Translating Discrete Time SIMULINK to SIGNAL

Safa Messaoud
University of Illinois at Urbana-Champaign, Department of Electrical and Computer Engineering
messaou2@illinois.edu

Neda Saeedloei
University of Minnnesota Duluth, Department of Computer Science
nsaeedlo@d.umn.edu

Sandeep Shukla
Virginia Polytechnic and Sate University, Department of Electrical and Computer Engineering
shukla@vt.edu

Abstract

As Cyber Physical Systems (CPS) are getting more complex and safety critical, Model Based Design (MBD), which consists of building formal models of a system in order to be used in verification and correct-by-construction code generation, is becoming a promising methodology for the development of the embedded software of such systems. This design paradigm significantly reduces the development cost and time while guaranteeing better robustness and correctness with respect to the original specifications, when compared with the traditional ad-hoc design methods. SIMULINK has been the most popular tool for embedded control design in research as well as in industry, for the last decades. As SIMULINK does not have formal semantics, the application of the model based design methodology and tools to its models is very limited. In this paper, we present a semantic translator that transforms discrete time SIMULINK models into SIGNAL programs. The choice of SIGNAL is motivated by its polychronous formalism that enhances synchronous programming with asynchronous concurrency, as well as, by the ability of its compiler of generating deterministic multi thread code. Our translation involves three major steps: clock inference, type inference and hierarchical top-down translation. We validate our prototype tool by testing it on different SIMULINK models.

Keywords

Cyber Physical Systems, SIMULINK, SIGNAL, Formal Methods, Code Generation

1. Introduction

Cyber Physical Systems (CPS) are engineering systems consisting of the integration of computational control and physical components with continuous dynamics. As these systems are becoming more complex and their reliability and safety requirements are becoming more and more crucial, and harder to guarantee by the traditional design tools and methodologies, new design paradigms are emerging. Model Based Design is a much discussed approach for developing such systems. It consists of building mathematical models that capture the specifications as well as the critical design decisions for the system in the different stages of the development life cycle. Different tools have been developed to generate correct by construction code from these models, as well as for the verification of the system behavior in early design phases. Despite the intensive research in the Model Based Design, the Mathwork's graphical environment SIMULINK[5] is still the most widely used tool for the design of embedded software. Although it is very convenient to use, SIMULINK does not have published and authentic formal semantics. Hence, its models can not be used with the Model Based Design framework. Its generated diagrams are verified through numerical simulations and its behavior is strongly correlated with the simulation configuration parameters. Although simulation based analysis is a well accepted technique in industrial practice, it becomes impossible to exhaustively simulate the system for verification purposes, once it gets very complex. The preservation of semantic is another issue, since the behavior equivalence between the simulated model and the generated code is unclear. Formal models, on the other hand, are less applied as they are less intuitive to use and harder to learn. In order

to close the gap between formal methods and industrial practices, researcher have attempted to either give formal semantics[8] to SIMULINK or translate it into formal models of computation[15][6][16]. In this paper, we present a prototype tool, SIM2SIG, that translates the discrete time blocks of SIMULINK to SIGNAL. SIGNAL is a data flow synchronous programming language, which was developed by IRISA[1]. Each variable (signal) within the SIGNAL program has its own clock, giving us the multi-rate (polychronous) formalism of SIGNAL. This timing model allows for streams to be computed asynchronously, which fits very easily to a multi-thread environment. This increases the embedded software reactivity and capabilities. Moreover, a number of formal verification tools such as the model checker SIGNALI[3] and the graphical developing interface SME[2] exist for Signal. These characteristics make SIGNAL an interesting model of computation for embedded software design. We follow the same translation methodology proposed in [16], for translating LUSTRE to SIMULINK, namely type and clock inference, and hierarchical block by block translation. The novelty in this work consists of bridging the gap between the 'almost' synchronous model of computation of SIMULINK and the polychronous model of computation of SIGNAL. In the past work by [16], the translation was straight forward due to the fact that the target language is synchronous and a global clock driven, whereas in SIGNAL language there is no global clock per se. A global clock may be calculated using the clock calculus if the translated SIMULINK model has the endochrony property. If a single global clock driver does not emerge, a polychronous model leading to multi-threaded behavior emerges. The other addition in this work is the use of affine clock relations between SIGNAL sub-processes, when multiple SIMULINK blocks have sampled inputs with varying sampling rates. The rest of the paper is organized as follows: Section 2 is a survey of the translation of SIMULINK to different models of computation. Section 3 is an overview of the SIGNAL formalism. In Sections 4 and 5, we compare SIMULINK and SIGNAL formalisms and present the translation goals and assumptions. Sections 6, 7 and 8 represent the three steps of the translation. Our prototype tool Sim2Sig is described in Section 9. In Section 10, Sim2Sig is tested on a SIMULINK model of a discretized DC-motor closed loop controller. We close this paper with some concluding remarks and suggested future work

2. Related work

A hand full of research efforts in the past have tried to give formal semantics to SIMULINK either by converting its models into a synchronous language [16] [9], hybrid automata[12], or I/O extended Finite Automata[6] or into a system of mathematical equations[8]. The main motivation for translating SIMULINK models to a formal language program lies in gaining access to the analysis and verification tools of the target language. In [16], discrete time SIMULINK was translated to LUSTRE[11] following three steps: clock inference, type inference and hierarchical bot-tom up block by block translation. Basic blocks like Addition or Multiplication are translated to primitive LUSTRE operators. Complex nodes like Subsystems are translated to Lustre nodes, which are carefully named in order to keep track of the original SIMULINK hierarchy. In [6], a semantic translator from SIMULINK to Hybrid System Interchange Format (HSIF) was introduced. HSIF is a network of hybrid automata, which can interact with each other using signals and shared variables. The translation was limited to continuous SIMULINK blocks and STATEFLOW Diagrams. The translation from SIMULINK/STATEFLOW is based on graph transformation. HA can model both continuous and discrete systems. However, they have not been formally standardized. Chapoutot et. al. [8] proposed to assign formal semantics to SIMULINK's simulation engine, solver and a subset of blocks that span discrete and continuous operations. The dynamical SIMULINK system is represented as a state space with continuous time, as well as discrete time state functions to represent the fact that SIMULINK models are hybrid systems. The simulation goal consists of finding the solution for the set of state space equations. This approach was validated by comparing the outputs of the SIMULINK simulator and the equation-based one for different case studies. Although the system of equations approach is able to cover both discrete and continuous blocks, this approach lacks tools for the equation grammar verification and simulation. In order to take advantage of the high computing power resulting from multi-core architectures, multi threading is very desired. Due to its multi-rate formalism, the polychronous language SIGNAL, leads naturally to multi-threaded code synthesis. This justifies our motivation for choosing SIGNAL as a target language.

3. The polychronous language signal

SIGNAL is a declarative multi-rate synchronous language. It satisfies the synchrony hypothesis, which assumes that the computation and communication time are instantaneous. While the synchronous languages have a totally ordered model of logical time, SIGNAL's model of logical time is partially ordered. The semantics of the language does not assume an a priori existence of a reference clock. Each variable (signal) is characterized by its own clock. In the following section, we introduce some preliminaries notions related to SIGNAL.

3.1. Preliminaries

The basic entity in a polychronous language is an event.

Definition 3.1. (Event). An event is an occurrence of a new value. We denote the set of all events in a system by Ξ. The relative occurrences of events can then be represented using the following binary relations over Ξ:

Definition 3.2. (Precedence, Preorder, Equivalence). Let < be a precedence relation between events in Ξ. It is defined such that ∀a, b ∈ Ξ, a < b if and only if a occurs before b. The relation < defines a partial order on Ξ such that ∀a, b ∈ Ξ, a < b if and only if a occurs before b or a, b occur logically simultaneously, or their order does not matter. Finally the equivalence relation ∼, is defined on Ξ such that a ∼ b = a < b ∧ b < a, meaning that a and b are equivalent only if they occur simultaneously or their order does not matter. Thus ∼ represents synchronicity of events.
An instant can also be seen as a maximal set of events that occur in reaction to any one or more events. Formally:

Definition 3.3. (Logical Instant). The set of all instants is denoted by Y. Each instant in Y can be seen as an equivalence partition obtained by taking the quotient of Ξ with respect to ∼ such that Y = Ξ / ∼. For each set S ∈ Y, all events in S will have the property ∀a, b ∈ S, a ∼ b, and ∀a, b, (a ∈ S1 ^ b ∈ S2 ^ S1 ≠ S2 ^ S1, S2 ∈ Y → a !∼ b. Each instant contains events on signals. If a signal has no event in an instant then it is considered absent. We denote a specific value of a signal x by function x(t) where t ∈ N and t represents the tth instant in the totally ordered set of instants where signal x is different from ⊥.

Definition 3.4. (Epoch, Clock). The epoch, σ(x), of a signal x is the maximum set of instants in Y where for each instant in σ(x), x takes a value from T. The clock of the signal x is a characteristic function that tells whether or not an event in x is absent or is in the set T . Clock is a function of type Y → [true, false] such that for a signal x it returns another signal ˆx defined by ˆx (t)= true if x(t) ∈ T .
Note that not all inputs and outputs are present or computed during every instant in Y which means that not all signals have the same epoch or clock. This gives the multi-clocked or polychronous behavior. Using the above definitions and characteristics, three possible relationships can be drawn between any two clocks x and y: equivalent, sub-clocked, or unrelated. If the clocks of x and y are true for the exactly the same set of instants, ˆx = yˆ, then it is said that these two clocks are equivalent, and the corresponding signals are also synchronous. If the clock of a signal x is true for a subset of instants where the clock of y is true then it is said that x is a sub-clock of y. If the clocks of x and y are not equivalent or subset or superset of the other then the clocks are said to be unrelated [14].

3.2. The SIGNAL Formalism

The primitive SIGNAL operators are Function, Delay, Under-sampling and Priority Merging (see Figure 1).

1. Function Operator: performs user defined operations on a set of signals x1, ... xn that must be present simultaneously and produces an output y at the same instant.

$$\begin{array}{c}\text{Operation: } y := f(x1,x2,\ldots xn)\\ \text{Clock Relation: } \hat{}y = \hat{}x1 = \hat{}x2 = \ldots \hat{}xn\end{array} \quad (1)$$

2. Delay Operator: sends a previous value of the input to the output with an initial value k as the first output. The original and delayed signals are synchronous.

$$\begin{array}{c}\text{Operation: } y := x \$ \text{ init } k\\ \text{Clock Relation: } \hat{}y = \hat{}x\end{array} \quad (2)$$

3. Under-Sampling Operator: down-samples an input signal x based on the true occurrence of another input signal z. The output signal clock is thus equal to the intersection of the clocks of x and z=true, noted [z].

$$\begin{array}{c}\text{Operation: } y := x \text{ when } z\\ \text{Clock Relation: } \hat{}y = \hat{}x * [z]\end{array} \quad (3)$$

4. Priority Merging Operator: merges two signals x and z into one signal y. At any logical instant, if x is present, then y will have the value present on x, else y will have the value present on z. If neither x nor z are present, y is absent.

$$\begin{array}{c}\text{Operation: } y := x \text{ default } z\\ \text{Clock Relation: } \hat{}y = \hat{}x + \hat{}z\end{array} \quad (4)$$

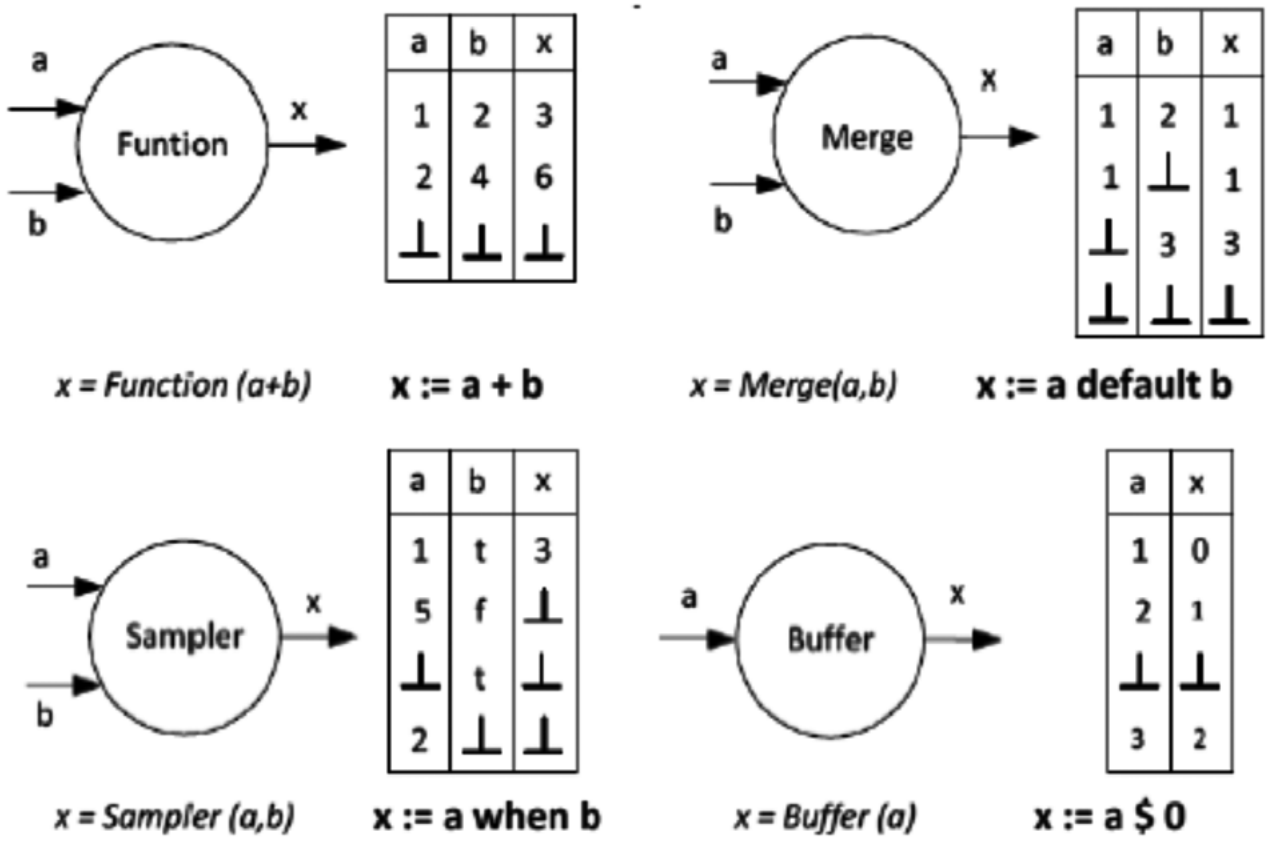

Figure 1. Primitive SIGNAL operations [7].

3.3. Advanced SIGNAL Constructs

Clock relations are not only inferred from the SIGNAL statement, they can be given explicitly [10]: The equation, *clk:=when b*, implies that *clk* represents the set of instants at which *b* holds true. The equation, *clk ^ = s*, implies that *clk* is the clock of *s*. The equation, *s1 ^ = s2*, specifies that the signals *s1* and *s2* are synchronous. Another useful construct is *Cell: y := x cell z init k*. In this case, the output signal contains the values of the first input signal *x* for all its instants and retains the previous value of *x* during the true instances of the second boolean input *k*. The clock of *y* is the union of the clocks of *x* and *z*. An example of the Cell operator is shown below:

x: ⊥ v2 v3 ⊥
z: t t ⊥ f
y: k v2 v3 v3

3.4. SIGNAL Processes

A SIGNAL program is a process. The parallel composition of two processes P and Q, noted P|Q is the union of equation systems defined by both processes. P and Q communicate via their common signals. The template of a SIGNAL process is:

```
Process MODEL ={ %parameters%}
      (? %inputs%;! %outputs% ;)
  (| %body of the process% |)
where
  %local declarations%
end ;
```

The input-output ports are declared using the symbol ? and ! respectively. Each input or output is associated with its type (event, integer, boolean, real). Each SIGNAL statement consists of the four primitive operators.

4. Comparing simulink and signal

Both SIMULINK and SIGNAL are data-flow languages. They both manipulate signals. In SIMULINK, signals are the wires that connect the blocks in a model. In SIGNAL, a signal is the program variable corresponding to a stream. A system performs a specified operation on an input signal and produces an output signal. The systems in SIMULINK are library blocks that could be simple (e.g., Adder, Product) or composed (subsystems). In SIGNAL, systems are built-in operators (e.g., when, default), as well as user defined ones, called processes. Another similarity consists of the hierarchical composition of systems. In SIMULINK, the subsystems are drawn graphically within their parent system, to form a tree structure. In SIGNAL, as well, a parent process can contain multiple subprocesses. Despite of these similarities, SIMULINK and SIGNAL are different in several major ways: First, SIGNAL has a well defined formal semantic, whereas SIMULINK's behavior strongly depends on the choice of the simulation parameters. For example, some models are accepted if we allow to handle rate change automatically, others are rejected if the automatic rate change opting is unchecked. Second, SIGNAL has a discrete time semantics, whereas SIMULINK has a continuous one. Even the blocks belonging to the discrete library produce piecewise constant continuous-time signals. Third, SIGNAL is a strongly typed language that explicitly specifies the type of each flow. However, SIMULINK does not require the type specification for each block. This can be done, using, for instance, a Data Type Converter Block (see Section 6). Finally, SIGNAL is a multi-rate language, which means that two variables can be of different rates and can remain unrelated throughout the program. However, SIMULINK, both in sample-driven and event-driven cases, has a global clock, namely the simulation clock, that is synchronous with every clock in the model (see Section 7).

5. Translation goals and assumptions

The problem of semantic translation can be formulated as follows: Given a SIMULINK model of a dynamic system, compute a flow equivalent dynamic system model in SIGNAL which produces the same execution traces as the simulation output in SIMULINK. Our tool rejects the models with typing or timing errors flagged by SIMULINK. We limit our translation to the discrete time part of SIMULINK. This is justified by the fact that only the controller in safety critical systems is implemented on the computer, hence it must be designed in discrete time. The list of supported SIMULINK blocks is shown in Figure 2. As SIMULINK semantics depend on the simulation method, we limit our translation to one method. We chose the solver to be discrete and fixed step, the simulation mode to be auto and to automatically handle rate transition for data transfer deterministically. We also assume that the boolean logic signals flag is on. We developed our translation method using MATLAB 7.12.01 (R2011a) and SIMULINK block Library V7.7.

6. Type inference

6.1. Types in SIMULINK

Unlike in SIGNAL, variable types are not explicitly declared in SIMULINK. However, implicitly, SIMULINK has some typing rules. The simulation engine rejects some models because of typing errors. The basic types for SIMULINK are: boolean, double, single, int8, uint8, int16, uint16, int32, uint32. The main SIMULINK typing rules are:

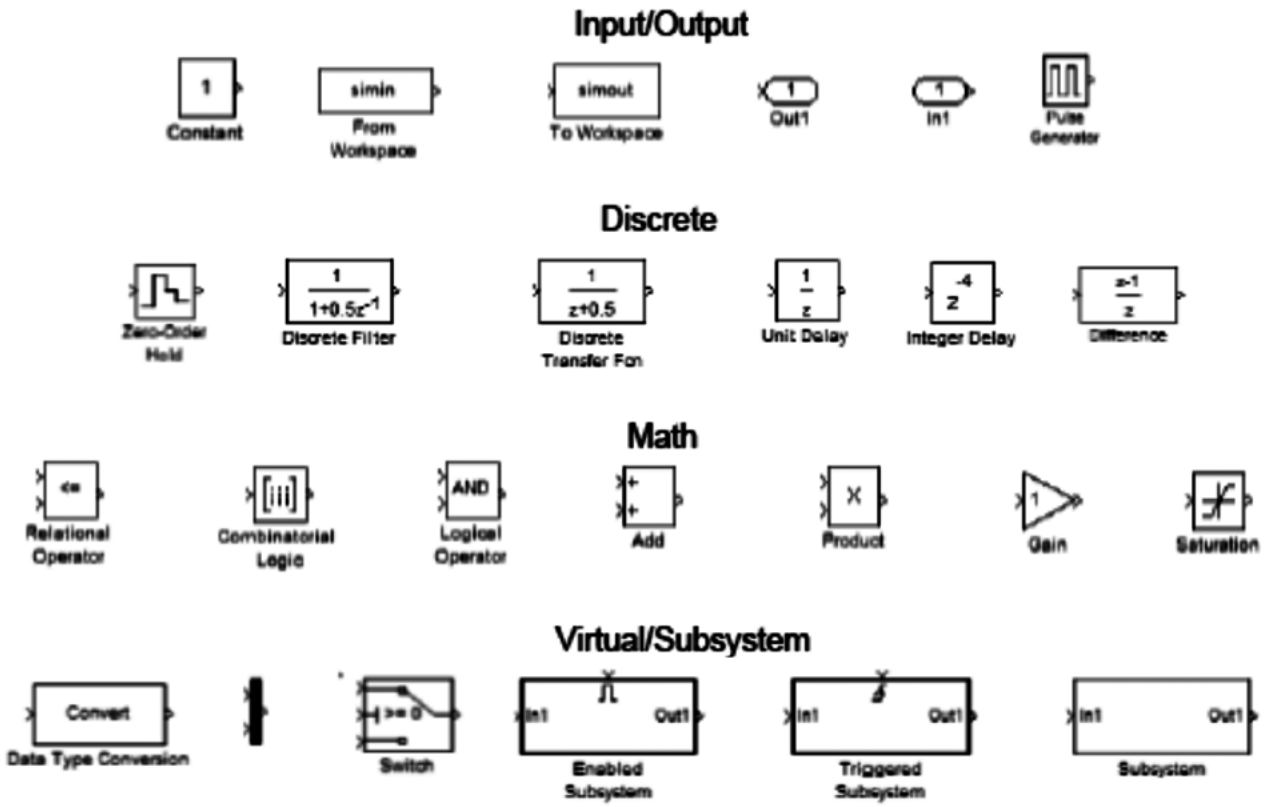

Figure 2. Supported SIMULINK Blocks.

- By default, all signals are of type double, except when a block requires a defined type. For example, the inputs of Logical Operator blocks must be of type boolean.
- The user can explicitly set the type of a signal to another type (e.g., by a Data Type Converter Block)
- An error type occurs when incompatible types are fed in one block, for example, when a boolean and an integer are fed to the same Adder block. The typing rules for each block are given in Table 1. We define T_{Num} = {double, single, int8, uint8, int16, uint16, int32, uint32}, and T_{Bool} ={boolean}. Let $\{\alpha, \phi\} \in T_{Num}$, $\theta \in T_{Bool}$ and $\{\gamma, \beta\} \in \{T_{Bool}, T_{Num}\}$

SIMULINK Block	Typing Rule
Constant	α
Adder	$\alpha \bullet \ldots \bullet \alpha \rightarrow \alpha$
Gain	$\alpha \rightarrow \alpha$
Relational Operator	$\alpha \bullet \alpha \rightarrow \theta$
Logical Operator	$\theta \bullet \ldots \bullet \theta \rightarrow \theta$
Discrete Transfer Function	$\alpha \rightarrow \alpha$
Unit Delay, Inport, Outport	$\gamma \rightarrow \gamma$
Data Type Converter	$\gamma \rightarrow \beta$
Switch	$\alpha \bullet \phi \bullet \alpha \rightarrow \alpha$

Table 1. Typing Rules for Some SIMULINK blocks.

6.2. Types In Signal

SIGNAL is a strongly typed language: variables have a declared type and operations have precise type signatures. The basic types for SIGNAL are integer, real and boolean. Type casting can be performed as in C. For example, an integer x is converted to a real y as follows: y = integer(x). The array type allows grouping synchronous elements of the same type. An array of size N with elements of type *element_type* is defined as follows: $[inp_1, ..., inp_N]$ *element_type*.

6.3. Type inference

The goal of this step consists of inferring the type of each signal in SIMULINK, so that its corresponding type in SIGNAL can be used in the translation. For the type inference, we use a fix-point algorithm on the lattice shown in Figure 3. ⊥ means undefined type and error means typing error. We call $x^T \in T_{Sim}$ the type variable corresponding to the variable x, with $T_{Sim} = \{T_{Num}, T_{Bool}\}$. We define a monotonic function sup: $(T_{Sim})n \rightarrow T_{Sim}$ in the type lattice, where n is the number of blocks in the SIMULINK model. Sup(x^T ,y^T)=z^T , denotes that z^T is a least common upper bound of x^T and y^T . The fixed point is calculated on the set of equqtions shown in Table 2.

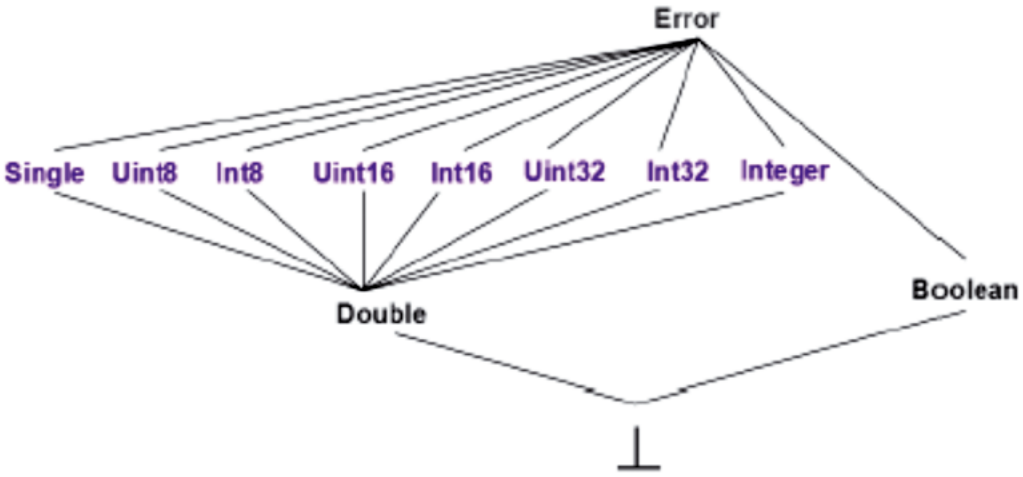

Figure 3. The Type Lattice.

SIMULINK Equation	Type Equation
y = Adder(x_1, . . . , x_k)	$y^T = x^T_1 = \cdots = x^T_k$ = Sup (double, y^T , x^T_1 , . . . , x^T_k)
y = Constant$_\alpha$	y^T =if y^T <= α then α else error
y = DataT ypeConv$_\alpha$(x)	y^T =if y^T <= α then α else error
y = UnitDelay(x)	$x^T = y^T$
y = RelOp(x_1, x_2)	$x^T_1 = x^T_2$=Sup (double, x^T_1, x^T_2), y^T =bool
y = LogOp(x_1, . . . , x_k)	$x^T_1 = x^T_2 = x^T_1 = \ldots = x^T_k = y^T$ =bool
y = Switch(x_1, x_2, x_3)	$x^T_1 = x^T_1 = y^T$ =Sup (x^T_1, x^T_3, y^T)

Table 2. Type Inference Equations [16].

7. CLOCK INFERENCE

7.1. Time in SIMULINK

SIMULINK has two different timing mechanisms, namely samples and triggers.

7.1.1. Sample Time

The discrete time SIMULINK signals are piecewise-constant continuous-time signals. Blocks in SIMULINK can be assigned sample times, as configuration parameters. A sample time equal to 2, means that the block should be evaluated every two ticks of the global simulation clock. The sample time corresponds then to the period π of the block output signals. Some blocks (e.g, Pulse Generator) can also be characterized by initial phase θ, which is propagated to the neighboring blocks. Hence, in general, every block is characterized by a period π and a phase θ. It is evaluated every k π + θ (k=0, …, n). By default, blocks have their sample time set to -1, which corresponds to an inherited (from the inputs or the parent subsystem) value. We assume that the configuration option, *Automatically handle rate transition* for deterministic data transfer, is chosen. SIMULINK has some timing rules, if violated, the model is rejected:

- The inputs of a simple block B must have sample times that are multiplier or divisor of the block sample time: ($\pi_{Inp1...n} = k\ \pi_B$) OR $\pi_{Inp1...n} = 1/k * \pi_B$), with k = 0, …, n
- Enabled and Triggered Subsystems' inputs should have the same sample times: $\pi_{Inp1} = \ldots = \pi_{Inpn}$.

7.1.2. Triggered Subsystem

The second timing mechanism of SIMULINK is the *triggers*. Only subsystems can be triggered by a signal Trig. The triggered subsystem is evaluated if *Trig* has a *rising* or *falling* transition. The sample time of the blocks inside a triggered subsystem are all equal to the period T of the trigger signal. The example below shows the execution of a triggered subsystem (rising trigger). We assume that the triggered subsystem has no sub-blocks inside.

Trigger:	0	1	1	1	0	1	0	1	1
Input:	0	1	2	3	4	5	6	7	8
Output:		1				5		7	

7.1.3. Enabled Subsystem

The timing mechanism of the enabled subsystem is ambiguous [16]. It cannot be understood from a set of experiments. For the sake of the translation, we assume that the enabled subsystems have the same timing mechanisms as the triggered ones. The only difference lies in evaluating the block, if the *Enable* signal is equal to 1. The example below shows the execution flow of an enabled subsystem. We assume that the subsystem contains no sub-blocks.

Enable:	0	1	1	1	0	1	0	1	1
Input:	0	1	2	3	4	5	6	7	8
Output:		1	2	3		5		7	8

7.2. Time in Signal

SIGNAL has a partially ordered logical time. This means that the duration is abstracted to a point, namely the logical instant, and the time instants are partially ordered. Simi-lar to the synchronous languages, SIGNAL also assumes the synchrony hypothesis. However, SIGNAL does not have a global clock, as a reference for sampling all the signals at each tick. Each SIGNAL flow x is characterized by a boolean flow bx, called the clock of x. If x is present at instant i, bx(i) is equal to true, otherwise it is equal to false. The signal clocks can be independent until the end of the program. In case of synchronization requirements, extra timing constraints can be added. Epoch analysis is performed, in order to determine whether a sequential program can be synthesized from the SIGNAL specifications. In other words, it determines whether a *Master Trigger* can be found. If not, exogenous constrains are required from the user to form a *Master Trigger* [13]. We refer the reader to [4] for more detailed discussion on SIGNAL timing model.

7.3. Clock Inference

The blocks inside a triggered or enabled subsystem must have a sampling period and phase equal to the ones of the enclosing Triggered/Enabled subsystem. Otherwise, we consider two cases. In a first case, the sample time of a given block bi is defined (Periodes[i] != -1). If it is a multiplier or divisor of the input signals'periods, it is kept. If, however, the sample time of the block is undefined (Periodes[i] == -1), it is inferred as the greatest common divisor of the input signals' periods [16] (See Formulas 5, 6 and 7).

$$(\pi_B, \theta_B) = GCD_{rule}\ ((\pi_i, \theta_i)_{\ i=1...n}) \tag{5}$$

$$\pi_B = \begin{cases} \gcd(\pi_1, \ldots, \pi_n) & \text{if } \theta_1 = \ldots = \theta_n \\ \gcd(\pi_1, \ldots, \pi_n, \theta_1, \ldots, \theta_n) & \text{otherwise} \end{cases} \tag{6}$$

$$\theta_B = \begin{cases} \theta_1 \bmod \pi & \text{if } \theta_1 = \ldots = \theta_n \\ 0 & \text{otherwise} \end{cases} \tag{7}$$

For Example:
GCDrule ((12,4), (12,0)) = (4,0), GCDrule ((12,4), (4,4)) = (4,4),
GCDrule ((12,4), (12,3)) = (1,0), GCDrule ((4,0), (3,0)) = (1,0).

8. Translation

8.1. Type Translation

Once the type Inference step is completed, the obtained SIMULINK types are mapped to their corresponding SIGNAL ones, as it is shown in Table 3.

SIMULINK Type	SIGNAL Type
bool	Boolean
int8, uint8, int16, uint16, int32 or uint32	Integer
⊥, double, single	Real

Table 3. Type Translation.

8.2. Clock Translation

Once all the sample times are inferred, we use these information to reproduce the same traces of the SIMULINKmodel simulation. The blocks inside a triggered or enabled subsystems are assigned the periods and phases of the trigger/enable signal. For the rest of the blocks, according to the ratio between the periods of the every two directly connected blocks A and B, we distinguish three cases:

- Case 1 : $\alpha = Periode_A / Periode_B = 1$
  The output signal clock of A is equal to the output signal clock of B:
  $output_A \hat{} = input_B \hat{} = output_B$
- Case 2 : $\alpha = Periode_A / Periode_B > 1$
  Undersampling should be performed in B. The clock relation between the input and output signals of B is: $\hat{}$ outputB = α $\hat{}$ inputB + β. β here is the phase difference between $output_B$ and $input_B$: $\beta = Phase_B - Phase_A$. In order to implement the discussed affine clock relation, we consider a counter variable cnt, that has the same clock as the input signal outputA. Hence, staring from the initial phase, cnt is incremented every time a new input is read. When cnt reaches a multiplier of α, the function f performed by the block is evaluated and the output is produced. The following SIGNAL code illustrates the above explained algorithm for a Unit Delay block:
  | *cnt := (cnt +1) $ init* $(PHASE_B - PHASE_A)$*;*
  | *cnt2 := cnt modulo* $(PERIODE_A / PERIODE_B)$*;*
  | *cnt* $\hat{}$ = $input_B$ *;*
  | *tmp :=* $input_B$ *$ init 1;*
  | $output_B$ *:= tmp when (cnt2=0);*
  The flow of the output signal outputB for the Unit Delay block, with α=2 and β=0 and initial value v_0, is:

$Output_A$	v_1	v_2	v_3	v_4	v_5	v_6
Cnt	0	1	2	3	4	5
Cnt2	0	1	0	1	0	1
tmp	v_0	v_1	v_2	v_3	v_4	v_5
$Output_B$	v_0	.	v_2	.	v_4	.

- Case 3 : $1/\alpha = Periode_A / Periode_B < 1$
  Oversampling should be performed in B. Similar to the Undersampling case, the clock relation between the input and output signals of B is: $\hat{}$ $output_B$ = α $\hat{}$ $input_B$ + β. β here is the phase difference between $output_B$ and $input_B$:
  $\beta = Phase_B - Phase_A$. In order to implement the discussed affine clock relation, we consider a counter variable cnt, that has the same clock as the output signal outputB . Hence, staring from the initial phase, cnt is incremented. When cnt reaches a multiplier of α, a new input is read, the function f performed by the block is evaluated and a new output is produced. Otherwise, the old output is emitted. The following SIGNAL code illustrates the oversampling algorithm in case of a Unit Delay block:

| cnt := (cnt+1) $ init (PHASE$_B$ - PHASE$_A$);
| cnt2 := cnt modulo (PERIODE$_B$/PERIODE$_A$);
| cnt ^= output$_B$;
| input$_B$ ^= when (cnt2=0);
| tmp := input $ 1 init 1;
| output$_B$:= tmp cell ^ outputB;

The flow of the output signal Output$_B$ for the Unit Delay block, with α =2 and β =0 is shown below:

Output$_A$	v_1	.	v_2	.	v_3	.	v_4	.	v_5
Cnt	0	1	2	3	4	5	6	7	8
Cnt2	0	1	0	1	0	1	0	1	0
tmp	v_0	.	v_1	.	v_2	.	v_3	.	v_4
Output$_B$	v_0	v_0	v_1	v_1	v_2	v_2	v_3	v_3	v_4

8.3 Basic SIMULINK Blocks translation

In this section, we illustrate how the main basic blocks are translated. The remaining blocks are translated similarly.

- Sum Block: performs addition or substraction on its inputs. The operation of the block is specified by the list of signs parameters ((+) and (-)), indicating the operations to be performed on the inputs:
  | out := inp$_1$ + inp$_2$ -inp$_3$
- Gain Block: performs a multiplication of the input with a constant:
  *| out := inp * GAIN*
- Logical Operator Block: performs a boolean operation Op ∈{AND,OR,NOT} on its inputs:
  | out := inp$_1$ Op Inp$_2$
- Unit Delay and Integer Delay Block: The output of the Unit Delay and Integer Delay blocks is a delayed version of the input by NB DELAY instants. NB DELAY is equal to 1, in case of a Unit Delay Block. INIT VALUE is the initial value of the output:
  | out := inp$ NB DELAY init INIT VALUE
- Data Type Conversion Block: is translated into a Type Casting operation. The following code translates a real input into an integer output:
  | out := (integer) inp
- Zero-Order Hold Block: If the sample time of the Zero-Order Hold Block is set to -1, it is equivalent to the identity function:
  | out := inp. Otherwise, the clock translation, as explained in Section 8.2 is performed.
- Constant Block: The Constant Block value is added in SIGNAL to the list of the parent process parameters.
- Saturation Block: truncates its inputs according to an upper limit (LIM$_{UP}$) and a lower limit (LIM$_{LOW}$) bounds given by the user:
  | out := (LIM$_{UP}$ when (inp > LIM$_{UP}$)) default (LIM$_{LOW}$ (inp < LIM$_{LOW}$))) default input
- Switch Block: The Switch Block has three inputs. It compares its middle input inp$_2$ to a threshold value. If it is greater than the THRES, the first input is passed to the output, otherwise the third input is emitted as an output:
  | out := (inp1 when (inp$_2$ > THRES)) default (inp$_3$ when (inp$_2$ < THRES))
- Pulse Generator: The Pulse Generator with a period= 4, a phase= 2, an amplitude= AMP and a pulse width=2 is translated into the following SIGNAL code:

```
process PulseGenerator =  { real AMP}
(? ! real out;)
(| dpg1:= dpg2 $1 init AMP
| dpg2 := dpg3 $1 init AMP
          | dpg3 := dpg4 $1 init real(0)
| dpg4 := dpg5 $1 init real(0)
| pha1 := dpg1 $1 init real(0)
| pha2 := pha1 $1 init real(0)
| out := pha2 |)
where
     real dpg1, dpg2, dpg3, dpg4, pha1, pha2;
end;
```

- Discrete Filter and Discrete Transfer Function: The Discrete Filter/Discrete Transfer Function's parameters are the nominator coefficients number values COEFFN , the denominator coefficients values COEFFD and the initial state value INIT VAL. The transfer function $1/(1+0.5z^{-1})$ is translated into the following SIGNAL code:
  | *output := (input * $COEFF_N$[0] + tmp0)/$COEFF_D$[0]*
  | *tmp0 :=(- $COEFF_D$[1] * output)$1 init INIT_VAL*
- Mux: The Mux block combines its inputs into a single vector output. It is generally used to merge the output of different blocks. The SIGNAL code for a Multiplexer with three inputs inp1, inp2 and inp3 is:
  |*out := [inp1, inp2, inp3]*
- Combinatorial Logic: implements a truth table. It reads a boolean number, and outputs the row in the boolean table corresponding to the read input.

```
Process CombinatorialLogic =
    { integer N, M, K; [N] boolean TruthTab; }
    (? [M] boolean inp; ! [K] boolean out );
    (| array i to (M-1) of
            (| Row:= Row[?]+ ((1 when inp[i])
              default (0 when not inp[i])) |)
            with
            (| Row:=0 |)
            end
    | index0:= Row*K-K..Row*K
    | output := TruthTab[index0] |)
```

- FromWorkspace/ToWorkspace: are translated respectively into an input and an output of the SIGNAL process.
- Trigger: takes a real/integer flow and transforms it into a boolean flow. We distinguish between *Rising Trigger, Falling Trigger* or *Either*. The *Rising Trigger* gets the value true when an input transition from a negative number to a positive one happens. The *Falling Trigger* is true when an input transition from a positive to a negative value occurs. The *Either trigger* is true, if either a rising or a falling transition happens. The following example, illustrates the trigger mechanism:

Input	-1	0	1	2	-2	3	1	4	-1
Rising Trigger	f	t	f	f	f	t	f	f	f
Falling Trigger	f	f	f	f	t	f	f	f	t
Either Trigger	f	t	f	f	t	t	f	f	t

  The following SIGNAL code generates a Rising Trigger flow. The not_before variable ensures that the trigger is only produced, if no one happened in the previous time step. The
  Falling Trigger is defined similarly:
  | *RiseTriggerold := Trig $ init false*
  | *Trig := neg_to_nonneg OR (nonpos_to_pos and not_before)*
  | *neg_to_nonneg:= (inp_{old} < 0) AND (inp >= 0))*
  | *nonpos to pos := (inp_{old} <= 0) AND (inp > 0))*
  | *not_before := NOT ($RiseTrigger_{old}$)*
- Enable: Similar to the Trigger, the Enable block transforms a real/integer input flow to a boolean flow en. en has the value true, when the input inp is positive:
  | *en:=(true when (inp > 0)) default (false when (inp <= 0))).*

8.4. Subsystems translation

As the model increases in size, its complexity can be reduced by grouping the functionality related blocks together into subsystems. A subsystem can be executed conditionally or unconditionally. A conditionally executed subsystem may or may not execute depending on a control signal. We distinguish between triggered and enabled subsystems.

8.4.1. Plain Subsystems Translation

A SIMULINK diagram can be constructed in SIGNAL by recursively translating subsystems into processes and the enclosed atomic blocks into subprocesses. The first step in the top down translation is the blocks parameter list generation, then the outputs and inputs definition. After that, the subprocesses in the first hierarchy level are called.

Additional equations for the block connections are generated. In the Where part, the local variables are defined. The subprocesses body is also implemented. For every block, type and clock translation are performed as described is Section 8.1 and Section 8.2.

```
Process P= {integer N;}
        (? integer inp; boolean b; ! integer out;)
        (| tmp := Q{N}(inp)
        | out := tmp when b|)
Where
        Integer tmp ;
        Process Q={ integer M; }
         (? integer s1; ! integer s2;)
        (| s2:= s1 * M |);
```

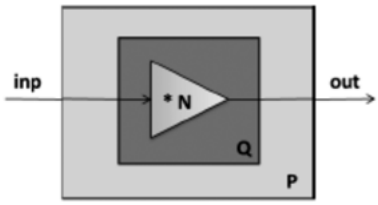

Figure 4. Plain Subsystem Translation.

8.4.2 Triggered Subsystems Translation

The Triggered Subsystem is a subsystem with a control input, namely the *trigger* input. The subsystem is executed, each time a trigger event occurs. If no trigger happens, the output is either reset or it holds its old value. Figure 5 shows a Triggered Subsystem enclosing a Unit Delay block.

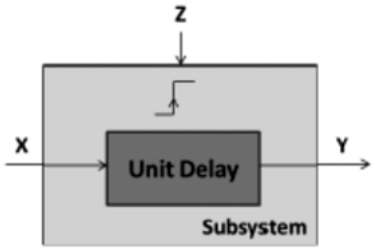

Figure 5. Triggered Subsystems Translation.

Below is the corresponding SIGNAL code. The parent process *Sim2Sig* calls the subprocesses *Trigger* and *SubSys*. Trigger generates the rising trigger $SubSys_{trig}$ from the in-put Y. *SubSys* is the Triggered Subsystem. It is only executed when $SubSys_{trig}$ is true, otherwise it emits its old output, each time a new input with no trigger event arrives.

```
Process Sim2Sig = {integer NB_D, INIT_V; }
        (? integer X, Y; ! integer Z;)
        (| SubSys_trig := Trigger(Y)
        | Z := SubSys { NB_D, INIT_V } (X when SubSyst_rig)cell ^X | )
 Where
        boolean SubSystrig;
        Process Trigger = (As defined in Section 8.3)
        Process SubSys= { integer NB_D, INIT_V; }
                (? integer in; ! integer out;)
                (| UnitDelay_out := UnitDelay {NB_D, INIT_V} (UnitDelay_in)
                | UnitDelay_in := in
                | out := UnitDelay_out |)
        Where
                integer UnitDelay_in, UnitDelay_out;
                Process UnitDelay= (As defined in Section 8.3)
        End ;
```

8.4.3. Enabled Subsystems Translation

The same translation method discussed for the case of a triggered subsystem, applies for the enabled one. The only difference is replacing the Trigger block with an Enable one.

9. Implementation: the tool SIM2SIG

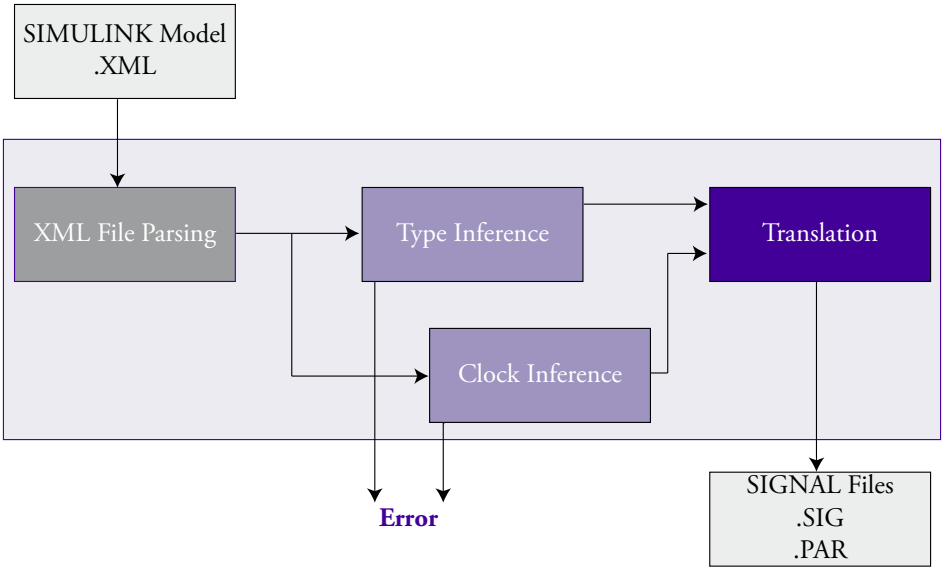

Figure 6. Translation Framework.

Our prototype tool SIM2SIG is written in C++. It architecture is shown in Figure 6. It reads an XML file generated by SIMULINK, parses it and builds a data structure representing the original SIMULINK model. After that, type inference (Section 6), clock inference (Section 7) and the translation (Section 8) steps are performed. The tool outputs a SIG file containing the SIGNAL program and a PAR file with the parameters, as well as error messages.

10. Case study: discretized dc-motor closed loop controller

In this section, we use our tool to translate into SIGNAL a SIMULINK model consisting of a system consisting of an input sampler and a discretized DC-motor in a loop with a PID controller. The electrical and mechanical dynamics of the three-level DC-motor are represented by the following equations:

$$V_{in} - R \cdot i - K_e \cdot \theta[n+1] = L \cdot i\,[n+1] \quad (8)$$

$$K_t \cdot i - b \cdot \theta\,[n+1] = J \cdot \theta[n+2] \quad (9)$$

R stands for the resistance, b for the damping factor L^{-1} is the inductance, J^{-1} is the inertia, i is the current and θ is the angular frequency. Equation 8 is implemented in the subsystem S1. Equation 9 is implemented in the subsystem S2.

The system inputs are sampled by the *Input sampling* block. The *Adder* Block in the PID subsystem down-samples the voltage with a factor of two. The *Rate Adjustment* block has a sampling rate equal to 1. Hence, the DC-Motor output is over-sampled with a factor of two. This case study shows then how the hierarchical translation is performed, as well as how different SIMULINK Timing mechanisms are translated into SIGNAL. The flow equivalence is validated by comparing the traces generated from the SIMULINK model and its corresponding SIGNAL translation.

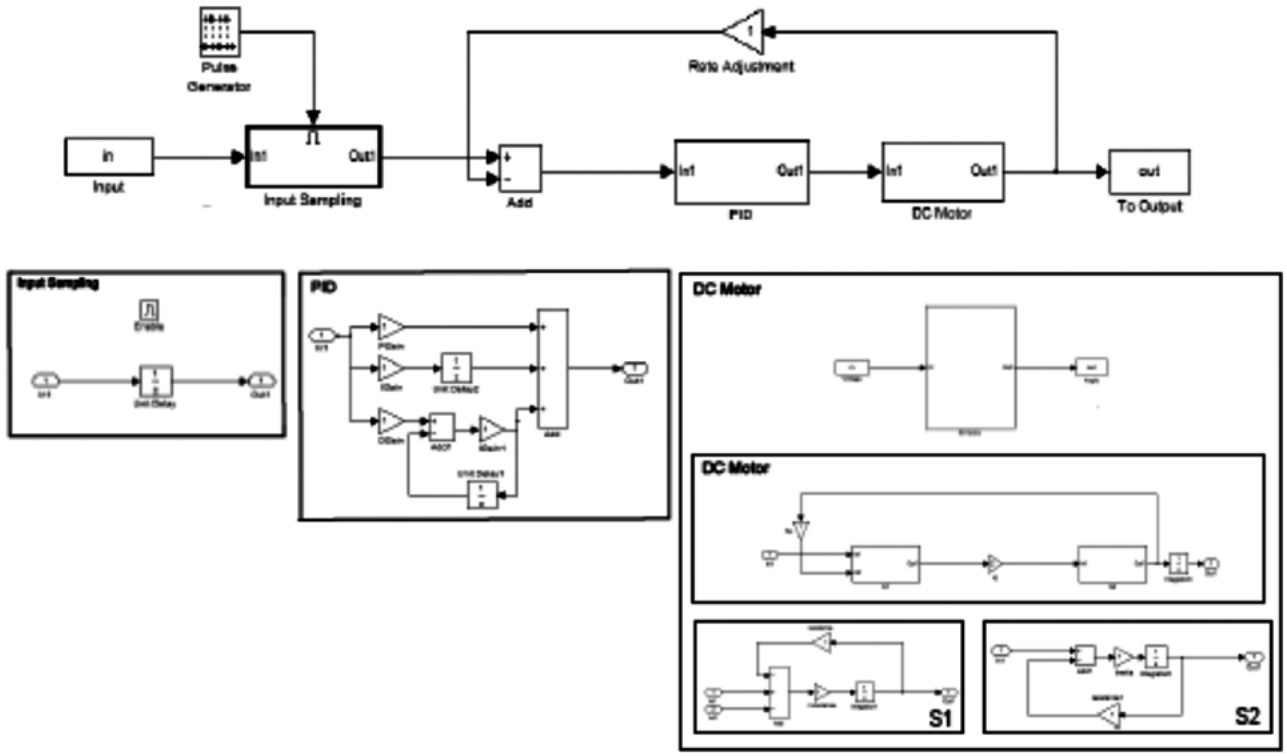

Figure 7. Case Study. Discretized DC-Motor Closed Loop Controller.

11. Conclusion

In this paper, we developed a tool that translates a discrete time subset of SIMULINK into the polychronous for-mal language SIGNAL. The motivation behind this work, lies in the lack of formal sematics of the most popular embedded software design tool, SIMULINK. Hence, the correctness of the generated models can not be completely and efficiently verified. On the other hand, formal languages, are less popular as they are harder to learn. They have, however, clear and precise semantics, that allow the application of powerful design methodologies. The choice of SIGNAL as a target language for this work, is justified by its multirate nature that allows for signal streams to be computed asynchronously, which fits very easily to a multi-threaded environment. The novelty in this work consists of bridging the gap between the synchronous and polychronous models of computation, through constructing affine clock relations between every block's inputs and outputs. This allows the generation of flow equivalent SIGNAL programs from the SIMULINK models. The translation follows three major steps: type inference, clock inference and hierarchical topdown translation. The SIGNAL program is generated by recursively translating the SIMULINK blocks. Subsystems are translated into SIGNAL processes and their enclosing blocks are translated into subprocesses. Our tool is tested on a discretized DC-Motor controller. Apart from SIGNAL code generation, our tool can be used for checking typing and timing rules of the SIMULINK models. Models that are rejected in SIMULINK are also rejected by our tool. The main drawback of this tool is its dependency on SIMULINK semantics, which keeps changing from one version to another. Besides, this tool is still incomplete, as it does not translate all the SIMULINK blocks. In fact, the behavior of many blocks is ambiguous, despite of the multiple experiments performed to understand it (ex. Enabled SubSystem). In the future, this work can be extended in different ways. One research direction would be to translate STATEFLOW to SIGNAL, since, SIMULINK and STATEFLOW are complementary tools, that are used together in many applications. Another interesting direction, would be to compare the concurrency of the SIGNAL produced C code, with the one generated from LUSTRE and the one provided by the SIMULINK code generator. This would prove the advantage of choosing SIGNAL as a target language instead of other synchronous languages. Formally proving the flow equivalence between SIMULINK and SIGNAL is also within the scope of our future work. Finally, the scalability of the tool can be further tested by applying the translation tool to more complicated SIMULINK models from the industry. In this case, the fault coverage obtained from using SIGNAL verification tools over the SIMULINK ones can be compared.

12. References

[1] `http://www.irisa.fr/`.
[2] `http://www.irisa.fr/espresso/polychrony/index.php`.
[3] `http://www.irisa.fr/vertecs/logiciels/sigali.html`.
[4] `http://www.isr.umd.edu/sites/default/files/sandeepshukla.pdf`.
[5] Simulink: User's Guide. The Mathworks.
[6] Aditya Agrawal, Gyula Simon, and Gabor Karsai. *Semantic translation of simulink/stateflow models to hybrid automata using graph transformations*. Electronic Notes in Theoretical Computer Science, 109:43-56, 2004.
[7] Julien Ouy Bijoy A. Jose, Abdoulaye Gamatie and Sandeep K. Shukla. Smt based false causal loop detection during code synthesis from polychronous specifications.
[8] Olivier Bouissou and Alexandre Chapoutot. *An operational semantics for simulink's simulation engine*. ACM SIGPLAN Notices, 47(5):129-138, 2012.
[9] Marius Dorel Bozga, Vassiliki Sfyrla, and Joseph Sifakis. *Modeling synchronous systems in bip*. In Proceedings of theseventh ACM international conference on Embedded software, pages 77-86. ACM, 2009.
[10] Abdoulaye Gamati. *Designing Embedded Systems with the SIGNAL Programming Language: Synchronous, Reactive Specification*. Springer Publishing Company, Incorporated, 1st edition, 2009.
[11] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. *The synchronous data flow programming language lustre*. Proceedings of the IEEE, 79(9):1305{1320, 1991.
[12] Thomas A Henzinger. *The theory of hybrid automata*. Springer, 2000.
[13] Bijoy A Jose and Sandeep K Shukla. *An alternative polychronous model and synthesis methodology for model-driven embedded software*. In Proceedings of the 2010 Asia and South Pacific Design Automation Conference, pages 13-18. IEEE Press, 2010.
[14] Julien Ouy Mahesh Nanjundappa, Matthew Kracht and Sandeep K. Shukla. *A new multi-threaded code synthesis methodology and tool for correct-by-construction synthesis from polychronous specifications*. 2013.
[15] Vassiliki Sfyrla, Georgios Tsiligiannis, Iris Safaka, Marius Bozga, and Joseph Sifakis. *Compositional translation of simulink models into synchronous bip*. In Industrial Embedded Systems (SIES), 2010 International Symposium on, pages 217-220. IEEE, 2010.
[16] Stavros Tripakis, Christos Sofronis, Paul Caspi, and Adrian Curic. *Translating discrete-time simulink to lustre*. ACM Transactions on Embedded Computing Systems (TECS), 4(4):779-818, 2005.