Centre Equation
2, avenue de VIGNATE
F-38610 GIERES
tel : +33 476 63 48 57
fax : +33 476 63 48 50
http://www-verimag.imag.fr

Translating Discrete-Time Simulink to Lustre

Paul Caspi, Adrian Curic, Aude Maignan, Christos Sofronis, Stavros Tripakis

July 2003

Abstract

We present a method of translating discrete-time Simulink models to Lustre programs. Our method consists of three steps: type inference, clock inference and hierarchical bottom-up translation. In the process, we formalise typing and timing mechanisms of Simulink. The method has been implemented in a prototype tool called S2L. The tool has been used to translate part of an industrial automotive controller provided by Audi.[1]

[1] Matlab, Simulink and Stateflow are Registered Trademarks of MathWorks, Inc. SCADE and Simulink Gateway are Registered Trademarks of Esterel Technologies, SA.

This work has been supported in part by European IST projects "NEXT TTA" under project No IST-2001-32111 and "RISE" under project No IST-2001-38117.

A shorter version of this report will appear in the proceedings of EMSOFT03.

Contents

1 Introduction

Model-based design has been established as an important paradigm in the development of embedded systems today. The main principle of the paradigm is to use models all along the development cycle, from design to implementation. Using models, rather than, say, building prototypes, is essential for keeping the development costs manageable. However, models alone are not enough. They need to be accompanied by powerful reasoning tools: simulation, verification, synthesis, code generation, and so on. Automation here is the key: high system complexity and short time-to-market make model reasoning a hopeless task, unless it is largely automatised.

In previous work [4] we have proposed a design approach in three layers (Simulink, Lustre, TTA) and sketched how passing from one layer to another can be automatised. In this paper, we focus on the first two layers, Simulink and Lustre. We describe in detail a technique for translating discrete-time Simulink models into Lustre programs. In the process, we formalise important parts of the semantics of Simulink, such as typing and timing.[2]

The motivation for using Simulink as a high-level design model comes from the fact that this language has become a de-facto standard in many application domains, such as automotive control. We assume that the designer uses Simulink blocks to design a controller. The designer can benefit from Simulink's simulator and graphical capabilities to check the correctness of the controller and measure its performance, in closed-loop with a plant, also modelled in Simulink.

As the first implementation step, we propose the translation of the Simulink model into the synchronous language Lustre [9]. There are a number of reasons motivating this choice. Lustre has been conceived primarily as a programming language, in particular for critical applications, and has several features desirable in such applications, such as formal semantics, strong typing and modularity. A number of formal validation tools exist for Lustre, such as the model-checker Lesar [17], the tester Lurette [18], plus a number of other tools for simulation, static-analysis and controller synthesis. Regarding implementation, a number of techniques and tools exist for generating C code from Lustre programs. In particular, SCADE, the graphical version of Lustre commercialised by Esterel Technologies, is endowed with a DO178B-level-A code generator which allows it to be used in highest criticality applications. SCADE has been used in important European avionic projects (Airbus A340-600, A380, Eurocopter) and is also becoming a de-facto standard in this field.

Simulink, on the other hand, started purely as a simulation environment and lacks many desirable features of programming languages. Simulink has a multitude of semantics (depending on user-configurable options), informally and sometimes partially documented. Although commercial code generators exist for Simulink (Real-time workshop from Mathworks, TargetLink from dSpace) these present major restrictions.

[2] It can be claimed that "Simulink has no semantics". We take a different view, namely, that Simulink has many semantics (depending on user-configurable options), they are not formal, but can be defined as "what is given by the simulator" (e.g., what is observed in terms of inputs/outputs).

For example, TargetLink does not generate code for blocks of the “Discrete” library of Simulink, but only for blocks of the dSpace library, and currently handles monoperiodic systems. Another issue not addressed by these tools is the preservation of semantics. Indeed, the relation between the behaviours of the generated code and those of the simulated model is unclear. Often, speed and memory optimisation is given more attention than semantic consistency.

In the rest of the paper, we present our translation method. Section 2 contains a more technical discussion of the main differences between Simulink and Lustre and states the main goals and limitations of our translation. In Section 3 we give a short description of Lustre. We assume that the reader is more or less familiar with Simulink (see www.mathworks.com). Important Simulink features are discussed in the corresponding sections of the paper. Sections 4, 5 and 6 present the translation method, which consists of three steps: type inference, clock inference and hierarchical translation. The algorithms have been implemented in a prototype tool presented in Section 7. The tool has been used to translate an industrial Simulink model, provided by Audi, presented in Section 8.

Related work

[22] report on an approach to co-simulate discrete controllers modelled in the synchronous language Signal [8] along with continuous plants modelled in Simulink. [6] present tools for co-simulation of process dynamics, control task execution and network communication in a distributed real-time control system. [20] 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.

A number of approaches are based in extending Simulink with libraries of predefined blocks and then using Simulink as a front-end or simulator. The hybrid-system model-checker CheckMate uses such an approach [19, 7]. [13] extend Simulink with the capability of expressing designs in the time-triggered language Giotto. [1] translate hybrid automata into Simulink and use its advanced numerical integration algorithms and user interface to perform simulation.

[10] report on translating Simulink to the *SPI model*, a model of concurrent processes communicating with FIFO queues or registers. The focus seems to be the preservation of value over-writing which can occur in multi-rate systems when a “slower” node receives input from a “fast” one.

[11] report on MAGICA, a type-inference engine for Matlab. The focus is on deriving information such as whether variables have real or imaginary values, array sizes for non-scalars, and so on.

[21] gives a formal semantics to Stateflow models using communicating pushdown automata. [16] presents the data model of Simulink and Stateflow as a UML class diagram.

2 The goals of the translation

Simulink and Lustre are models manipulating *signals* and *systems*. Signals are functions of time. Systems take as input signals and produce as output other signals. In Simulink, which has a graphical language, the signals are the "wires" connecting the various blocks. In Lustre, the signals are the program variables, called *flows*. In Simulink, the systems are the built-in blocks (e.g., adders, gains, transfer functions) as well as composite blocks, called *subsystems*. In Lustre, the systems are the various built-in operators as well as user-defined operators called *nodes*.

In the sequel, we use the following terminology. We use the term *block* for a basic Simulink block (e.g., adder, gain, transfer function) and the term *subsystem* for a composite Simulink block. We will use the term *system* for the root subsystem. We use the term *operator* for a basic Lustre operator and the term *node* for a Lustre node.

	Simulink	Lustre
Signals	"wires"	flows
Systems	adder, gain, unit delay, ..., subsystems	$+$, `pre`, `when`, `current`, ..., nodes

2.1 Differences of Simulink and Lustre

Both Simulink and Lustre are dataflow-like languages.[3] They both allow the representation of *multi-periodic sampled systems*. However, despite their similarities, they also differ in many ways:

- Lustre has a discrete-time semantics, whereas Simulink has a continuous-time semantics. It is important to note that even the "*Discrete library*" Simulink blocks produce piecewise-constant continuous-time signals.[4]

- 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 integration solver and rejected with a fixed-step solver.

- Lustre is a strongly-typed language with explicit types for each flow. In Simulink, explicit types are not mandatory, although they can be set (e.g., using the *data type converter* block, or an expression such as *single(1.2)* for the constant 1.2 of type *single*).

- Lustre is modular in certain aspects, while Simulink is not, in the sense that a Simulink model may contain implicit inputs (cf. 5.2).

[3] The foundations of data-flow models were laid by Kahn [12]. Various such models are studied in [14, 15, 5].

[4] 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.

- Hierarchy in Simulink is present both at the definition and at the execution levels. In Lustre, the execution graph is hierarchical (nodes calling other nodes), whereas the definition of nodes is not.

2.2 Translation goals and limitations

The ultimate objective of the translation is to automatise the implementation of embedded controllers as much as possible. We envisage a tool chain where controllers are designed in Simulink, translated to Lustre, and implemented on a given platform using the Lustre C code generator and a C compiler for this platform.

We only translate the discrete-time part of a Simulink model. Concretely, this means blocks of the “*Discrete*” library (such as “Unit-delay”, “Zero-order hold”, “Discrete filter” and “Discrete transfer function”), generic mathematical operators such as sum, gain, logical and relational operators, other useful operators such as switches, and, finally, subsystems or triggered subsystems. Discrete-time Simulink encompasses also blocks, like the “Read” and “Write” blocks, which exhibit side effects: their behaviour then depends on complex characteristics of the Simulink simulation algorithm and look poorly deterministic. As their use can be considered quite unsafe, they are not translated.

Of course, controllers can be modelled in continuous-time as well. This is typically done in control theory, so that analytic results for the closed-loop system can be provided (e.g., regarding its stability). Analytical results can also be provided using the sampled-data control theory.

In any case, the implemented controller must be discrete-time. How to obtain this controller is a control problem which is clearly beyond the scope of this paper.[5]

Other goals and limitations of our translation are the following.

(1) We aim at a translation method that preserves the semantics of Simulink. This means that the original Simulink model and the generated Lustre program should have the same observable output behaviour, given same inputs, modulo precisely defined conditions. Since Simulink semantics depends on the simulation method, we restrain ourselves only to one method, namely, “*solver: fixed-step, discrete*” and “*mode: auto*”. We also assume that the Lustre program is run at the time period the Simulink model was simulated. Thus, an outcome of the translation must be the period at which the Lustre program *shall* be run (see also Section 5).

(2) We do not translate *S-functions* or *Matlab functions*. Such functions are often helpful. On the other hand, they can also create side-effects, which is something to be avoided and contrary to the “functional programming” spirit of Lustre.

(3) As the Simulink models to be translated are in principle controllers embedded in larger models containing both discrete and continuous parts, we assume that for every input of the model to be translated (i.e., every input of the controller) the sampling time

[5] According to classical text-books [2], there are two main ways of performing this task: either design a continuous-time controller and sample it or sample the environment and design a sampled controller.

is explicitly specified. This also helps the user to see the boundaries of the discrete and the continuous parts in the original model.

(4) In accordance with the first goal, we want the Lustre program produced by the translator to type check if and only if the original Simulink model "type checks" (i.e., is not rejected by Simulink because of type errors). However, the behaviour of the type checking mechanism of Simulink depends on the simulation method and the "*Boolean logic signals*" flag (BLS). Thus, apart from the simulation method which must be set as mentioned in (1), we also assume that BLS is on. When set, BLS imposes that inputs and outputs of logical blocks (and, or, not) be of type *boolean*. Not only this is good modelling and programming practice, it also makes type inference more precise (see also Section 4) and simplifies the verification of the translated Simulink models using Lustre-based model-checking tools. We also set the "*algebraic loop*" detection mechanism of Simulink to the strictest degree, which rejects models with such loops. These loops correspond to cyclic data dependencies in the same instant in Lustre. The Lustre compiler rejects such programs.

(5) For reasons of traceability, the translation must preserve the hierarchy of the Simulink model as much as possible. Since Lustre has a textual language and nodes are declared sequentially rather than nested one inside another, the way to preserve hierarchy is by suitable naming.

(6) Current limitations of our tool are that it only handles scalar Simulink signals and does not translate "*virtual blocks*" such as "*Mux*".

It should also be noted that Simulink is a product evolving in time. This evolution has an impact on the semantics of the tool. For instance, earlier versions of Simulink had weaker type-checking rules. We have developed and tested our translation method and tool with Simulink 4.1 (Matlab 6 release 12). All examples given in this report refer to this version as well.

3 A short description of Lustre

A Lustre program essentially defines a set of equations:

$$\begin{array}{rcl} x_1 & = & f_1(x_1, ..., x_n, u_1, ..., u_m) \\ & \vdots & \\ x_n & = & f_n(x_1, ..., x_n, u_1, ..., u_m) \end{array}$$

where x_i are *internal* or *output* variables, and u_i are *input* variables. The variables in Lustre are also called *flows*. A flow is a partial function $x : N \rightarrow V_x$, where N is the set of natural numbers and V_x is the domain of x. N models *logical time*, counted in *instants*, $i = 0, 1, 2, ...$.

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 $i-1$ (at $i=0$, the value of y is undefined).

`->` initialises a flow. If $z = x$`->`y, then the value of z is equal to the value of x for $i=0$ and equal to the value of y for $i \geq 1$. The operator `->` is typically used to initialise a flow obtained using `pre`. For example, a counter is defined by the equation $x = 0$`->`$($`pre`$(x)+1)$.

`when` is used to *sample* a flow: x `when` b, where x is a flow and b is a boolean flow defined at the same instants as x, defines a flow y which is defined only at instants i where $b(i) = \mathtt{t}$ (where t is the boolean value “true” and f is the boolean value “false”). For these instants, $y(i) = x(i)$.

`current` is used to extend the instants where a sampled flow y is defined. If $y = x$ `when` b and $z =$ `current`(y) then z is defined at all instants where x is defined. At all instants i where y is defined, $z(i) = y(i)$. At instants where y is undefined, z keeps its previous value.

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:

```
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). Nodes B and C are declared elsewhere.

4 Type inference

A prior step to translation per se is type inference. In this section, we explain the need for this step and how it is performed. Notice that the type rules for Simulink that are stated here are with respect to the simulation method and flag options mentioned in Section 2.

4.1 Types in Lustre

Lustre is a strongly typed language, meaning that every variable has a declared type and operations have precise type signatures. For instance, we cannot add an integer

Constant$_\alpha$	:	$\alpha, \alpha \in SimNum$
Adder	:	$\alpha \times \cdots \times \alpha \rightarrow \alpha, \alpha \in SimNum$
Gain	:	$\alpha \rightarrow \alpha, \alpha \in SimNum$
Relation	:	$\alpha \times \alpha \rightarrow \mathit{boolean}, \alpha \in SimNum$
Switch	:	$\alpha \times \beta \times \alpha \rightarrow \alpha, \alpha, \beta \in SimNum$
Logical Operator	:	$\mathit{boolean} \times \cdots \times \mathit{boolean} \rightarrow \mathit{boolean}$
Discrete Transfer Function	:	$\mathit{double} \rightarrow \mathit{double}$
Zero-Order Hold, Unit Delay	:	$\alpha \rightarrow \alpha, \alpha \in SimT$
Data Type Converter$_\alpha$	:	$\beta \rightarrow \alpha, \alpha, \beta \in SimT$
InPort, OutPort	:	$\alpha \rightarrow \alpha, \alpha \in SimT$

Table 1: Types of some Simulink blocks.

with a boolean or even an integer with a real.[6] Each flow has a type. Basic types in Lustre are *bool*, *int* and *real*.

4.2 Types in Simulink

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 Lustre flow.

Basic "*data types*" in Simulink are: *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: (1) 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 (2) a signal is used in an operation which demands another type (e.g., all inputs and outputs of *Logical Operator* blocks are *boolean*. A type error occurs when incompatible types are used in an operation (e.g., trying to add a *boolean* or trying to input an explicitly defined *int8* into a *Discrete Transfer Function* block).

We can formalise the above type system as follows. First, denote by $SimT$ the set of all Simulink types and let $SimNum = SimT - \{\mathit{boolean}\}$. Then, the type system of Simulink can be seen as a standard type system with type *classes* ($SimT$, $SimNum$, $\{\mathit{boolean}\}$). Every Simulink block has a (polymorphic) type, given in Table 1.

[6] Predefined *casting* operators such as *int2real* can be used in the latter case.

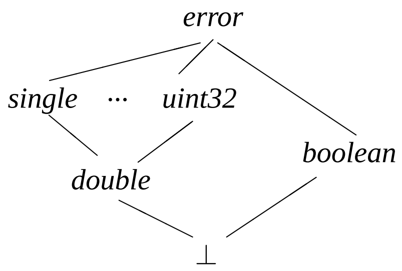

Figure 1: The type lattice

Simulink equation	Type equations
$y = \mathit{Adder}(x_1, ..., x_k)$	$y^T = x_1^T = \cdots = x_k^T = \sup(double, y^T, x_1^T, ..., x_k^T)$
$y = \mathit{Constant}_\alpha$	$y^T =$ if $y^T \leq \alpha$ then α else $error$
$y = \mathit{Data\ Type\ Converter}_\alpha(x)$	$y^T =$ if $y^T \leq \alpha$ then α else $error$
$y = \mathit{Unit\ Delay}(x)$	$x^T = y^T$
$y = \mathit{Zero\text{-}Order\ Hold}(x)$	$x^T = y^T$
$y = \mathit{Transfer\ Function}(x)$	$y^T = x^T =$ if $y^T \leq$ *double* then *double* else *error*
$y = \mathit{Relation}(x_1, x_2)$	$x_1^T = x_2^T = \sup(double, x_1^T, x_2^T), y^T = boolean$
$y = \mathit{Logical}(x_1, ..., x_k)$	$y^T = x_1^T = \cdots = x_k^T = boolean$
$y = \mathit{Switch}(x_1, x_2, x_3)$	$x_1^T = x_3^T = y^T = \sup(x_1^T, x_3^T, y^T)$

Table 2: Type inference equations.

4.3 Type inference

The type inference algorithm is a standard fix-point computation on the lattice shown in Figure 1. $\perp$ is the undefined type and $error$ means type error (model rejected). The fix-point is computed on a set of equations derived according to Table 2. In the table, $x_1, x_2, ..., y$ are variables of the Simulink model and $x_1^T, x_2^T, ..., y^T$ are their corresponding types (which must be inferred by the fix-point computation). For each Simulink equation, the table gives the corresponding type equations. The order $\leq$ and the $\sup$ operator are for the lattice of Figure 1. For example, $double \leq single$ and $\sup(double, boolean) = error$. In the “Switch” block, x_2 is the input deciding (based on a threshold) whether the output y will be set to x_1 or x_3. Thus, the type of x_2 does not influence the types of x_1, x_3, y.

The right-hand sides of equations in Table 2 define a monotonic function in the type lattice. However, we can notice that the equations are not always oriented from inputs to outputs. Thus, it can be the case that several equations apply to the same variable, one when the variable acts as an output and some other ones when it acts as an input. This is not a conventional fix-point system of equations. However we can easily turn it to a conventional one; if several equations apply to the same variable:

$$\begin{array}{rcl} x^T & = & e_1 \\ ... & = & ... \\ x^T & = & e_n \end{array}$$

these equations are replaced by the single one:

$$x^T = \sup(e_1, \ldots, e_n)$$

Thus a least fix-point exists. Computing it takes $O(n^3)$ time, where n is the number of signals in the model. Indeed, there can be at most n equations (one for each basic block, each block having at least one output signal) of n variables. The fix-point is reached in $O(n)$ iterations. In each iteration, at most n equations are examined. Examining each equation takes $O(n)$ time.

Once the inference of Simulink types is performed, these types are mapped to 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*; $\perp$ is mapped to *real*. The latter case is consistent with the fact that the default type in Simulink is *double*. Note that the type of some signals may indeed remain undefined ($\perp$) until the end. For example, this is the case of a system consisting of a single "unit-delay" block.

5 Clock inference

5.1 Time in Lustre

As mentioned in Section 3, Lustre time is logical and is counted in discrete instants. Logical time means there is no a-priori notion of instant duration or time elapsing between two instants. Associated with each Lustre flow x is a boolean flow b_x, called the *clock* of x, and specifying the instants when x is defined: x is defined at instant i iff $b_x(i) = \texttt{t}$. For example, if x is defined at $i = 0, 2, 4, ...$ then $b_x = \texttt{t f t f} \cdots$.

Input variables are by definition defined at every instant: their clock is called the *basic* clock, represented by the boolean flow $\mathit{true} = \texttt{t t} \cdots$. "Slower" clocks are obtained from the basic clock using the `when` operator. For example, if `in` is an input then the flow x defined only at even instants can be generated by the following Lustre code:

```
cl1_2 = true -> not pre(cl1_2) ;
x     = in when cl1_2 ;
```

Clocks can be seen as extra typing information. The compiler ensures that the Lustre program satisfies a set of constraints on clocks, otherwise the program is rejected. For example, in the expression $x + y$, x and y must have the same clock, which is also

expression e	$\mathtt{clock}(e)$	constraints
input x	*basic*	
$x+y$	$\mathtt{clock}(x)$	$\mathtt{clock}(x) = \mathtt{clock}(y)$
$\mathtt{pre}(x)$	$\mathtt{clock}(x)$	
x $\mathtt{when}$ b	b	$\mathtt{clock}(x) = \mathtt{clock}(b)$, b boolean
$\mathtt{current}(x)$	$\mathtt{clock}(\mathtt{clock}(x))$	

Table 3: Clock calculus of Lustre.

the clock of the resulting flow. The set of these constraints and how to calculate clocks is called *clock calculus*. A simplified version of this calculus is shown in Table 3.

It is worth mentioning that the Lustre compiler checks clock correctness in a *syntactic*, not *semantic* manner. Indeed, finding whether two boolean flows are semantically equivalent is an undecidable problem. Therefore, in an expression such as $(x$ $\mathtt{when}$ $b) + (y$ $\mathtt{when}$ $b')$, b and b' must be identical for the expression to clock-check.

5.2 Time in Simulink

Discrete-time Simulink signals are in fact piecewise-constant continuous-time signals. These signals can have associated timing information, called "*sample time*" and consisting of a *period* and an initial *phase*. Sample times may be set in blocks such as input ports, unit-delay, zero-order hold or discrete transfer functions. The sample time of a signal is derived from the block producing the signal and 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 also serve as an extra type system in Simulink: some models are rejected because of timing errors. An interesting example is shown in Figure 2. The sample times of inputs "In1" and "In2" are set to 2 and 3, respectively.[7] This model is rejected by Simulink. However, if the "Gain" block (a simple multiplication by 1) is removed, then the model is accepted! The explanation is given at the end of this section.[8]

Another timing mechanism of Simulink is by means of "*triggers*". Only subsystems (not basic blocks) can be triggered. A subsystem A can be triggered by a signal x (of any type) in three ways, namely, "*rising, falling*" or "*either*", which specify the time the trigger occurs depending on the direction with which x "crosses" zero. The sample time of blocks 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 T. Thus, in the

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

[8] Despite some Reviewers' remarks, the same anomaly persists when using Matlab Version 6.5.0.180913a Release 13 and Simulink version 5.0.1 (R13) dated 19-Sep-2002. But it only happens when setting simulation parameters to "fixed step", "auto".

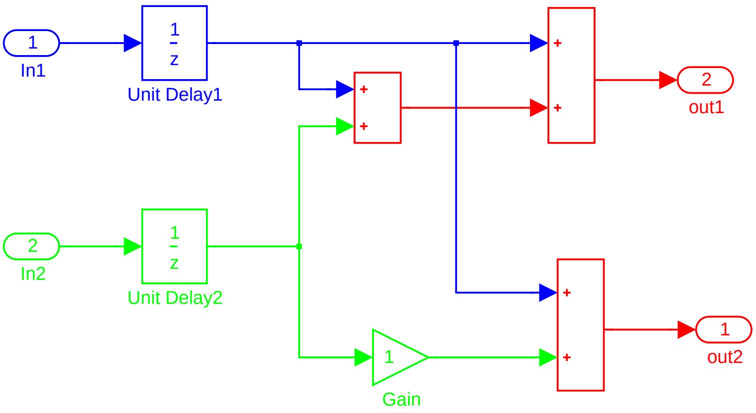

Figure 2: A Simulink model producing a strange error.

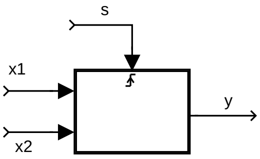

Figure 3: A triggered subsystem.

example shown in Figure 3, the sample times of s, x_1, x_2 and y are all equal.

Timing in Simulink is not as modular as in Lustre, in the following sense. Lustre nodes do not have their own time: they are only activated at instants where their inputs are active. Consequently, a node B called by a node A cannot be active at instants when A is not active. This is true for triggered subsystems in Simulink as well. However, Simulink allows a block inside a (non-triggered) subsystem to have any sample time, possibly smaller than the parent subsystem. Thus, the block can be active while its parent system is inactive. This can be considered a useful mechanism which allows so-called “oversampling”. However, when used in conjunction with the “enabling” mechanism, it can give birth to real monsters whose behaviour is hardly understandable. The basic reason is that an extra, implicit input: the “absolute simulation time” — appears to flow through every block (but the triggered ones), even when this block is not enabled. We consider this a non-modular feature of Simulink.[9]

Another difference lies in how changes of timing are performed in the two languages. In Lustre, this can only be done using the `when` and `current` operators. In Simulink, the sample time of a signal can be changed using the “unit-delay” block or the “zero-order hold” block. In order to do this, however, the following rules must be

[9] Still, we are able to translate such models in Lustre, by calling the parent node with the “faster” clock (on which the child block will run) and passing as parameter the “slower” clock as well.

obeyed:[10]

1. *"When transitioning from a slow to fast rate, a Unit Delay running at the slow rate must be inserted between the two blocks. When transitioning from a fast to a slow rate, a Zero Order Hold running at the slow rate must be inserted between the two blocks."*

2. *"Illegal rate transition found involving Unit Delay block ... When using this block to transition rates, the input must be connected to the slow sample time and the output must be connected to the fast sample time. The Unit Delay must have a sample time equal to the slow sample time and the slow sample time must be a multiple of the fast sample time. Also, the sample times of all destination blocks must be the same value."*

3. *"Illegal rate transition found involving Zero Order Hold block ... When using this block to transition rates, the input must be connected to the fast sample time and the output must be connected to the slow sample time. The Zero Order Hold must have a sample time equal to slow sample time and the slow sample time must be a multiple of the fast sample time. Also, the sample times of all source blocks must be the same value."*[11]

The second rule explains why the model of Figure 2 is rejected when the "Gain" block is present and accepted otherwise. Indeed, after computing the sample times according to the method described in the following section, we find that the output of "Unit Delay 2" has sample time 3. This output is "fed" to the "Gain" block, which also has sample time 3, and to the left-most "Adder" block, which has sample time 1 (the GCD of 2 and 3). This violates the second rule above.

5.3 Clock inference

The objective of clock inference is to compute the period and phase of each Simulink signal and use this information (1) when creating the corresponding Lustre flows and (2) for defining the period at which the Lustre program must be run.

The clock inference algorithm is based on a fix-point computation on a lattice representing the sample times. The lattice is defined to be $ST = (\mathsf{Q} \times \mathsf{Q}) \cup \{\bot\}$, where Q is the set of non-negative rational numbers (of finite precision). A pair (π, θ) represents the sample time with period π and initial phase θ. $\bot$ represents the undefined sample time. The order in this lattice is defined as follows:

$$(\pi_1, \theta_1) \leq (\pi_2, \theta_2) \quad \equiv \quad \exists \kappa . \pi_1 = \kappa \pi_2 \wedge (\theta_1 = \theta_2 \vee (\theta_2 = 0 \wedge \exists \lambda . \theta_1 = \lambda \pi_2)) \quad (1)$$

[10] These rules are quoted from error messages produced by the Simulink tool.

[11] The third rule may seem strange since it implies that a zero-order hold block can have more than one inputs. In fact, this happens when the input to this block comes from a "Mux" block, thus, encoding a vector of signals.

Simulink equation	Sample time equations
$y = Adder(x_1, ..., x_k)$	$y^C = \sup(x_1^C, ..., x_k^C)$
$(y_1, ..., y_l) = \mathit{Triggered}(s, x_1, ..., x_k)$	$s^C = y_1^C = \cdots = y_l^C = x_1^C = \cdots = x_k^C$
$y = \mathit{DTL}_{st}(x)$	$y^C =$ if $st = -1$ then x^C else st

Table 4: Clock inference equations.

The formula is slightly complicated because of the phases. If both phases are zero, then the order reduces to π_1 being a multiple of π_2 (κ and λ are integers). Naturally, we also set $\bot \leq (\pi, \theta)$, for all $(\pi, \theta) \in ST$. For example, $(8, 2) \leq (4, 2) \leq (2, 0) \leq (1, 0)$, but $(4, 1)$ and $(2, 0)$ are incomparable.

The order $\leq$ formalises the term "multiple" in the three Simulink rules quoted in Section 5.2. Thus, sample time s_1 is a "multiple" of sample time s_2 means $s_1 \leq s_2$.

The $\sup$ operator in this lattice corresponds essentially to a *GCD* (greatest common divisor) operation, complicated by the presence of phases. The precise definition is given in the lemma below.

Let $(\pi_i, \theta_i) \in ST$, for $i = 1, 2$. Then $(\pi, \theta) = \sup((\pi_1, \theta_1), (\pi_2, \theta_2))$ exists and is equal to:

$$\pi = \begin{cases} gcd(\pi_1, \pi_2), & \text{if } \theta_1 = \theta_2 \\ gcd(\pi_1, \pi_2, \theta_1, \theta_2), & \text{otherwise} \end{cases} \qquad \theta = \begin{cases} \theta_1, & \text{if } \theta_1 = \theta_2 \\ 0, & \text{otherwise} \end{cases}$$

For example,

$$\sup((2,0),(3,0)) = (1,0) \qquad \sup((12,3),(6,3)) = (6,3)$$
$$\sup((12,6),(12,0)) = (6,0) \qquad \sup((12,3),(12,4)) = (1,0)$$

The fix-point is computed on a set of equations derived according to Table 4. In the table, $x_1, x_2, ...$ are variables of the Simulink model and $x_1^C, x_2^C, ...$ are their corresponding sample-times. The equation for "*Triggered*" refers to the out-most triggered subsystem: we do not need to infer sample times inside a triggered subsystem, since they are all of type "inherited". The equation for "*DTL*" refers to blocks of the "Discrete" library. The user can set the sample time of such blocks, denoted st in the table. The default value of st is -1, meaning the sample time is "inherited" from the input.

Monotonicity ensures existence of a least fix-point. Although ST is an infinite lattice, the fix-point computation terminates. This is because there is a finite number of sample times in a given model. If the smallest fractional part among these sample times is 10^{-k} then it can be shown that the sample time $(10^{-k}, 0)$ is above all others according to the order $\leq$ defined above, thus acts as a "top" element for the given model.

Once the sample times are calculated, a number of rules need to be verified. First, we must verify a strange restriction of Simulink, which forbids sample times (π, θ)

with $\theta \geq \pi$. If such a sample time is produced during the inference procedure, Simulink seems to apply the following procedure: (1) if $\theta \bmod \pi = 0$ (where mod is the modulo operator) then change θ to $\theta \bmod \pi$ (note that this preserves the $\leq$ order); (2) otherwise, reject the model.

Then, the three Simulink rules given in Section 5.2 are checked. This can be done following the structure of the model. For example, in a block F with input x and output y, if $x^C \leq y^C$ (i.e., x is "slower" than y), we must verify that x is the output of a unit-delay block, according to the first rule. In the opposite case, where $y^C \leq x^C$, F must be a zero-order hold block, according to the same rule.

6 Translation

The type and clock inference steps are independent and can be performed in any order. Once this is done, the translation itself is performed, in a hierarchical manner. The Simulink model is organised as a tree, where the children of a subsystem (or system) are the subsystems (or blocks) directly appearing in it. The translation is performed following this hierarchy in a bottom-up fashion (i.e., starting from the basic blocks). Since the Lustre node declarations are not hierarchical, in order to preserve the Simulink hierarchy, we name each Lustre node with the corresponding path of names in the Simulink tree. For example, if a subsystem B contained in a subsystem A will be translated to a Lustre node called "`NameofA_B`" where "`NameofA`" is the name of the Lustre node for A.

Simple Simulink blocks (e.g., adders, multipliers, the $\frac{1}{z}$ transfer function) are translated into basic Lustre operators. For example, an adder is simply translated into $+$ and $\frac{x}{z}$ is translated to the Lustre expression `init ->pre x` where `init` is the initial value specified in the dialog box of the unit delay.

More complex Simulink blocks (e.g., discrete filters) are translated into Lustre nodes. For example, the discrete transfer function $\frac{z+2}{z^2+3z+1}$ is translated into the Lustre node:[12]

```
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 is translated into a Lustre node, possibly containing calls to other nodes. The Lustre node has the same inputs and outputs as the Simulink subsystem, plus, sometimes, the clock of some of its inputs (this is done for modularity).

[12] This translation comes very directly from the translation of the $\frac{x}{z}$ expression, by usual algebraic manipulations.

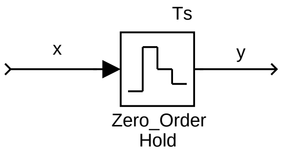

Figure 4: A Zero-Order-Hold block modifying the period of its input.

Here is an example of such a translation. Consider the Simulink model shown in Figure 5, with subsystems A and 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 : real);
...
```

An example where it is necessary to pass clock information as input to the Lustre node is given in Section 8.

Simulink signals are mapped into Lustre flows. When a block changes the sample time of a signal, the appropriate `when` and `current` operations need to be applied. This must also be done for triggered subsystems, as well as in cases where a subsystem receives input signals with different sample times.

For example, consider the Simulink model of Figure 4 and assume that the sample time of input x is 1 and the sample time set to the zero-order hold block is 2. Then, the sample time of the output y is also 2 and the generated Lustre code is as follows:

```
cl1_2 = true -> not pre(cl1_2) ;
y     = x when cl1_2 ;
```

As another example, consider the triggered subsystem of Figure 3. Assume that s and y are of type *double* and that the Lustre node corresponding to the triggered subsystem is called A. Then, the generated Lustre code for calling this subsystem is as follows:

```
s_rise = false -> (s>0 and not pre(s>0)) ;
y1     = A(x1 when s_rise, x2 when s_rise) ;
y      = if s_rise then current(y1) else (0.0 -> pre(y)) ;
```

The boolean flow `s_rise` is true whenever s crosses 0 (“rising edge” trigger) and serves as the basic clock for A. Since the output `y1` of this node also has the same clock, it has to be transformed to a flow `y` having the same clock (i.e., sample time) as `x1` and `x2`. Obtaining `y` from `y1` from `y1` is slightly more complicated than simply writing $y = \texttt{current}(y_1)$ because this would leave y undefined until the first time s rises. The definition shown above solves this problem by initialising y to 0.

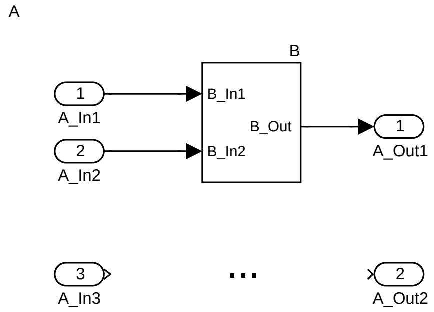

Figure 5: Simulink system A with subsystem B.

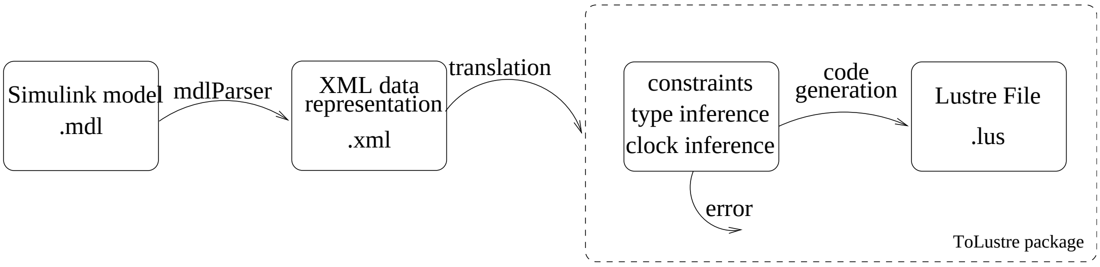

Figure 6: S2L architecture.

7 Implementation: the tool S2L

We implemented the algorithms described above in a prototype tool, called S2L. The tool is written in Java. It takes as input a Simulink model (".mdl" file) and produces a Lustre program (".lus" file) or an error message in case the input model contains type or timing errors. S2L uses an intermediate representation of the Simulink model, in XML. This is done to facilitate evolution of the tool as the syntax of the ".mdl" file changes.

All the above are implemented by a tool named *s2l* — Simulink to Lustre, in Verimag. This tool uses the algorithms described throughout the this paper to generate a faithful translation of a simulink model. *s2l* provides a layered approach using the XML technology as intermediate level of data. The tool 's architecture is depicted in the Figure 6.

8 Case study: Audi's warning processing system

In this section we describe the translation of a Simulink model provided by the automotive constructor Audi. The model defines a warning processing system and is part of a larger controller used by Audi in production vehicles. The controller is itself part

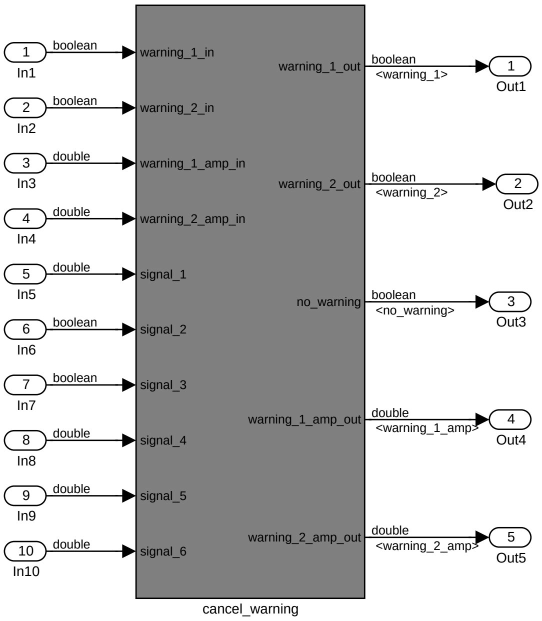

Figure 7: Cancel Warning subsystem.

of a larger steer-by-wire application. The model has been given to Verimag in the context of the European IST project “NEXT TTA”.

Due to intellectual property restrictions, we only present parts of the entire model here. The entire model has a hierarchy depth of 6 layers (including the top-level system). It contains 20 subsystems and a total of 113 components (subsystems and basic Simulink blocks).

The objective of the system is to recognize if a car is moving towards its physical limits and to generate a warning to the driver. However, at several cases it is not necessary to generate this warning, because the car is not in a dangerous situation. This is decided by the main subsystem, called *cancel_warning* and shown in Figure 8.

The *cancel_warning* subsystem has ten inputs and five outputs. Inputs *warning_1_in* and *warning_2_in* are generated on the same Control Unit and are sampled with a rate of 20 ms. Each warning is a boolean, representing an a-priori need to issue a warning. The input signals *warning_1_amp_in* and *warning_2_amp_in* are the amplitudes of the warnings. They are 8 bit signals sampled with a rate of 20 ms. The rest of the input signals (*signal_1*, *signal_2*, . . .) are used to compute the warning permission *no_warning* and they have a sampling rate of 4ms. The outputs of the main subsystem are the warning signals and their amplitudes (with the sampling rate of 20ms) as long as the warning permission (with the sampling rate of 4ms).

The *cancel_warning* subsystem is itself composed of a number of subsystems and basic blocks, as shown in Figure 8. The subsystem *no_warning_conditions* is supposed to check a number of conditions which cancel the warning. Depending on the this

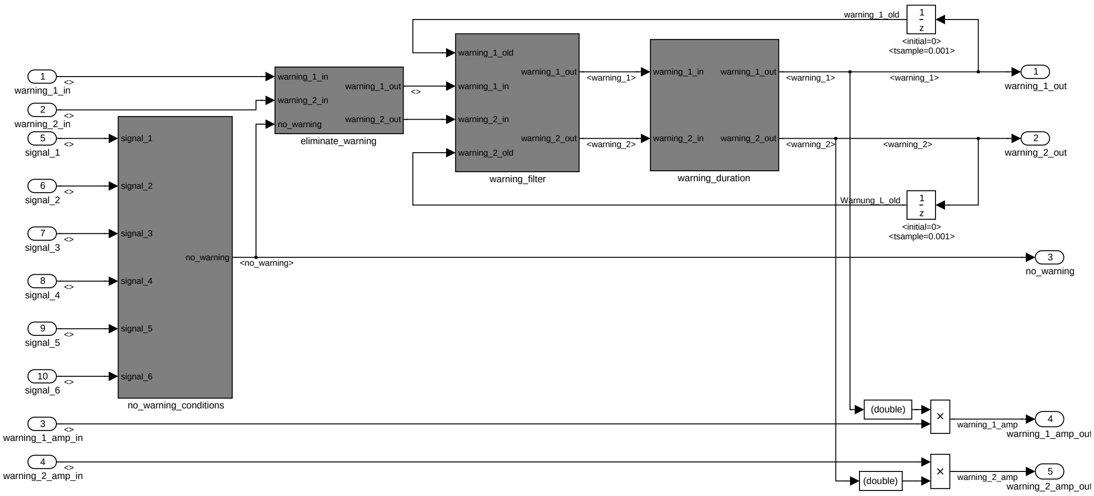

Figure 8: Cancel Warning subsystem contents.

output, the subsystem *eliminate_warning* filters the warning. The subsystem *warning_filter* blocks one of the warning signals if the other one is already active. The subsystem *warning_duration* ensures that a warning which has been issued will be sustained long enough in order to be perceived by the user.

Translating the entire Simulink model to Lustre using the S2L tool takes less than a second. The resulting Lustre program is 718 lines long. A small part of the program is shown in Figure 9. It contains the signature of the Lustre node corresponding to the subsystem *cancel_warning*. This node is not the main node of the Lustre program, but is called from the main node. The assumption is that the Lustre program will be run periodically every 4 ms, which corresponds to a basic clock of the same period. For the inputs sampled at 20 ms, a slower clock is defined (in the main node): the boolean flow `cl1_5`, equal to `t f f f f t f f` $\cdots$. This clock is passed as an input to node `cancel_warning`. Notice that inputs and outputs having a clock different than the basic clock of the node declare this information explicitly in their type.

9 Conclusions

We have presented a method for translating a discrete-time subset of Simulink models into Lustre programs. The translation is done in three steps: type and clock inference, followed by a hierarchical bottom-up translation. We have implemented the method in a tool called S2L and applied it to a controller used in Audi cars. The interest of our tool is that it opens the way to the use of formal and certified verification and implementation tools attached to the Lustre tool chain. Also, in the process of translation, we formalised (for the first time, to our knowledge) the typing and timing mechanisms of Simulink.

```
node cancel_warning( cl1_5:bool;
                     warning_1_in:bool when cl1_5;
                     warning_2_in:bool when cl1_5;
                     warning_1_amp_in:real when cl1_5;
                     warning_2_amp_in:real when cl1_5;
                     signal_1:real;
                     signal_2_n:bool;
                     signal_3:bool;
                     signal_4:real;
                     signal_5:real;
                     signal_6:real )
    returns ( warning_1_out:bool when cl1_5;
              warning_2_out:bool when cl1_5;
              no_warning:bool;
              warning_1_amp_out:real when cl1_5;
              warning_2_amp_out:real when cl1_5 );
```

Figure 9: Cancel Warning signature in Lustre program.

Perhaps the most significant drawback of our approach is its dependency on syntax and semantics of Simulink models. New versions of Simulink appear as often as every six months and sometimes major changes are made with respect to previous versions. This situation seems difficult to avoid given the relative “monopoly” of Simulink/Stateflow in the control design landscape. Another weakness of our tool is its incompleteness: several unsafe constructs of Simulink are not translated. Yet this can be seen as the price to pay for having a sound translation.

We are currently enhancing the capabilities of S2L. Features such as “masked” subsystems are already handled. We are now studying the translation of some “virtual” blocks such as “Mux”. We are also currently studying the translation of Stateflow. Stateflow presents a number of semantic problems, such as behaviour depending on how the model is graphically drawn, or non-termination of a simulation step. Our first goal is to identify a subset of Stateflow avoiding these problems.

References

[1] E. Asarin, T. Dang, and J. Esteban. Simulation of hybrid automata using Matlab/Simulink. Technical report, Verimag, 2003.

[2] K.J. Åström and B. Wittenmark. *Computer Controlled Systems*. Prentice-Hall, 1984.

[3] P. Caspi, A. Curic, A. Maignan, C. Sofronis and S. Tripakis. Translating Discrete-Time Simulink to Lustre. Verimag Research Report, July 2003.

[4] 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 *Languages, Compilers, and Tools for Embedded Systems (LCTES'03)*. ACM, 2003.

[5] P. Caspi and M. Pouzet. Synchronous Kahn networks. In *ACM SIGPLAN International Conference on Functional Programming*, pages 226–238, 1996.

[6] Anton Cervin, Dan Henriksson, Bo Lincoln, and Karl-Erik Årzén. Jitterbug and Truetime: Analysis tools for real-time control systems. In *Proceedings of the 2nd Workshop on Real-Time Tools*, Copenhagen, Denmark, August 2002.

[7] A. Chutinan and B.H. Krogh. Computational techniques for hybrid system verification. *IEEE Trans. Automatic Control*, 48(1), 2003.

[8] 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.

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

[10] M. Jersak, D. Ziegenbein, F. Wolf, K. Richter, R. Ernst, F. Cieslog, J. Teich, K. Strehl, and L. Thiele. Embedded system design using the SPI workbench. In *Proc. of the 3rd International Forum on Design Languages*, 2000.

[11] P.G. Joisha and P. Banerjee. The MAGICA type inference engine for MATLAB. In *Compiler Construction (CC)*. LNCS 2622, Springer, 2003.

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

[13] C.M. Kirsch, M.A. Sanvido, T.A. Henzinger, and W. Pree. A Giotto-based helicopter control system. In *EMSOFT'02*. LNCS 2491, Springer, 2002.

[14] E. Lee and D. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75, 1987.

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

[16] S. Neema. Simulink and Stateflow data model. Technical report, ISIS, Vanderbilt University, 2001.

[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] P. Raymond, D. Weber, X. Nicollin, and N. Halbwachs. Automatic testing of reactive systems. In *19th IEEE Real-Time Systems Symposium*, Madrid, Spain, December 1998.

[19] B.I. Silva, K. Richeson, B.H. Krogh, and A. Chutinan. Modeling and verifying hybrid dynamical systems using CheckMate. In *ADPM*, 2000.

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

[21] A. Tiwari. Formal semantics and analysis methods for Simulink Stateflow models. Technical report, SRI International, 2002.

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