classified as *untimed or timed*:

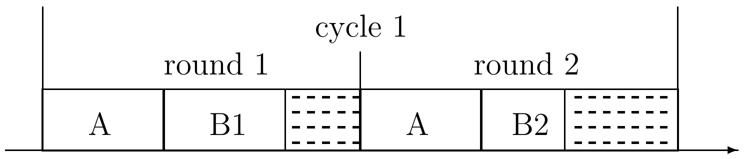

Figure 9: Module scheduling on one processor

- Static (untimed): a total order for each processor is chosen and a task is started as soon as possible (i.e., as soon as all its predecessor tasks in the partial order have finished).
- Static timed: the start time s_i of each task i is predefined.

In this paper, we only consider static timed strategies.

5.2.2 The scheduling technique: We use a Branch-and-Bound (BB) search which starts from the partial order at the root and produces a total task order at the leaves. The novelty is the use of strong necessary conditions which are tested at each step of the BB algorithm. If the necessary conditions are not satisfied then the BB sub-tree is pruned and we backtrack. Once a total order is found, a linear program (LP) is solved in order to find the strategy (start vector s). The LP may also fail, in which case we backtrack.

It should be noted that, although Equation 7 contains disjunctions and universal quantifiers, once a total order on the tasks is fixed, the equation is equivalent to a convex polyhedral set, which can be expressed as a set of linear constraints and solved using an LP solver. The transformation consists at replacing all "positive" occurrences of t_i in Equation 7 by u_i and all "negative" occurrences by l_i. For example, $r_{ij} \leq (s_i + t_i) - (s_j + t_j) \leq d_{ij}$ is equivalent to $(s_i + u_i) - (s_j + l_j) \leq d_{ij} \wedge (s_j + u_j) - (s_i + l_i) \leq -r_{ij}$.

5.2.3 Necessary conditions: We want to detect as soon as possible infeasible orders. This is done with the help of an iterative procedure which manipulates (restricts) the set of possible start times of each task (called *due-date* set) and the absolute or relative deadline intervals (absolute meaning with respect to time zero). The procedure checks that

- the end time of a task is smaller than its (absolute) deadline;
- the sum of all computation times does not exceed the maximum deadline;
- no cycle occurs in the transitive closure;
- the due-date sets are not empty;
- the relative deadline intervals are not empty;
- no time interval is assigned to more than one task.

The set of constraints can be enriched recursively as follows:

- Relative deadline intervals can be reduced by transitive closure according to the partial order.
- Due-date intervals can also be reduced by combining the partial order with execution times, relative deadlines and other task due-dates. Then, absolute deadlines can be reduced accordingly.
- If two due-date intervals become disjoint then a new precedence is induced between the corresponding tasks.

The set of constraints is strengthened at each step of the BB search, thus allowing to prune more branches and reach a total order faster.

5.2.4 Multi-processor generalization: The BB search is now used to find a total task order on each processor (which is a restriction of the global partial order). At each step, two tasks are ordered on a processor, say P, and the necessary conditions are applied on P. If a modification of a due-date can influence the parameters of another task in another processor, say Q, then the necessary conditions are applied on Q and the modifications are processed. In that way, the conditions between processors are taken into account.

Many different strategies exist for ordering tasks during the BB search. Two of them are described in what follows. Their efficiency depends on the type of constraints we have.

(1) Processor-by-processor method: here, a total order is first found on one processor, before proceeding to the next processor. If the relative deadlines between tasks within processors are strong and if there are few relative deadlines which link tasks of different processors then this is a good approach. The set of tasks which must be scheduled first is the one which is the most constrained. During the construction of the total order of one processor, new constraints appears for some tasks

of others processors and the necessary conditions test whether those new constraints lead to a non-feasible problem.

(2) Rank method: here, we alternate the choice of processors. If the relative deadlines are important between tasks of different processor, this method is usually better. We place the first tasks on each processors, the second ones and so one until we have a total order for each processor. Because the constraints between processors are strong, the position of one task deeply affects other tasks in other processors.

5.2.5 Multi-period generalization: If the tasks do not have the same period we consider the least common multiple (LCM) of the set of periods as the working period and for each task of periodicity p we make $\frac{LCM}{p}$ copies and their corresponding precedences and relative deadlines. Moreover, new constraints are added in the due-date intervals in order to satisfy the release time constraints of the new tasks.

5.2.6 TTA bus constraints: As we have seen before, the TTA bus is periodic and a processor can transmit only once per round. To take into account TTA bus constraints, we consider messages as tasks which must be computed on a special processor: the bus. The messages have the periodicity of their sending tasks. The duration of a round is taken to be equal to or a divisor of the GCD of all periods. The duration of the cycle is taken to be equal to the least common multiple of all periods. Precedence constraints are built between sender and receiver of a message. Moreover, messages sent by the same processor are grouped in order to form a slot. In practice, this is achieved thanks to relative deadlines between messages. During the construction of the total order for every processor (except the bus) the necessary conditions enrich the due-dates of messages. Moreover, for the sake of periodicity of the bus, message due-dates propagate from a round to the following ones.

6 Conclusions

We have presented an end-to-end layered approach for the design and implementation of embedded software on a distributed platform. It is worth noting that our tools, although conceived as part of the above chain, can be used independently, as stand-alone tools or in other design chains. Sim2Lus, the Scheduler and the extended Lustre parser are available upon request to the authors. The Analyzer and Integrator are still under development. Apart from completing these tools, future work includes extending our approach to Stateflow, enhancing the capabilities of the Scheduler so that the allocation of tasks to processors is not fixed, and automatizing the interaction between the Scheduler and the Analyzer (module fragmentation request).

Regarding related work, a number of approaches exist at various levels of the design chain, but very few are end-to-end. [19] report on an approach to co-simulate discrete controllers modeled in the synchronous language Signal [6] along with continuous plants modeled in Simulink. [18] use a model-checker to verify a Simulink/Stateflow model from the automotive domain, however, they translate their model manually to the input language of the model-checker.

[9] report on the *SPI workbench*, a framework for modeling, analysis and synthesis of embedded system, based on an abstract model of concurrent processes communicating with FIFO queues or registers, called the *SPI model*. The authors report on translating Simulink to SPI, given an (externally provided) analysis of execution times of Simulink blocks. It is not clear whether the translation is manual or automatic. The focus of the authors seems to be the timing analysis of Simulink models.

Giotto [8] is a time-triggered programming language, similar in some aspects to SCADE/Lustre. The main differences is the logical-time semantics of SCADE/Lustre versus real-time semantics of Giotto, and the fact that Giotto compilation is parameterized by a run-time scheduler, while in SCADE/Lustre scheduling is done once and for all at compile-time. MetaH [3] is an architecture description language and associated tool-suite. It uses an "asynchronous" model based on the Ada language, and real-time scheduling techniques such as rate-monotonic scheduling [15] to analyze the properties of the implementation.

Annotations of programming languages with external information, as we propose here for Lustre, is also sometimes undertaken in *aspect-oriented programming* approaches (e.g., as in [16]).

Naturally, our work on scheduling is related with the vast literature on job-shop-like scheduling or real-time scheduling. However, we could not find scheduling techniques that can deal with relative deadlines, as we do here. Another originality of our scheduling problem is the periodicity constraints imposed by the TTA bus.

Acknowledgments: We would like to thank Thierry Le Sergent and Jean-Louis Colaco from Esterel Technologies, Dietmar Kant from Audi, Reinhard Maier from TTTech and Guenther Bauer from TU Vienna.

References

[1] R. Alur and D.L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[2] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[3] P. Binns and S. Vestal. Scheduling and communication in metah. In *Real-Time Systems Symposium*, Raleigh-Durham NC, December 1993.

[4] E. Closse, M. Poize, J. Pulou, J. Sifakis, P. Venier, D. Weil, and S. Yovine. Taxys: a tool for the development and verification real-time embedded systems. In *CAV'01*. LNCS 2102, Springer, 2001.

[5] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool kronos. In *Hybrid Systems III, Verification and Control*, volume 1066 of *LNCS*, pages 208–219. Springer-Verlag, 1996.

[6] P. Le Guernic, T. Gautier, M. Le Borgne, and C. Le Maire. Programming real-time applications with signal. *Proc. of the IEEE*, 79(9):1321–1336, 1991.

[7] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. *Proceedings of the IEEE*, 79(9), September 1991.

[8] T.A. Henzinger, B. Horowitz, and C. Meyer Kirsch. Giotto: A time-triggered language for embedded programming. In *EMSOFT'01*. Springer, LNCS 2211, 2001.

[9] M. Jersak, D. Ziegenbein, F. Wolf, K. Richter, and R. Ernst. Embedded system design using the spi workbench. In *Proc. of the 3rd International Forum on Design Languages*, 2000.

[10] G. Kahn. The semantics of a simple language for parallel programming. In *Proc. of the IFIP Congress*, 1974.

[11] R. Kirner, R. Lang, G. Freiberger, and P. Puschner. Fully Automatic Worst-Case Execution Time Analysis for Matlab/Simulink Models. In *Proceedings of the 14th Euromicro International Conference on Real-Time Systems*, pages 31–40, Vienna, Austria, June 2002. Vienna University of Technology, IEEE.

[12] H. Kopetz. *Real-Time Systems Design Principles for Distributed Embedded Applications*. Kluwer, 1997.

[13] H. Kopetz and G. Grunsteidl. TTP – a protocol for fault-tolerant real-time systems. *IEEE Computer*, pages 14–23, January 1994.

[14] E.A. Lee and T.M. Parks. Dataflow process networks. *Proceedings of the IEEE*, pages 773–799, May 1995.

[15] C.L. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, January 1973.

[16] M. Mousavi, M. Reniers, T. Basten, M. Chaudron, G. Russello, A. Cursaro, S. Shukla, R. Gupta, and D.C. Schmidt. Using Aspect-GAMMA in the design of embedded systems. In *Proc. of 7th Annual IEEE International Workshop on High Level Design Validation and Test*. IEEE Computer Society Press, 2002.

[17] C. Ratel, N. Halbwachs, and P. Raymond. Programming and verifying critical systems by means of the synchronous data-flow programming language Lustre. In *ACM-SIGSOFT Conference on Software for Critical Systems*, 1991.

[18] S. Sims, K. Butts, R. Cleaveland, and S. Ranville. Automated validation of software models. In *ASE*, 2001.

[19] S. Tudoret, S. Nadjm-Tehrani, A. Beneviste, and J.-E. Stromberg. Co-simulation of hybrid systems: Signal-Simulink. In *FTRTFT*, 2000.