Translating Discrete Time SIMULINK to SIGNAL

Safa Messaoud

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Engineering

Sandeep K. Shukla, Chair
Michael S. Hsiao
JoAnn M. Paul

May 28, 2014
Arlington, Virginia

Keywords: SIMULINK, SIGNAL, Embedded Software, Code Generation

Translating Discrete Time SIMULINK to SIGNAL

Safa Messaoud

(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, capability 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 thesis, we present a semantic translator that transform 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 the semantic preservation of our prototype tool by testing it on different SIMULINK models.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Cyber Physical Systems (CPS) are engineering systems consisting of the integration of computational control and physical components with continuous dynamics. CPS are omnipresent in different sectors, such as agriculture, energy, transportation, building, healthcare and manufactoring. As this systems are becoming more complex and their reliability, safety and capability 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. The models have semantics derived from different theories such as finite state machines, tagged signal models [2], synchronous languages[3][4] and Metropolis meta models[5]. 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. Some of the most popular formal verification techniques for embedded software include model checking and abstraction. Model checking is used to exhaustively search through all behaviors of the finite abstraction. Abstraction is used to reduce the infinite state space systems into a finite ones.

Despite the intensive research in the model based design, the Mathwork's graphical environment SIMULINK[6] is still the most widely used tool for the design of embedded software. Although it is very convenient to use and easy to learn, 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 simulation and its behavior is strongly correlated with the simulation configuration parameter. For example, switching from a fixed to a variable simulation step alters the output traces as well. 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. Besides, little research work is done on quantifying the coverage obtained through

multiple simulations. SIMULINK has commercial code generators (Real-Time Workshop from Mathworks, TargetLink from dSpace). However, they have many restrictions. For example, TargetLink, does not generate code for blocks for the discrete Library. The preservation of semantic is another issue, since the behavior equivalence between the simulated model and the generated code is unclear.

Despite these limitations, SIMULINK remains a de-facto tool in the embedded control design for its convenience. 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 SIMULINK formal semantics[7] or translate it into formal models of computation[8][9][10]. In this thesis, we present a framework for translating discrete time SIMULINK models to SIGNAL[11] programs. SIGNAL is a data flow synchronous programming language which was developed by IRISA[12]. It does not assume the existence of any external trigger or global clock for reacting to the inputs. It is paced by the rate at which data arrives. Hence, each variable (Signal) within the software has its own clock, giving us the multi-rate (polychronous) formalism of SIGNAL. This timing model allows for streams to be computed asynchronously to one another 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[13] and the graphical developing interface SME[14] exist for Signal. These characteristics make SIGNAL an interesting model of computation for embedded software design.

In this thesis, we develop a tool that only translate the discrete time blocks of SIMULINK. This choice is justified by the fact that, controller should be modeled as discrete time systems, so that they can be implemented on a computer.We follow the same translation methodology proposed in [10], for translating LUSTRE to SIMULINK. The first two steps are clock inference and type inference. These steps provide us with information that will be needed in the atomic blocks translation and their hierarchical composition, while preserving the informal semantics of SIMULINK, given by the behavior of the simulator. The preservation of semantics is checked by running with the same input sequence, both the SIMULINK simulation and the corresponding SIGNAL program and obtaining in both cases the same output sequence. The novelty in this work consists of bridging the gap between the almost synchronous model of computation of SIMULINK into a polychronous model of computation. In the past work by [10], 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 subprocesses when multiple SIMULINK blocks have sampled inputs with varying sampling rates but can be related by affine relations.

The rest of the thesis is organized as follows: Chapter 2 is an overview of SIMULINK

and SIGNAL formalisms. Chapter 3 is a survey of the translation of SIMULINK to different models of computation. In Chapter 4, we present the translation framework. The results of applying our prototype tool on different SIMULINK models are shown in Chapter 5. We close this thesis with some concluding remarks and suggested future work.

Chapter 2

Background

2.1 Model Based Design

Figure 2.1 shows a typical design cycle of an embedded system. Traditionally, engineers deal with each stage of the process separately. Specification, design, coding, and testing are mostly done independently. Engineers rely on design documents to provide the communication between each of theses steps. This process suffers from a variety of drawbacks, including the difficulty of keeping documentation updated. Another problem is that the coding process is often removed from the general design process. These are two issues where model-based design can greatly improve the design process. Model Based Design consists of using Models of Computation that capture the specifications as well as the critical design decisions for the system in the different stages of the development life cycle. A Model of Computation (MoC) is defined as the manner in which computation and communication are being performed. Examples of *MoC* are Petri-Nets [15], Kahn Process Networks[16], synchronous languages (Esterel [3], Lustre[4]) and polychronous languages (SIGNAL).

Model driven software design tools are based on using a high level language/Model of Computation, that are translated into a lower level language (C, RTL, etc.). The advantages of this design methodology are two fold. Since the code generation is based on precise mathematical models, this code is said to be *correct-by-construction*. Besides, the verification can be done at higher levels of abstraction, which will reduce the costs, as well as the time to market. Table 2.1 gives an overview on the abstraction levels of the tools, programming languages and their implementation paradigms. Properties of the synchronous and polychronous languages like reactivity, concurrency and deterministic execution fit the requirements of the safety critical embedded software.

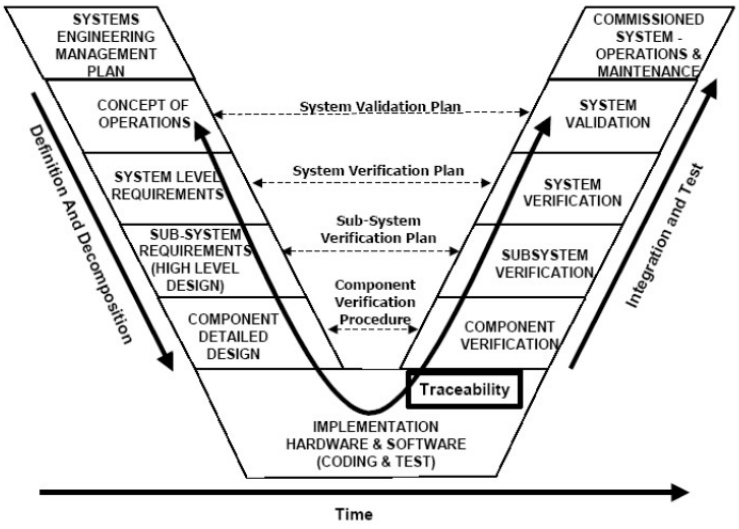


Figure 2.1: The V Design Process. http://www.engineering.com/DesignSoftware. Used under fair use, 2014

Table 2.1: Abstraction Levels of Software Languages, Directives, Utilities and Tools

Model Driven Tools	Programming Languages	Implementation Level
LabVIEW	MATLAB	Java
SIMULINK	ESTEREL	RTL
Polychrony	SIGNAL	C++
ESTEREL Studio	LUSTRE	C
SCADE		

2.2 SIMULINK

SIMULINK[6] is a computer-aided model driven design tool for embedded systems. It provides an interactive graphical environment and a customizable set of block libraries for the design, simulation, implementation, and test a variety of time-varying systems, including communications, controls, signal, video and image processing. In SIMULINK, a system is hierarchically modeled using a set of blocks that are interconnected by ports. The blocks can be categorized into virtual and non virtual blocks. A virtual block only defines the interconnections of signals and has no memory element. Examples of virtual blocks are the multiplexer *Mux*, *Outputport* and *subsystems*. Non virtual Blocks represent a set of equations, called block methods, which define a relation between the block inputs , outputs and the state variables. Examples of non virtual blocks are *Gain* and *Sum*. The signals are the communication conduits through which the blocks communicate with each others. We

distinguish between data ports, defining data flow connection endpoints, and control ports, producing triggering or enabling events for the execution of subsystems. SIMULINK enables to incorporate MATLAB algorithms into models and export simulation results to MATLAB for further analysis. SIMULINK has a multitude of semantics which depend on the user configuration. These semantics are only partially documented, mostly in natural language. The absence of formal semantics, makes the application of the static analysis techniques ineffective and restricts the verification of models to simulation based testing.

2.3 Synchronous Programming Languages

In order to better explain the motivation behind choosing SIGNAL for being the translation target language, we start by introducing the synchronous formalism. The essence of synchronous model of computation is the *synchrony hypothesis* which assumes that the computation and communication time are instantaneous. In other words, time is organized as a sequence of instants. In each *instant*, input events possibly occur, computation take place and control and data are propagated until an output is produced. This execution cycle at each instant is called a *reaction*. Hence, the history of a system is a totally ordered sequence of logical instants. Only the simultaneity or precedence between events matters. Thus, synchronous systems can be characterized by a global clock that acts as a reference for each round of the input-output events. The global clock has no relation to the hardware clock, it only ticks at the different instants. This simple mathematical model makes it easier to reason about the system, as it is deterministic (the output of the system entirely depends on the inputs values and instants at which they occur) and predictable. Programs for synchronous languages can be efficiently compiled into code for a finite state machine that can be executed on a single processor. Although the synchrony assumption might seem unrealistic at the beginning, it can be easily validated in the implementation phase, on an actual execution platform on which the time of reactions is fully taken into account. Real time constraints should be satisfied by the platform, so that satisfactory bounded delays of the reactions are ensured. Examples of synchronous languages are Lustre[4] and Esterel [3].

2.4 The Multi-rate Synchronous Language SIGNAL

SIGNAL is a declarative multi-rate synchronous language. While the synchronous languages have a totally ordered model of logical time, SIGNAL's model of logical time is partially ordered. As some variables are independent from each other until the end of the program, their instants are incomparable, leading the notion of partial order of time. 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.

2.4.1 Preliminaries

The basic entity in a polychronous language is an *event*.

Definition 2.1. (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 a binary relations, over Ξ. Such relations define whether or not an event happened before, simultaneously to, or after another event in the system, or if their relative occurrences do not matter. These relations are defined below:

Definition 2.2. (Precedence, Preorder, Equivalence). Let $\prec$ be a precedence relation between events in Ξ. It is defined such that $\forall a, b \in \Xi, a \prec b$ if and only if a occurs before b. The relation $\preceq$ defines a partial order on Ξ such that $\forall a, b \in \Xi, a \preceq b$ if and only if a occurs before b or a, b occur logically simultaneously, or their order does not matter. Finally the equivalence relation $\sim$, is defined on Ξ such that $a \sim b = a \preceq b \wedge b \preceq a$, meaning that a and b are equivalent only if they occur simultaneously or their order does not matter. Thus $\sim$ 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. A formal definition of instants follows:

Definition 2.3. (Logical Instant or Instant). The set of all instants is denoted by Υ. Each instant in Υ can be seen as an equivalence partition obtained by taking the quotient of Ξ with respect to $\sim$ such that $\Upsilon = \Xi / \sim$. For each set $S \in \Upsilon$, all events in S will have the property $\forall a, b \in S, a \sim b$, and $\forall a, b, (a \in S_1 \wedge b \in S_2 \wedge S_1 \neq S_2 \wedge S_1, S_2 \in \Upsilon \rightarrow a \nsim b$.

Because all instants are equivalence classes, a precedence relationship can be drawn between instants. We define the relationship $\prec$ between two sets in Υ such that, $S, T \in \Upsilon, S \prec T$ if and only if $\forall (s, t) \in S \times T, s \prec t$. Each instant contains events on signals. If a signal has no event in an instant then it is considered absent.

Definition 2.4. (Signal) Let T by the domain of values that a signal can take and let $\perp$ denote a special absent value such that $T_\perp = T \cup \perp$; then a *signal* can be defined as a total function of type $\Upsilon \rightarrow T_\perp$. This means that for each instant in Υ, a value or absence of value is implied on a signal x. We denote the set of all events in a signal x as $E(x)$.

We denote a specific value of a signal x by function $x(t)$ where $t \in \mathbb{N}$ and t represents the t^{th} instant in the totally ordered set of instants where signal x has a value different from $\perp$. $x(t)$ thus returns the t^{th} event value in the signal x.

Definition 2.5. (Epoch, Clock) The *epoch*, $\sigma(x)$, of a signal x is the maximum set of instants in Υ where for each instant in $\sigma(x)$, 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 $\Upsilon \rightarrow [true, false]$ such that for a signal x it returns another signal $\hat{x}$ defined by $\hat{x}(t) = true$ if $x(t) \in T$.

Note that not all inputs and outputs are present or computed during every instant in Υ 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, $\hat{x} = \hat{y}$, 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** [17]. It is obvious that some specific subsets of relationships may be drawn from clocks that are unrelated. One type of relationship is mutual exclusion, meaning that $x(t) = true$ if and only if $y(t) = \perp$ and $y(t) = true$ if and only if $x(t) = \perp$. These relationships are stored in a structure called a *clock tree*.

2.4.2 The SIGNAL Formalism

Primitive Signal operators are:

1. **Function:**
   The function operator performs user defined operations on a set of signals x_1,x_2, $\ldots x_n$ that must be present simultaneously and produces an output y at the same logical instant.

   Operation: $y{:=}f(x_1,x_2,\ldots x_n)$
   Clock Relation: $\hat{y} = \hat{x}_1 = \hat{x}_2 = \ldots \hat{x}_n$

   An example of an AND operation $y{:=}$AND(x,z) ,where a boolean signal is represented by true (t) or a false (f) is shown below.
   $\perp$ represents an absent event and each column represents an instant.

   x: $f \perp t\ t$
   z: $t \perp f\ t$
   y: $f \perp f\ t$

2. **Delay:**
   The delay operator in SIGNAL 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 to each other.

Operation: *y:=x $ init k*
Clock Relation: $\hat{y}=\hat{x}$

An example of the delay operation is given below:

$$\begin{array}{ll} x: & v_1\ v_2\ v_3 \\ y: & k\ \ v_1\ v_2 \end{array}$$

3. **Under sampling:**
   The under sampling operator down samples an input signal x based on a given condition, namely 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]*.

   Operation: *y:=x when z*
   Clock Relation: $\hat{y}=\hat{x}$ * *[z]*

   An example of a sampler is shown below:

$$\begin{array}{ll} x: & v_1\ v_2\ v_3\ v_4 \\ z: & \perp\ \ t\ \ f\ \ t \\ y: & \perp\ v_2\ \perp\ v_4 \end{array}$$

4. **Priority Merging:**
   This 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 as well.

   Operation: *y:=x default z*
   Clock Relation: $\hat{y}=\hat{x}+\hat{z}$

   An example of a default operation is shown below:

$$\begin{array}{ll} x: & \perp\ \ v_2\ \ v_3\ \ v_4 \\ z: & w_1\ \ \perp\ \ w_3\ \ \perp \\ y: & w_1\ \ v_2\ \ v_3\ \ v_4 \end{array}$$

The primitive SIGNAL operators and their corresponding clock relations are summarized in the following table:

Table 2.2: SIGNAL Primitive Operators and Clock Relations

Operator	Expression	Clock Relation
Function	*y:=f(x_1,x_2,... x_n)*	$\hat{y} = \hat{x}_1 = \hat{x}_2 = \ldots \hat{x}_n$
Delay	*y:=x $ init k*	$\hat{y}=\hat{x}$
Sampler	*y:=x when z*	$\hat{y}=\hat{x}$ * *[z]*
Merge	*y:=x default z*	$\hat{y}=\hat{x} + \hat{z}$

2.4.3 Advanced SIGNAL Constructs

Clock relations are not only inferred from the SIGNAL statement, they can be given explicitly[11]:

- 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
- The equation *clk := s1 ^* s2* specifies the signal *clk* as the intersection of the clocks of the signals *s1* and *s2*.
- The equation *clk := s1 ^+ s2* specifies the signal *clk* as the union of the clocks of the signals *s1* and *s2*.
- The equation *clk := s1 ^- s2* specifies the signal *clk* as the difference of the clocks of the signals *s1* and *s2*.

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:

$$\begin{array}{llllll} x: & \perp & v_2 & v_3 & \perp \\ z: & t & t & \perp & f & t \\ y: & k & v_2 & v_3 & v_3 \end{array}$$

2.4.4 SIGNAL Processes

A process is set of signal definitions specifying relations between values on the one hand and clocks on the other hand of involved signals [11]. 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 interface%
process MODEL =
{ %parameters%}
   ( ? %inputs%;
   ! %outputs%; )
   (|
   %body of the process%
   |)
where
   %local declarations%
end;
%end of MODEL%
```

An example of a process is shown below:

```
Process Sum =
   ( ? integer s;
   ! integer sum;)
   (| Sum := OldSum+s
   | OldSum := Sum $ init 0
   |)
where
   integer OldSum;
end;
```

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

2.4.5 Endochrony and Weak Endochrony

Endochrony and weak endochrony are properties of polychronous programs, describing the schedulability of their computation. They determine whether a specification has deterministic order of execution, which is required by compilers for sequential or multi-threaded code synthesis.

Definition 2.6. (Endochrony) A SIGNAL process is endochronous, if and only if the scheduling of the computation based on the events arriving at the input signals can be correctly inferred without any additional information from the environment during the runtime of the process. This means that a correct scheduling of the internal computations to

produce the correct sequence of output events can be statically scheduled.

In order to better illustrate the endochrony notion, we introduce Process ENDO [18]. The inputs x and z are synchronous, since they have to be present at the same time, otherwise the computation of val is not possible. The second equation tells that z and y are synchronous as well. In this case the scheduling of the computation can be determined statically: first read the inputs x and y, then evaluate z and val. So ENDO is an endochronous process.

```
Process ENDO =
   ( ? integer x, y;
   ! integer val;)
   (| val := x + z $ init 0
   | z := y + 1 $ init 0
   |)
where
   integer z;
end;
```

Definition 2.7. (Weak Endochrony) A SIGNAL process is weakly endochronous, if and only if, it has the 'diamond property' [19]. In other words, for such a process, even when a complete static schedule of the computations is not possible, order of the computations may dynamically depend on the occurrence of events on the input signals, the computation is confluent. The confluence here means that irrespective of the dynamic order of computation, the final state of a reaction is the same as the case where all input events occur synchronously.

WEAKENDO is an example of a weakly endochronous process [18]. The clocks of a and b can not be inferred from the program. Hence, a static schedule can not be determined. However, regardless of the order of occurrence of a and b, the final result after a pair a and b is read, is the same.

```
Process WEAKENDO =
   ( ? boolean a, b;
   ! integer val1, val2;)
   (| val1 := (x + 1) when a
   | val2 := (z + 1) when b
   | x := val1 $ init 0
   | z := val2 $ init 0
   |)
where
   integer x, z;
end;
```

2.4.6 Uses of SIGNAL

A great advantage of SIGNAL is its convenience for component based design approaches that allow the incremental modular development of complex systems, since each component has its own clock. SIGNAL is very adequate for modeling the globally synchronous locally asynchronous systems (GALS). Besides, as multi-rate models do not have a global clock, they give more freedom to compilers to chose from different schedules of computation. This gives the opportunity for more optimized code synthesis. Also, contrary to the synchronous languages, SIGNAL natively embraces the multi-clock version, and hence multi-threading is easier to implement.

2.5 Comparison between SIMULINK and SIGNAL

Both SIMULINK and SIGNAL are data-flow languages. They both manipulate *Signals*. A signal is a function of time. A system performs a specified operation on an input signal and produces an output signal. In SIMULINK, signals correspond to the wires that connect the blocks in a model. In SIGNAL, a signal is the program variable corresponding to a *stream*. 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 in 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 this similarities, SIMULINK and SIGNAL are different is 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[3]. Even the blocks belonging to the discrete library produce piece-wise 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*. The typing mechanisms of both languages are discussed more in details in Section 4.3.
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. A more detailed description of the timing mechanisms of both SIMULINK and SIGNAL will follow in Section 4.5.

Chapter 3

Related Work

In this section, we discuss the past research aimed at formalizing SIMULINK modeling language. 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 like Lustre, into hybrid automata, or I/O extended Finite Automata or into a system of mathematical equations.

3.1 Translating SIMULINK to a Formal Language

Translating SIMULINK models to a formal langage is one of the most popular solutions for dealing with the informal semantics of SIMULINK. The main motivation for this approach lies in gaining access to the analysis and verification tools of the target language.

3.1.1 SIMULINK to LUSTRE

Lustre is a synchronous data flow language. A number of formal validation tools exist for Lustre, such as the model checker Lesar [20], the tester [21] and the graphical design environment commercialized by Esterel Technologies, SCADE [22]. SCADE also allows C code generation. As Lustre has discrete semantics, while SIMULINK has hybrid one, only a subset of the SIMULINK models, namely the discrete ones can be safely translated. The translation flow, followed in [10] starts with clock and type inference, to derive useful information to be used in the hierarchical bottom 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.

3.1.2 SIMULINK to Synchronous BIP

Synchronous BIP [23] is a subset of the BIP framework for data-flow modeling. BIP stands for "Behavior, Interation, Priority". It is a component framework. Each component is the superposition of three layers: Behavior, described by Automata extended with C Code, Interaction specifying the relationship between behavioral actions and Priority, providing scheduling rules for the component interactions. A system is then described as the composition of generic atomic components. Similar to Lustre, BIP has discrete semantics. It also has a toolset for design verification and automatic code generation. The translation from SIMULINK to BIP[8] is compositional: each SIMULINK component is translated into a BIP component. Basic blocks are translated into their equivalent elementary blocks in BIP. Structured blocks like subsystems are translated recursively.

The disadvantage of choosing the formal methods as a target language for the translation lies in their restriction to discrete time semantics.

3.2 Translating SIMULINK to a Hybrid Automata

Unlike the formal languages, Hybrid Automata[24] (HA) are designed to describe systems with both continuous and discrete semantics. HA is a nite state machine with a nite set of continuous variables whose values are described by a set of ordinary differential equations. A HA is characterized by a tuple (M,M_0,Σ,X,Δ,I,F,V_0):

- M: is a finite set of control modes
- M_0: is the initial state of control modes
- Σ: is a finite set of actions
- X: is a finite set of real valued variables
- $\Delta \subseteq M \times \text{pred}(X) \times \Sigma \times \text{pred}(X \cup \acute{X})$
- I: $M \rightarrow$ is the mode invariant function
- F: $M \rightarrow \text{pred}(X \cup \acute{X})$ is the mode dependent flow function
- $V_0 \in \text{pred}(X)$ is the set of initial valuations

In [9], 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 (single writer multiple reader variables) and shared variables. Signal propagation among automata follow the topological order of dependencies. HSI is characterized by a tuple (HA, V, P, C).

- HA: Hybrid Automata
- V: Finite Set of Variables(Inputs, Outputs, Shared Variables, Local Variables)
- P: Set of Parameters
- C: Input Constraint

Many simulation, verification and code generation tools have HSIF import or export ports. The translation was limited to continuous blocks (Integrator, Zero-Pole, ...), Mathematial Operators (excluding logical blocks), Sources (Constant, In), Sink (Out), Switch and STATEFLOW Diagrams. The translation from SIMULINK/STATEFLOW is based on graph transformation. :

1. **Enumeration of switching signals:** Switching signals need to be identified, since they change the structure of the dynamics.
2. **Transformation of states to locations:** The number of locations in HSIF is determined by the Stateflow machine, since all the switching signals originate from there.
3. **Transformation of state transition to location transitions:** One state transition might be mapped to many transitions in HSIF.
4. **Generation of equations:** Differential and algebraic equations are generated for each location based on the SIMULINK diagram.
5. **Generation of invariants:** Invariants are generated from transition conditions and STATEFLOW variables.
6. **Pruning unreachable locations:** Unreachable locations are deleted.

Similar to the formal languages,HA have the advantage of available static analysis tools and methods. They can also model both continuous and discrete systems. However, they have not been formally standardized.

3.3 Translating SIMULINK to a System of Equations

Chapoutot et. al.[7] 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 (3.1). Continuous time $f_x : \mathbb{R} \times \mathbb{R}^{n_x} \times \mathbb{R}^{n_d} \longrightarrow \mathbb{R}^{n_x}$ as well as discrete time $f_d : \mathbb{R} \times \mathbb{R}^{n_x} \times \mathbb{R}^{n_d} \longrightarrow \mathbb{R}^{n_d}$ state functions, are considered to represent the fact that SIMULINK models are hybrid systems. The output function is $g : \mathbb{R} \times \mathbb{R}^{n_x} \times \mathbb{R}^{n_d} \longrightarrow \mathbb{R}^m$. Here, x is the continuous state, d is the discrete state,

n_x is the number of continuous state variables, n_d is the number of discrete state variables and m is the number of outputs.

$$\begin{aligned} \dot{x}(t) &= f_x(t, x(t), d(t)) && (3.1)\\ \dot{d}(t) &= f_d(t, x(t), d(t)) && (3.2)\\ y &= g(t, x(t), d(t)) && (3.3)\\ d(0) &= d_0 && (3.4)\\ x(0) &= x_0 && (3.5) \end{aligned}$$

The simulation goal consists of finding the solution for the set of equations (3.1). The simulation algorithm can be formulated as follows:

Algorithm 1: Simulation Algorithm [7]

Data: Input: x_0,d_0,h_0,t_0
n=0;
while *($t_n > t_{end}$)* **do**
  Evaluate $g(t_n, x_n, d_n)$
  Compute $\dot{d}(t) = f_d(t, x(t), d(t))$
  Solve $\dot{x}(t) = f_x(t, x(t), d(t)) \; \forall \; t \in [t_n, t_n + h_n]$
  Check for zero-crossing
  Compute h_{n+1}, t_{n+1}
end

t is the time and h is the integration step size. f_d and f_x are derived from the equations of the single SIMULINK blocks by inspection of the *.mdl* file. The following BNF grammar is used for the equation generation:

$$\begin{aligned} e &::= r|l|x|d|e_1 \diamond e_2|e_1 \triangle e_2|if(e_1, e_2, e_3) && (3.6)\\ eq &::= l :=_s e|l := e|\dot{x} := e|\dot{d} :=_s e && (3.7)\\ p &::= eq|eq; p && (3.8) \end{aligned}$$

In this context, $\diamond$ represents the arithmetic operations, $\triangle$ represents the boolean operations, x is the continuous state variable, d is the discrete state variable, r is a real constant and l is the output of a given block. For the purpose of the simulation, the equations can be categorized into four subsets:

- **Major Step Equations**: of the form $l :=_s e$ or $l := e$, are basically all equations sampled and continuous
- **Minor Step Equations**: are the subset of Major Step Equations that are without any time sampling
- **Update Step Equations**: of the form $\dot{d} :=_s e$, correspond to discrete blocks

- **Solver Step Equations**: of the form $\dot{x} := e$, correspond to continuous blocks

These subsets of equations evaluate one after the other during the simulation loop. Table 3.1 shows examples of the formulas derived for some blocks. This approach was validated by

Table 3.1: SIMULINK Blocks Equations Using the BNF grammar

Blocks	Equations
Constant	$l_1 = constant$
Output	$out_1 = l_1$
Add	$l_3 = l_1 + l_2$
Switch	$l_4 = if(p_r(l_2), l_1, l_3)$
Integrator	$l_2 = x; \dot{x} = l_1; x(0) = init$
Unit Delay	$l_2 = d; \dot{d} = l_1; d(0) = init$

comparing the outputs of the SIMULINK simulator and the simulator based on the system of equations for three case studies. The system of equations approach is able to cover both discrete and continuous blocks, similarly to the HA. However, this approach lacks tools for the equation grammar verification and simulation.

3.4 Contribution of the Thesis

Since in safety critical systems, we are only interested in implementing the controller on a digital computer, it is sufficient to use a model of computation with discrete semantics. 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 makes SIGNAL an interesting candidate for the design of such systems. This justifies our motivation for choosing SIGNAL as a target language.
Although we adopt a similar translation flow as in [10], the novelty in this thesis consists in bridging the gap between the almost synchronous model of computation of SIMULINK into a polychronous model of computation. Unlike Lustre, SIGNAL does not have an external global clock. However, if the SIMULINK model has the endochronous property, a global clock can be derived using the clock calculus. If a global clock cannot be found, a polychronous model leading to multi-threaded behavior emerges. We replace the sampling mechanism in SIMULINK by the generation of affine clock relations between the subprocesses of SIGNAL, which would guarantee the equivalence between the traces generated by SIGNAL and the ones resulting from SIMULINK. If not used for the synthesis of SIGNAL,

our tool can be utilized for type checking and clock inference, which detect errors in the SIMULINK model.

Chapter 4

Translating discrete time SIMULINK to SIGNAL

4.1 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 of the simulation mode. Our tool rejects the models with typing or timing errors flagged by SIMULINK. We limit our translation to the discrete time part of SIMULINK into Signal. This is justified by the fact that only the controller in safety critical systems is implemented on the computer, hence the controller must be designed in discrete time. The list of supported SIMULINK blocks is (see Figure 4.1):

- **Input/Output Blocks:** From Workspace, To Workspace, Output, Input, Constant, Pulse Generator
- **Discrete Time Blocks:** Difference, Unit Delay, Integer Delay, Zero-order Hold, Discrete Filter, Discrete Transfer Function
- **Math and Logic Operations:** Add, Product, Gain, Saturation, Relational Operator.
- **Virtual Blocks:** Mux, Switch, Data Type Conversion, Subsystem, Enabled Subsystem, Triggered Subsystem

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.

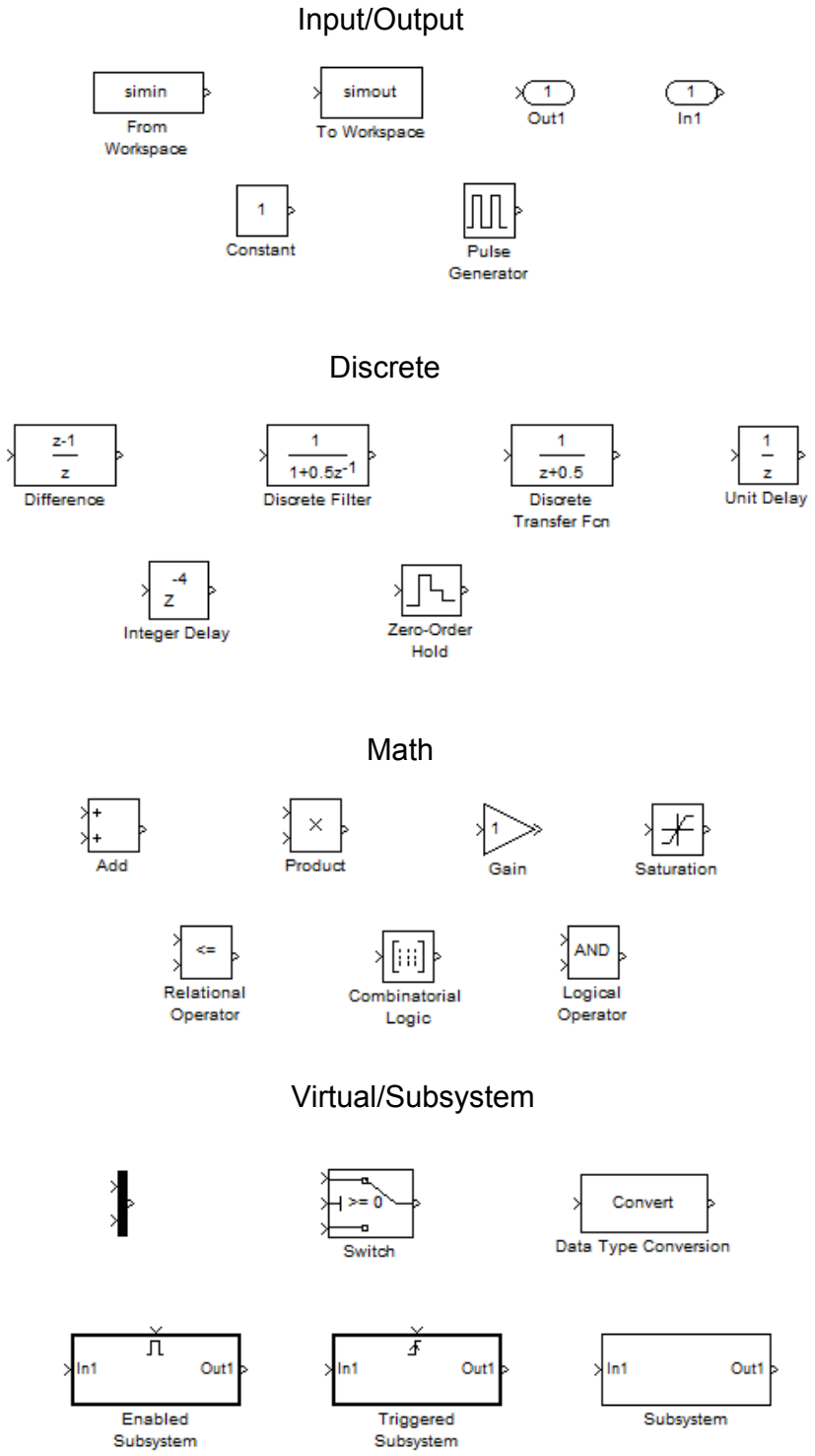


Figure 4.1: Supported SIMULINK Blocks

We have developed our translation method based on MATLAB 7.12.01 (R2011a) and SIMULINK block Library V7.7.

4.2 Translation Flow

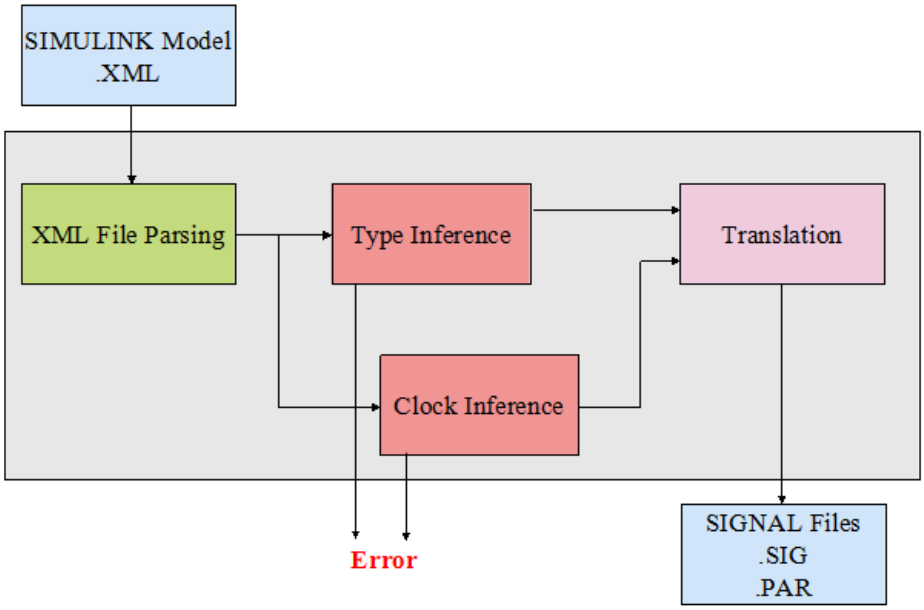


Figure 4.2: Translation Framework

As it is shown in Figure 4.2, the translation involves in five steps:

- **XML File Generation and Parsing:** The SIMULINK model (*.mdl*) can be directly saved as an *XML* file in the SIMULINK environment. Our tool parses the *XML* file using a *C++ XML* parser called *pugiXML* and builds a data structure representing the original SIMULINK model.
- **Type Inference:** In this step, type checking of the SIMULINK model is performed. In case of type incompatibilities, the model is rejected. Otherwise, the types of signals in SIMULINK are inferred and mapped to their corresponding ones in SIGNAL (See Section 4.3).
- **Clock Inference:** In this step, timing rules for SIMULINK are checked and invalid models are rejected. Otherwise, the sampling period and phase for each block are inferred. From the generated timing information, affine clock relations between the single blocks inferred and used to generate a semantically equivalent SIGNAL program (See Section 4.5).

- **Translation:** The translation is done in a hierarchical way, each subsystem is translated to a SIGNAL program, the blocks in the subsystem are translated into subprocesses in the parent process. The clock relations and types from the previous step are used for characterization and rate adaption between the processes (See Section 4.7 and Section 4.8).

4.3 Type Inference

4.3.1 Types in SIMULINK

Unlike 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:

1. 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.
2. The user can explicitly set the type of a signal to another type (e.g., by a *Data Type Converter Block*)
3. 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 by Table 4.1. 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 $\{\gamma,\beta\} \in \{T_{Bool}, T_{Num}\}$

4.3.2 Types in SIGNAL

SIGNAL is a strongly typed language. This means that 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 similarly to 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. The notation is as follows: *[inp$_1$, ...,inp$_N$] element_type*. This means an array of size N with elements of type *element_type*.

Table 4.1: Typing Rules for Some SIMULINK blocks

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

4.3.3 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 during the translation. For the type inference, we use a fix-point algorithm on the lattice shown in Figure 4.3. $\perp$ 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*: $(T_{Sim})^n \longrightarrow T_{Sim}$ in the type lattice, with n 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 equations shown in Table 4.2.

To implement the type inference algorithm, we consider the *Type Matrix* corresponding to

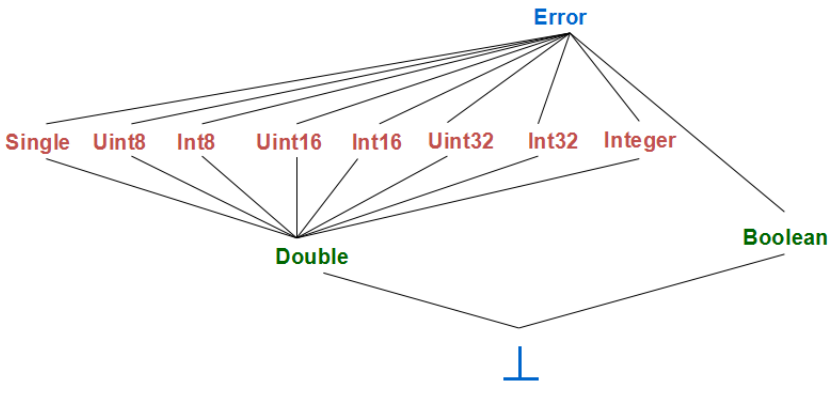


Figure 4.3: The Type Lattice

Table 4.2: Type Inference Equations. Stavros Tripakis, Christos Sofronis, Paul Caspi, and Adrian Curic. Translating discretetime simulink to lustre. ACM Transactions on Embedded Computing Systems (TECS), 4(4):779818, 2005. Used under fair use, 2014

SIMULINK Equation	Type Equation
$y = Adder(x_1, \ldots, x_k)$	$y^T = x_1^T = \cdots = x_k^T = Sup(double, y^T, x_1^T, \ldots, x_k^T)$
$y = Constant_\alpha$	y^T=if $y^T \leqslant \alpha$ then α else $error$
$y = DataTypeConverter\alpha(x)$	y^T=if $y^T \leqslant \alpha$ then α else $error$
$y = UnitDelay(x)$	x^T=y^T
$y = RelationalOperator(x_1, x_2)$	x_1^T=x_2^T=sup(double,x_1^T,x_2^T),y^T=boolean
$y = LogicalOperator(x_1, \ldots, x_k)$	x_1^T=x_2^T=x_1^T= $\ldots$ = x_k^T=y^T=boolean
$y = switch(x_1, x_2, x_3)$	x_1^T=x_1^T=y^T=sup(x_1^T,x_3^T,y^T)

the SIMULINK Block. The *Type Matrix* for a model with three components A, B and C is:

$$TypeMatrix_{m,n} = \begin{pmatrix} t_{AA} & t_{AB} & t_{AC} \\ t_{BA} & t_{BB} & t_{BC} \\ t_{CA} & t_{CB} & t_{CC} \end{pmatrix} \tag{4.1}$$

The corresponding *Type Matrix* for the SIMULINK model shown in Figure 4.4 is:

$$TypeMatrix_{m,n} = \begin{pmatrix} Undef & \mathbf{Double} & Undef & Undef & Undef & Undef \\ \mathbf{Double} & Undef & \mathbf{Double} & Undef & Undef & Undef \\ Undef & \mathbf{Double} & Undef & Undef & \mathbf{Double} & \mathbf{Double} \\ Undef & Undef & Undef & Undef & \mathbf{Double} & Undef \\ Undef & Undef & \mathbf{Single} & \mathbf{Double} & Undef & Undef \\ Undef & Undef & \mathbf{Double} & Undef & Undef & Undef \end{pmatrix} \tag{4.2}$$

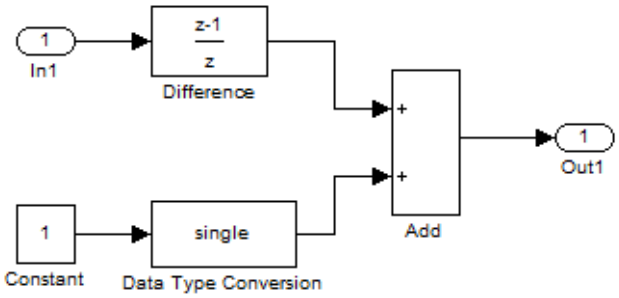


Figure 4.4: The Type Matrix Generation Example

t_{ij} corresponds to the *undefined* type, if there is no connection between the components i and j in the SMULINK model M, otherwise t_{ij} corresponds to the output type of the

component i and t_{ji} is the input type of component j. The fixed point algorithm consists of calculation T_Matrix_{n+1} from (T_Matrix_n), so that the equations in Table 4.2 are satisfied, until T_Matrix_{n+1}=T_Matrix_n. Each column i in T_Matrix_{n+1} contains all the types of the inputs and outputs of the block i. The inferred type of a given block b_i inputs (T_Inp) and outputs (T_Out), results from evaluating the function *Sup* over all the elements of the column i. This is valid for most of the blocks whose inputs and outputs are from the same type (e.g, Unit Delay, Adder). For the blocks, whose outputs and inputs are from a different types (e.g, Logical Operator, Data Type Converter), we apply the *Sup* function separately on the inputs and outputs. The types in T_Matrix_n are then replaced by the newly inferred type. We repeat this process until T_Matrix does not change between two successive iterations ($T\_Matrix_n = T\_Matrix_{n+1}$). In each iteration, we check whether a type incompatibility occurs, in which case the model is rejected. Our algorithm is slightly different from the conventional fixed point ones, since the block input and output types are initialized with the types double or boolean (not $\perp$) according to the block type. In the first iteration we check the T_Matrix symmetry, this is equivalent to checking whether a signal is assigned two different types. If the types are incompatible, the model is rejected, otherwise the signal type is set to *Sub(T_Matrix_0[i][j],T_Matrix_0[j][i])*. The type inference algorithm is shown in Alg 2.

To illustrate the discussed Algorithm, we consider two examples of SIMULINK models. The First example (See Figure 4.5) illustrates the case, where a model is rejected because of type incompatibility between the input type of *B2* and the output type of *B1*, since *Sup(T_Matrix[1][2],T_Matrix[2][1])=Sup(bool, double)=error*. The iteration steps are shown in Table 4.3.3.

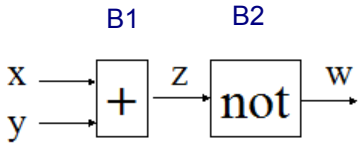


Figure 4.5: Type Inference (Example 1)

Iteration 1	Iteration 2
$\begin{pmatrix} \textit{Undefined} & \textbf{Double} \\ \textbf{Boolean} & \textit{Undefined} \end{pmatrix}$	TYPING ERROR!

Algorithm 2: Type Inference Algorithm

Data: Input: T_Matrix_0
n=0;
forall the *(blocks b_i AND b_j in M)* **do**
  if *(T_Matrix_0[i][j] != T_Matrix_0[j][i])* **then**
    Sp = Sup (T_Matrix_0[i][j],T_Matrix_0[j][i])
    if *(Sp = Error)* **then**
      return TYPING ERROR
    else
      T_Matrix_0[i][j] = Sp
      T_Matrix_0[j][i] = Sp
    end
  end
end
while *((n==0) **OR** (T_Matrix_{n+1} != T_Matrix_n))* **do**
  if *(n > 0)* **then**
    n++
  end
  forall the *all blocks b_i of M* **do**
    if *((b_i.Type = DataTyperConverter) OR (b_i.Type = RelationalOperator))* **then**
      T_Inp = Sup (T_Matrix_n[1][i],...,T_Matrix_n[n][i],$Inputs$[i])
      T_Out = Sup (T_Matrix_n[1][i],...,T_Matrix_n[n][i],$Outputs$[i])
    else
      T_Inp = T_Out = Sup(T_Matrix_n[1][i],...,T_Matrix_n[n][i],$Inputs$[i],$Outputs$[i])
    end
    if *(Sp = Error)* **then**
      return TYPING ERROR
    else
      T_Matrix_{n+1} = Replace (T_Inp,T_Out,T_Matrix_n)
    end
  end
end

For the model shown in Figure 4.6, two iteration steps are performed (See Table 4.3.3). As *Sup(T_Matrix[1][2],T_Matrix[2][1])=Sup(int8, double)=int8*, the inferred types are:

$$\begin{bmatrix} \mathrm{u}^T \\ \mathrm{x}^T \\ \mathrm{y}^T \\ \mathrm{z}^T \end{bmatrix} = \begin{bmatrix} \text{Double} \\ \text{Int8} \\ \text{Int8} \\ \text{Int8} \end{bmatrix} \tag{4.3}$$

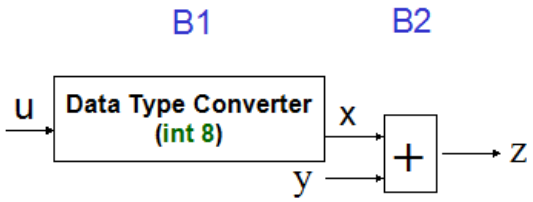


Figure 4.6: Type Inference (Example 2)

Iteration 1	Iteration 2
$\begin{pmatrix} \textit{Undefined} & \textbf{Int8} \\ \textbf{Double} & \textit{Undefined} \end{pmatrix}$	$\begin{pmatrix} \textit{Undefined} & \textbf{Int8} \\ \textbf{Int8} & \textit{Undefined} \end{pmatrix}$

Hierarchical SIMULINK models are flattened, in order to generate a Type Matrix, where only basic blocks are related to each others. For example, the Type Matrix of the model in Figure 4.7 is the same as the one generated from the model in Figure 4.6

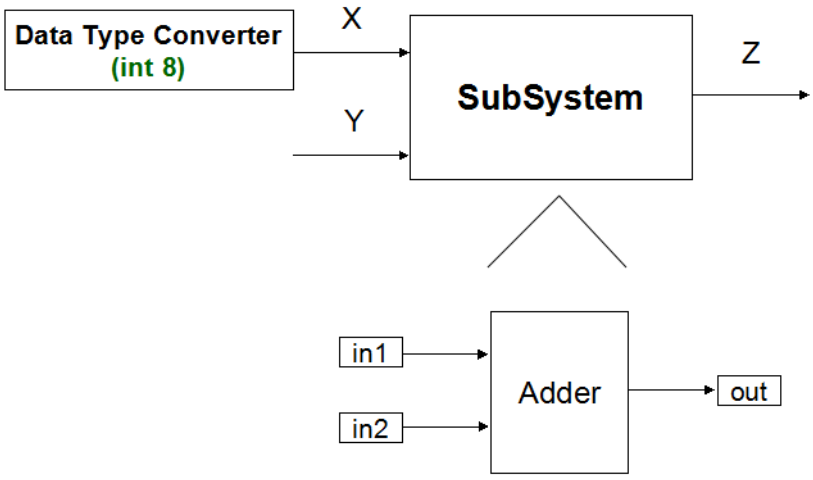


Figure 4.7: Type Inference (Example 3)

4.4 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 4.3.

Table 4.3: SIGNAL Primitive Operators and Clock Relations

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

4.5 Clock Inference

4.5.1 Time in SIMULINK

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

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 $k\pi+\theta$ (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 basic block B (apart from Subsystems) must have sample times that are multiplier or divisor of the block sample time.
  $(\pi_{Inp_{1\ldots n}} = k\ \pi_B)$ **OR** $(\pi_{Inp_{1\ldots n}} = \frac{1}{k}\ \pi_B)$, with $k = 0, \ldots, n$
- Enabled and Triggered Subsystems' inputs should have the same sample times.
  $\pi_{Inp_1} = \cdots = \pi_{Inp_n}$

Figure 4.8 shows an example of the execution flow of a SIMULINK model. In the Adder block undersampling with a factor of 2 is performed .

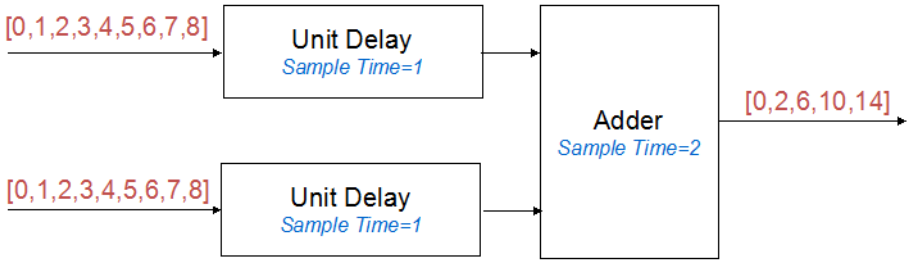


Figure 4.8: Timing Mechanism in SIMULINK (Sample Time)

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 triggering signal. Figure 4.10 shows the execution flow of a triggered subsystem. We assume that the triggered subsystem has no sub-blocks inside. The Subsystem's input is directly linked to its output. Only the inpu values happening at a rising trigger signal (transition from 0 to 1) are emitted at the output.

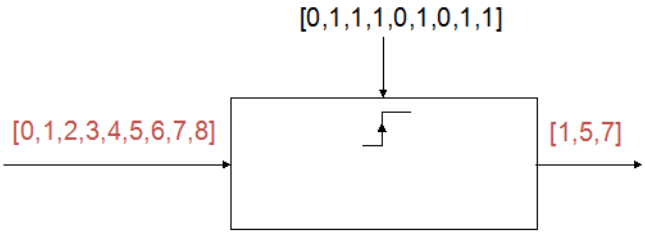


Figure 4.9: Timing Mechanism in SIMULINK (Triggered Sub-System)

Enabled Subsystem

The timing mechanism of the enabled subsystem is ambiguous [10]. 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 the block is evaluated if the *Enable* signal is equal to 1. Figure 4.9 shows the execution flow of an enabled subsystem. We assume that the subsystem contains no sub-blocks. The subsystem's output is directly linked to its input.

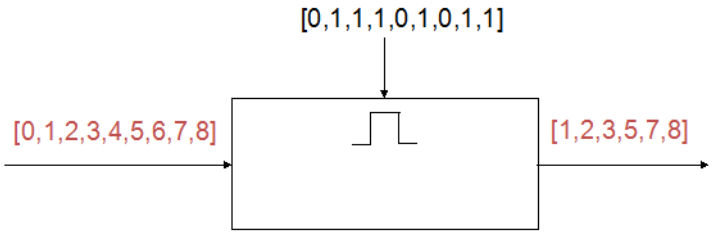


Figure 4.10: Timing Mechanism in SIMULINK (Enabled Sub-System)

4.5.2 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. Similar to the synchronous languages, SIGNAL also assumes the synchrony hypothesis (See 2.3). 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 b_x, called the clock of x. If x is present at instant i, b_x(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*[**?**]. We refer the reader to [**?**] for more detailed discussion on SIGNAL timing model.

4.5.3 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 b_i 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[10] (See Formulas 4.4/4.5). The clock inference Algorithm is shown below:

$$(\pi_B, \theta_B) = GCD_{rule}((\pi_i, \theta_i)_{i=1\ldots n}) \tag{4.4}$$

where:

$$\pi_B = \begin{cases} \gcd(\pi_1, \ldots, \pi_n) & \text{if } \theta_1 = \cdots = \theta_n \\ \gcd(\pi_1, \ldots, \pi_n, \theta_1, \ldots, \theta_n) & otherwise \end{cases} \tag{4.5}$$

$$\theta_B = \begin{cases} \theta_1 mod \pi & \text{if } \theta_1 = \cdots = \theta_n \\ 0 & otherwise \end{cases} \tag{4.6}$$

For Example:
GCD_{rule}((4,0),(3,0)) = (1,0)
GCD_{rule}((12,4),(12,0)) = (4,0)
GCD_{rule}((12,4),(4,4)) = (4,4)
GCD_{rule}((12,4),(12,3)) = (1,0)

Figure 4.11 and Figure 4.12 illustrate timing errors, because of which the SIMULINK model is rejected. In Figure 4.11, the triggered Subsystem has three inputs with different periods. In Figure 4.12, the adder has a sample time equal to 5, which is neither a multiplier nor a divisor of the input periods 4 and 2. Figure 4.13 shows an example of clock inference. The sample periods of the Adder and Multiplier are initially undefined. First, the Adder period is set to the greatest common divisor of the periods of the Constant and Unit Delay blocks, namely 2. Similarly, the sample time of the Multiplier is inferred to be 1.

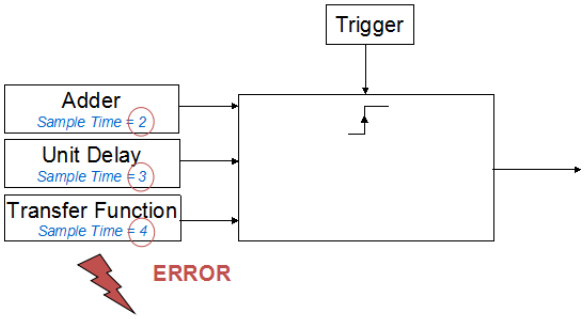


Figure 4.11: Example 1: Clock Inference Timing Error

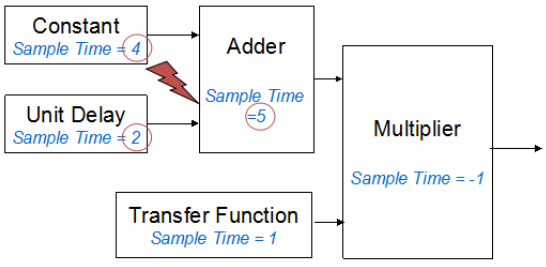


Figure 4.12: Example 2: Clock Inference Timing Error

Algorithm 3: Clock Inference Algorithm

Data: Input: SIMULINK Model M
% *Set the periods of the blocks enclosed in a Triggered or Enabled Subsystem*
forall the *(blocks b_i in M)* **do**
  if *((b_i.ParentBlockType=Enabled SubSystem)* ***OR*** *(b_i.ParentBlockType=Triggered SubSystem))* **then**
    b_i.Period=ParentBlockPeriod
    b_i.Phase=ParentBlockPhase
    b_i.IsFixedPeriod=true
  else
    b_i.IsFixedPeriod=false
  end
  Periodes_1[i]=b_i.Period
  Phases_1[i]=b_i.Phase
end
% *Set the number of iterations to 0*
n=0;
% *Iterate until a fixed point is reached*
while *(n==0)* ***OR****(Periodes_n!=Periodes_{n+1})* **do**
  if *(n > 0)* **then**
    n++
  end
  forall the *(blocks b_i in M)* **do**
    if *(b_i.IsFixedPeriod==false)* **then**
      % *Clock Inference*
      if *(Periodes_n[i]==-1)* **then**
        (Periodes_{n+1}[i],Phases_{n+1}[i])=GCD_{Rule}(InputPeriodes[i],InputPhases[i])
      else
        % *Detect Timing Errors*
        if *(!IsMultOrDiv(Periodes_n[j],Phases_n[j],InputPeriodes[i],InputPhases[i])* **then**
          return TIMING ERROR
        end
        if *((b_i.Type=Enabled Subsystem)* ***OR*** *(b_i.Type=Triggered Subsystem)* ***AND*** *AreDifferent(InputPeriodes[i]))* **then**
          return TIMING ERROR
        end
      end
    end
  end
end

Apart from SIGNAL code synthesis, our tool can be used for checking typing and timing

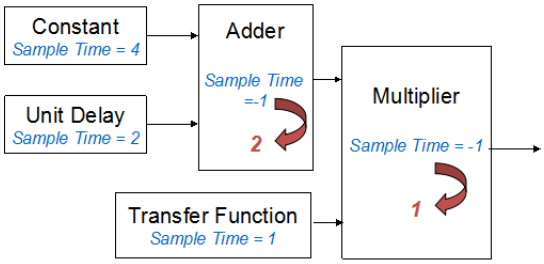


Figure 4.13: Example 3: Clock Inference

errors in SIMULINK. Hence if the model is rejected in SIMULINK, our tool will reject it as well.

4.6 Clock Translation

Once all the sample times are inferred from the previous step, we use these information to reproduce the same traces of SIMULINK. 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 connected blocks A and B, we distinguish three cases:

Figure 4.14: Clock Translation

- **Case 1**: $\alpha = \frac{Periode_A}{Periode_B} = 1$
  In this case, no extra timing constraints should be taken into account. The output signal clock of A is equal to the output signal clock of B.

$$\text{output}_A \hat{} = \text{input}_B \hat{} = \text{output}_B$$

- **Case 2**: $\alpha = \frac{Periode_A}{Periode_B} > 1$
  The clock of the output signal of A is faster than the one of B. In this case, **Under-sampling** should be performed in B. The clock relation between the input and output signals of B is:

$$\hat{} \text{output}_B = \alpha \hat{} \text{input}_B + \beta$$

β here is the phase difference between output$_B$ and input$_B$: β = 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 *output*$_A$. 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 how undersampling is performed, in case of a Unit Delay block:

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

The flow of the output signal *output*$_B$ for the Unit Delay block, with α=2 and β=0 and initial value v_0, is shown below. The obtained flow is equivalent to the one produced by SIMULINK, when the sample time of B is double the one of A.

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**: $\frac{1}{\alpha} = \frac{Periode_A}{Periode_B} < 1$
The clock of the output signal of B is faster than the one of A. In this case, **Oversampling** should be performed in B.
Similarly to the Undersampling case, the clock relation between the input and output signals of B is:

$$\hat{}\ \text{output}_B = \alpha\ \hat{}\ \text{input}_B + \beta$$

β here is the phase difference between output$_B$ and input$_B$: $\beta = Phase_B\text{-}Phase_A$
In order to implement the discussed affine clock relation, we consider a counter variable *cnt*, that has the same clock as the output signal *output*$_B$. 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 how oversampling is performed, 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 ^$output_B$;

The flow of the output signal $Output_B$ for the Unit Delay block, with α=2 and β=0 is shown below. The obtained flow is equivalent to the one produced from SIMULINK, where the sample time of A is double the one of B.

$Output_A$	v_1	·	v_2	·	v_3	·	v_4	·	v_5	·	v_6
Cnt	0	1	2	3	4	5	6	7	8	9	10
Cnt2	0	1	0	1	0	1	0	1	0	1	0
$s$tmp	v_0	·	v_1	·	v_2	·	v_3	·	v_4	·	v_5
$Output_B$	v_0	v_0	v_1	v_1	v_2	v_2	v_3	v_3	v_4	v_4	v_5

4.7 Basic SIMULINK Blocks translation

- **Sum Block**
  The 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

- **Product Block**
  The Product Block multiplies two inputs with each other.

  | out := inp_1 * inp_2

- **Difference Block**
  The Difference Block subtracts the old output from the new input.

  | tmp := inp $ init INITVAL
  | out := inp - tmp

- **Gain Block**
  The Gain Block performs a multiplication of the input with a constant.

| out := inp * GAIN

- **Logical Operator Block**
  The Logical Operator block performs a boolean operation Op ∈ {AND,OR,NOT} on its inputs.

  | out := inp_1 **Op** Inp_2

- **Relational Operator Block**
  The Relational Operator block performs a logical operation Op ∈ {<,>,=,≥,≤} on its inputs.

  | out := inp_1 **Op** inp_2

- **Unit Delay** and **Integer Delay Block**
  The output of the Unit Delay/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**
  The data type conversion is translated into a Type Casting operation. For example, 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 4.6 is performed.

- **Constant Block**
  The Constant Block value is added in SIGNAL to the list of the parent process parameters.

- **Saturation Block**

  The Saturation Block truncates its inputs according to an upper limit (UP_LIM) and a lower limit (LOW_LIM) bounds given by the user.

| out := (UP_LIM when (inp > UP_LIM)) default (LOW_LIM when (inp < LOW_LIM))) default input

- **Switch Block**

  The Switch Block has three inputs. It compares its middle input textitinp_2 to a THRES value. If it is greater than the THRESHOLD, the first input is passed to the output, otherwise the third input is emitted as an output.

  | out := (inp_1 when (inp_2 > THRES)) default (inp_3 when (inp_2 < THRES))

- **Pulse Generator**
  The Pulse Generator with a period= 5, a phase= 2 and a pulse width=2 is translated into the following SIGNAL code:

```
process PulseGeneratorL1=
  { real AMPLITUDE}
  (? !real out;)
  (| dpg1:= dpg2 $1 init AMPLITUDE
  | dpg2:= dpg3 $1 init AMPLITUDE
  | dpg3:= dpg4 $1 initreal(0)
  | dpg4:= dpg5 $1 initreal(0)
  | dpg5:= dpg1 $1 initreal(0)
  | pha1 := dpg1 $1 initreal(0)
  | pha2 := pha1 $1 initreal(0)
  | out := pha2
  |)
where
  real dpg1,dpg2,dpg3,dpg4,dpg5,dpg6,dpg7,dpg8,dpg9,dpg10,pha0;
end;
```

- **Discrete Filter** and **Discrete Transfer Function**
  The Discrete Filter/Discrete Transfer Function considered parameters during the translation are the nominator coefficients number NB_COEFF_N and values VAL_COEFF_N, the denominator coefficients number NB_COEFF_D and values VAL_COEFF_D and the initial state value INIT_VAL. The transfer function $\frac{1}{1+0.5z^{-1}}$ is translated into the following SIGNAL code:

```
Process Filter=
  { integer NB_COEFF_N;
  integer NB_COEFF_D;
  [ NB_COEFF_D ] real INIT_VAL;
  [ NB_COEFF_N ] real VAL_COEFF_N;
```

```
   [ NB_COEFF_D] real VAL_COEFF_D;}
   (? real input;
   ! real output;
   )
   (| output := (input * VAL_COEFF_N[0]+tmp0)/VAL_COEFF_D[0]
   | tmp0:=(- VAL_COEFF_D[1] * output )$1 init INIT_VAL
   |)
where
   real tmp0;
end;
```

- **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, with respectively N1, N2 and N3 elements is:

```
Process Mux=
   {integer N1;
   integer N2;
   integer N3;}
   (? [N1] integer inp1;
   [N2] integer inp2;
   [N3] integer inp3;
   ! [N1+N2+N3] integer out;)
   (|out := [inp1,inp2,inp3]
   |)
where
end;
```

- **Combinatorial Logic**
  The Combinatorial Logic block 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;
   integer M;
   integer 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]
```

- **From Workspace**
  This block is translated into an input of the SIGNAL process.

- **To Workspace**
  This block is translated into an output of the SIGNAL process.

- **Trigger**
  The Trigger block 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 better the trigger mechanism:

Input	-1	0	1	2	-2	3	1	4	-1	5	6
Rising Trigger	f	t	f	f	f	t	f	f	f	t	f
Falling Trigger	f	f	f	f	t	f	f	f	t	f	f
Either Trigger	f	t	f	f	t	t	f	f	t	t	f

The following SIGNAL code generates a Rising Trigger flow. The not_before variable ensures that no trigger is only produced, if no one happened in the previous time step.

| RiseTrigger_{old} := Trig $ init false
| Trig := neg_to_nonneg **OR** (nonpos_to_pos and not_before)
| neg_to_nonneg := (($\text{inp}_{old} < 0$) **AND** (inp ≥ 0))
| nonpos_to_pos := (($\text{inp}_{old} \leq 0$) **AND** (inp > 0))
| not_before := **NOT** (RiseTrigger_{old})

The Falling Trigger is defined in a similar way:

| FallTrigger_{old} := Trig $ init false
| Trig := nonneg_to_neg **OR** (pos_to_nonpos **AND** not_before)
| nonneg_to_neg := (($\text{inp}_{old} \leq 0$)**AND**(inp < 0))

| pos_to_nonpos := (($inp_{old} > 0$)**AND**($inp \leq 0$))
| not_before := **NOT**($FallTrigger_{old}$)

- **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 ($inp \leq 0$)))

4.8 SubSystems translation

As the model increases in size, the 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.

4.8.1 Plain SubSystems Translation

Every block in SIMULINK is translated into a SIGNAL process. In the case of subsystems, their enclosed blocks are translated into subprocesses. The subprocesses are declared in the "where" part, as illustrated in the example below:

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

A SIMULINK diagram can be constructed in SIGNAL by recursively translating subsystems into processes and the enclosed atomic blocks into subprocesses. The model parameters are read from the data structure obtained from the XML file and saved in the SIGNAL parameter file *Sim2Sig.PAR*. The SIGNAL code is generated in *Sim2Sig.SIG*. The top down translation algorithm is presented in Alg 4. First the parameter list for all the blocks is generated, then the outputs and inputs are defined. After that, the subprocesses in the first hierarchy level are called. Additional equations equation for the block connections are generated. In the *Where* part, the intermediate variables, that are neither input nor output signals, are defined. The subprocesses body is also implemented. In case one of the subprocesses is a SubSystem, the described process is recursively repeated. For each block, type and clock translation are performed as described is Section 4.4 and Section 4.6. Examples of SIMULINK model translation are presented in Section 5.

4.8.2 Triggered SubSystems Translation

The Triggered SubSystem is a SubSystem with a control input, namely the *trigger* input. The Subsystem executes, each time a trigger event occurs.If no trigger happens, the output is either reset or it holds its old value. Figure 4.15 shows a Triggered SubSystem enclosing a Unit Delay block. Below is the corresponding SIGNAL code. The parent Process *Sim2Sig*

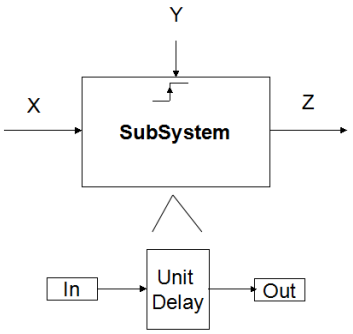


Figure 4.15: Triggered SubSystems Translation

calls the subprocesses *Trigger* and *SubSys*. *Trigger* generates the rising trigger $SubSys_{trig}$ from the input 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 comes.

```
Process Sim2Sig =
    {integer NB_DELAY;
    integer INIT_VALUE;
    }
    (? integer X;
    integer Y;
```

```
    ! integer Z;)
    (| SubSys_trig := Trigger(Y)
    | Z := SubSys{NB_DELAY,INIT_VALUE}(X when SubSys_trig) cell ^X |)
Where
    boolean SubSys_trig;
    process Trigger=
        ( ? real inp;
        ! boolean RiseTrigger;)
        (| RiseTrigger_old := 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)|)
   End;
    Process SubSys=
        {integer NB_DELAY;
        integer INIT_VALUE;
        }
        (? integer in;
        ! integer out;)
        (| UnitDelay_out := UnitDelay{NB_DELAY,INIT_VALUE}(UnitDelay_in)
        | UnitDelay_in := in
        | out := UnitDelay_out |)
    Where
        integer UnitDelay_in;
        integer UnitDelay_out;
        process UnitDelay=
            {integer NB_DELAY;
            integer INIT_VALUE;
            }
            (? real input;
            ! real output;)
            (| output := input $ NB_DELAY init INIT_VALUE
            |)
        End;
    End;
```

4.8.3 Enabled SubSystems Translation

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

Algorithm 4: Hierarchical Translation of SIMULINK into SIGNAL

Data: Input: SIMULINK Model M
% *write the Process Name in Sim2Sig.SIG*
WriteName(Sim2Sig.SIG,ProcessName)
% *write the Process Parameters in Sim2Sig.SIG and their values in Sim2Sig.SIG*
forall the *(blocks b_i in M)* **do**
  WriteParam$_{.SIG}$(Sim2Sig.SIG,b_i.Parameters)
  WriteParam$_{.PAR}$(Sim2Sig.SIG,b_i.Parameters.Values)
end
% *start with Level 1*
Level=0
Recursiv Call:
Level++;
% *write the input list in Level$_i$ in Sim2Sig.SIG (the inputs are the signals connected to the From Workspace or In Blocks)*
WriteInputs(Sim2Sig.SIG,Level$_i$,b_i.FromWorkspace)
% *write the output list Level$_i$ in Sim2Sig.SIG (the outputs are the signals connected to the To Workspace or Out Blocks)*
WriteOutputs(Sim2Sig.SIG,Level$_i$,b_i.ToWorkspace)
% *Call the SubProcesses in Level$_i$*
forall the *(For all blocks b_j in Level$_i$)* **do**
  b_j.CallProcess()
end
% *Create Block Connections in Level$_i$*
forall the *(blocks b_i and b_j)* **do**
  if *(b_i is connected to b_j)* **then**
    b_i.output = b_j.input
  end
end
% *write the "Where" part of the process*
% % *define the internal signals of the process in Level$_i$*
write(Sim2Sig.SIG,InternalSignals)
forall the *(For all blocks b_j in Level$_i$)* **do**
  b_j.ProcessBody()
  if *(b_j.Type==Subsystem)* **then**
    Goto Recursiv Call
  end
end

Chapter 5

Case Studies

In this section, we present three case studies. We use our tool to translate into SIGNAL three SIMULINK models with increasing complexity. The first model, is a system composed of a plant and a controller in the loop. The second example, is a discretized DC Motor. In the last case study, we translate a system consisting of an input sampler and the discretized Motor from the second case study in a loop with a PID controller.

5.1 Closed Loop Controller

In this case study, the goal is to better illustrate the translation flow on a simple system composed of a plant in a loop with a controller. The naming convention for the signals is as follows (the + operator denotes string concatenation):

- **Input**: Block Name + L + Level Number + _in + Input Port Number
  For example, the first input of an Adder block is: *AddL1_in1*
- **Output**: Block Name + L + Level Number + _out + Output Port Number
  For example, the output of an Adder block is: *AddL1_out1*

Similar to the signal names, each parameter name is preceded by the corresponding block name and its level. Below is a snippet of the generated SIGNAL code. The full version is attached in Index A.

```
Process Sim2Sig =
% Parameters
    {real GainL1_GAIN;
    integer UnitDelayL1_NB_DELAY;
    [UnitDelayL1_NB_DELAY] real UnitDelayL1_INIT_VALUE;
```

```
    real controllerL1_INIT_VAL;
    integer plantL1_NB_COEFF_N;
    integer plantL1_NB_COEFF_D;
    [plantL1_NB_COEFF_D] real plantL1_INIT_VAL;
    [plantL1_NB_COEFF_N] real plantL1_VAL_COEFF_N;
    [plantL1_NB_COEFF_D] real plantL1_VAL_COEFF_D;
    integer BUF}
% Inputs/Outputs
    (? real AddL1_in1;
    ! real plantL1_out1;)
% Process Body
    (| AddL1_out1 := AddL1(AddL1_in1,AddL1_in2)
    | GainL1_out1 := GainL1{GainL1_GAIN}(GainL1_in1)
    | UnitDelayL1_out1 :=UnitDelayL1{UnitDelayL1_NB_DELAY,UnitDelayL1_INIT_VALUE}
    (UnitDelayL1_in1)
    | controllerL1_out1 :=controllerL1{controllerL1_INIT_VAL}(controllerL1_in1)
    | plantL1_out1 := plantL1{plantL1_NB_COEFF_N,plantL1_NB_COEFF_D,plantL1_INIT_VAL,
    plantL1_VAL_COEFF_N,plantL1_VAL_COEFF_D}(plantL1_in1)
    | GainL1_in1 := AddL1_out1
    | controllerL1_in1 := GainL1_out1
    | AddL1_in2 := UnitDelayL1_out1
    | plantL1_in1 := controllerL1_out1
    | UnitDelayL1_in1 := plantL1_out1
    |)
```

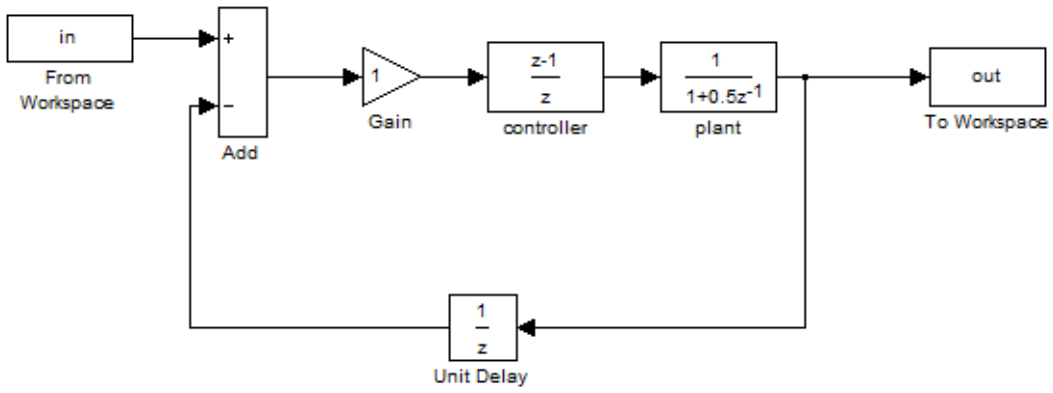


Figure 5.1: Case Study 1: Closed Loop Controller

5.2 Discretized DC-Motor

Figure 5.1 shows a three level discretized DC-Motor. The electrical and mechanical dynamics of the model are represented by the following equations:

$$V_{in} - R \cdot i - K_e \cdot \theta[n+1] = L \cdot i[n+1] \tag{5.1}$$

$$K_t \cdot i - b \cdot \theta[n+1] = J \cdot \theta[n+2] \tag{5.2}$$

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 5.1 is implemented in the SubSystem S1. Equation 5.2 is implemented in the SubSystem S2. In oder to better illustrate how the hierarchical translation is performed, a simplified version of the generated SIGNAL code is presented below (The full version is attached in appendix B). We skip the body of the basic processes, the listing of the signal connections and the intermediate signal definitions.

```
process Sim2Sig =
  {Parameters}
  (? real DCMotorL1_in1;
  ! real DCMotorL1_out1;)
  (|DCMotorL1_out1:=DCMotorL1{Parameters}(DCMotorL1_in1)|)
where
   Process DCMotorL1=
     {% Parameters}
     (? real in1;
     !real out1;)
     |Integrator3L2_out1:=Integrator3L2{Parameters}(Integrator3L2_in1)
     |KeL2_out1 := KeL2{Parameters}(KeL2_in1)
     |KtL2_out1 := KtL2{Parameters}(KtL2_in1)
     |S1L2_out1:=S1L2{Parameters}(S1L2_in1,S1L2_in2)
     |S2L2_out1:=S2L2{Parameters}(S2L2_in1)
      % Create Signal connections |)
   where
      % Define Intermediate Signals
      Process Integrator3L2= ···
      Process KeL2= ···
      Process KtL2= ···
      Process S1L2=
         {Parameters}
         (? real in1;
         real in2;
         ! real out1;)
        (|AddL3_out1 := AddL3(AddL3_in1,AddL3_in2,AddL3_in3)
        |InductanceL3_out1 := InductanceL3{Parameters}(InductanceL3_in1)
```

```
      |Integrator1L3_out1 :=Integrator1L3{Parameters}(Integrator1L3_in1)
      |resistanceL3_out1 := resistanceL3{Parameters}(resistanceL3_in1)
      % Create Signal connections
      |)
    where
      % Define Intermediate Signals
      Process AddL3= ···
      Process InductanceL3= ···
      Process Integrator1L3= ···
      Process ResistanceL3= ···
    end;
    Process S2L2=
      {Parameters}
      (? real in1;
      !real out1;)
      (|Add1L3_out1 := Add1L3(Add1L3_in1,Add1L3_in2)
      |InertiaL3_out1 := InertiaL3{Parameter}(InertiaL3_in1)
      |Integrator2L3_out1 :=Integrator2L3{Parameter}(Integrator2L3_in1)
      |resistance1L3_out1 := resistance1L3{Parameter}(resistance1L3_in1)
      % Create Signal connections
    where
      % Define Intermediate Signals
      Process Add1L3= ···
      Process InertiaL3= ···
      Process Integrator2L3= ···
      Process resistance1L3= ···
    end;
  end
end
```

5.3 Discretized DC-Motor Closed Loop Controller

In this case study, we generate a system composed of the DC Motor from the second case study, in a loop with a PID controller. The system inputs are sampled by the *Input sampling* block. The goal of this case study is to show how the different SIMULINK Timing mechanisms are translated into SIGNAL. 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. Below is a simplified version of the code generated by our tool. We skip the body of the basic processes, the body of the DC Motor SubSystem as it s presented in the previous case study, the listing of the signal connections and the intermediate signals definitions. The full code is attached in index C.

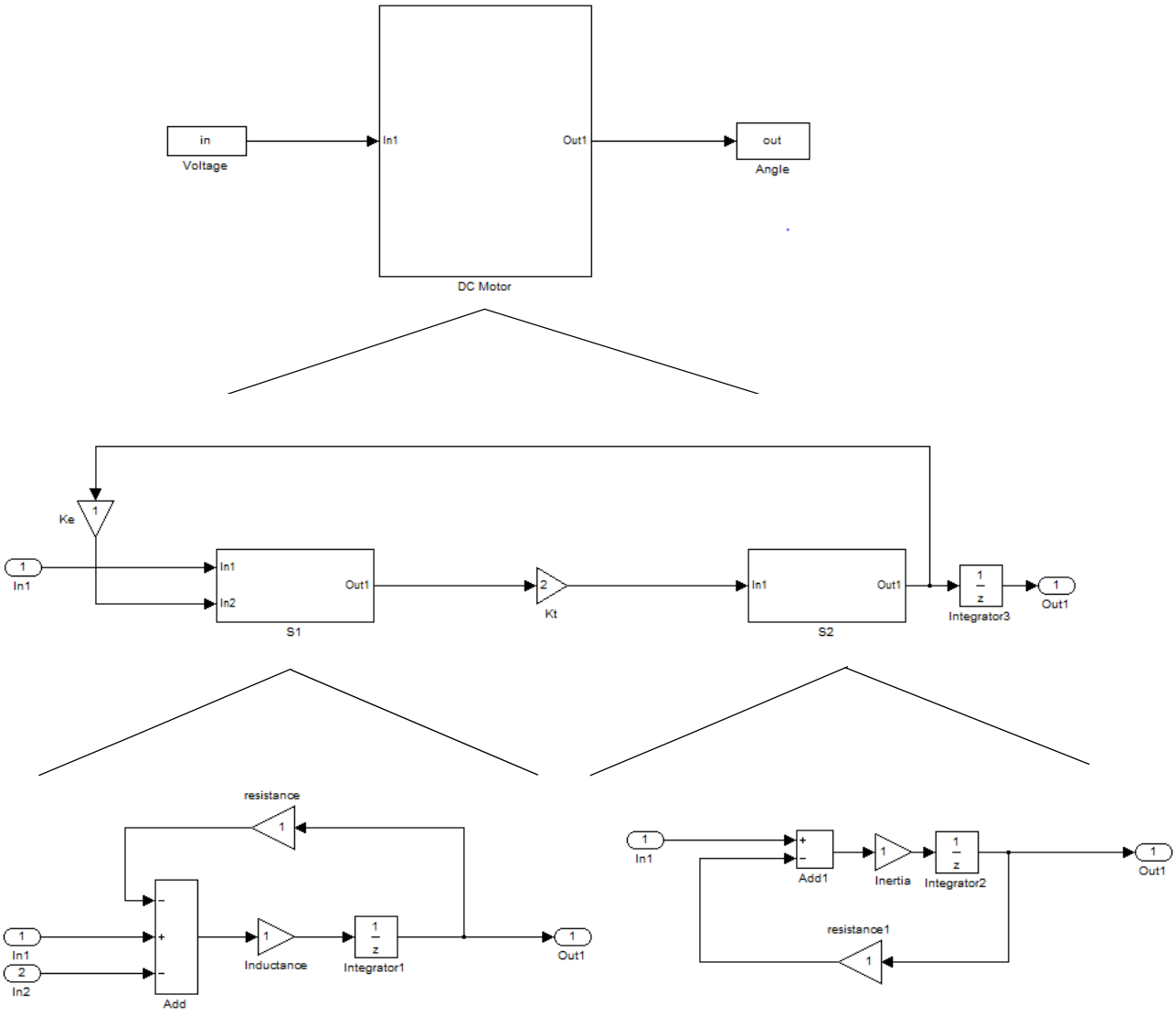


Figure 5.2: Case Study 2: Discretized DC-Motor

```
process Sim2Sig =
   {Parameters }     (? real InputSamplingL1_in1;
   ! real DCMotorL1_out1;)
   (|AddL1_out1 := AddL1(AddL1_in1,AddL1_in2)
   |DCMotorL1_out1:=DCMotorL1{Parameters}(DCMotorL1_in1)
    |InputSamplingL1_out1:=InputSamplingL1{Parameters}(InputSamplingL1_in1
   when InputSamplingL1_en) cell ^InputSamplingL1_in1
```

```
  |InputSamplingL1_en := EnableL2(InputSamplingL1_in_en)
  |PIDL1_out1:=PIDL1{Parameter}(PIDL1_in1)
  |PulseGeneratorL1_out1:=PulseGeneratorL1{Parameter}()
  |RateAdjustmentL1_out1 := RateAdjustmentL1{Parameter}(RateAdjustmentL1_in1)
  % Signal Connections
  |)
where
   % Intermediate Signals Definition
   Process AddL1= ···
   Process PulseGeneratorL1= ···
   Process RateAdjustmentL1=···
   Process DCMotorL1= ···
   Process InputSamplingL1=
      {Parameters }
       (? in;
       ! out;)
       (| UnitDelayL2_out1 :=UnitDelayL2{Parameters}(UnitDelayL2_in1)
       % Signal Connections |)
   where
      % Intermediate Signals Definition
      Process UnitDelayL2= ···
      Process EnableL2= ···
   end
   Process process PIDL1=
     {Parameters}
     (? real in1;
     ! real out1)
     (| AddL2_out1 := AddL2{Parameters}(AddL2_in1,AddL2_in2,AddL2_in3)
     |Add1L2_out1 := Add1L2(Add1L2_in1,Add1L2_in2)
     |DGainL2_out1 := DGainL2{Parameters}(DGainL2_in1)
     |IGainL2_out1 := IGainL2{Parameters}(IGainL2_in1)
     |IGain1L2_out1 := IGain1L2{Parameters}(IGain1L2_in1)
     |PGainL2_out1 := PGainL2{Parameters}(PGainL2_in1)
     |UnitDelay1L2_out1 :=UnitDelay1L2{Parameters}(UnitDelay1L2_in1)
     |UnitDelay2L2_out1 :=UnitDelay2L2{Parameters}(UnitDelay2L2_in1)
     % Define Signal Connections
  where
     % Define Intermediate Signals
      process AddL2= ···
      process Add1L2= ···
      process DGainL2= ···
      process IGainL2= ···
```

process IGain1L2= $\cdots$
process PGainL2= $\cdots$
process UnitDelay1L2= $\cdots$
process UnitDelay2L2= $\cdots$
end
end

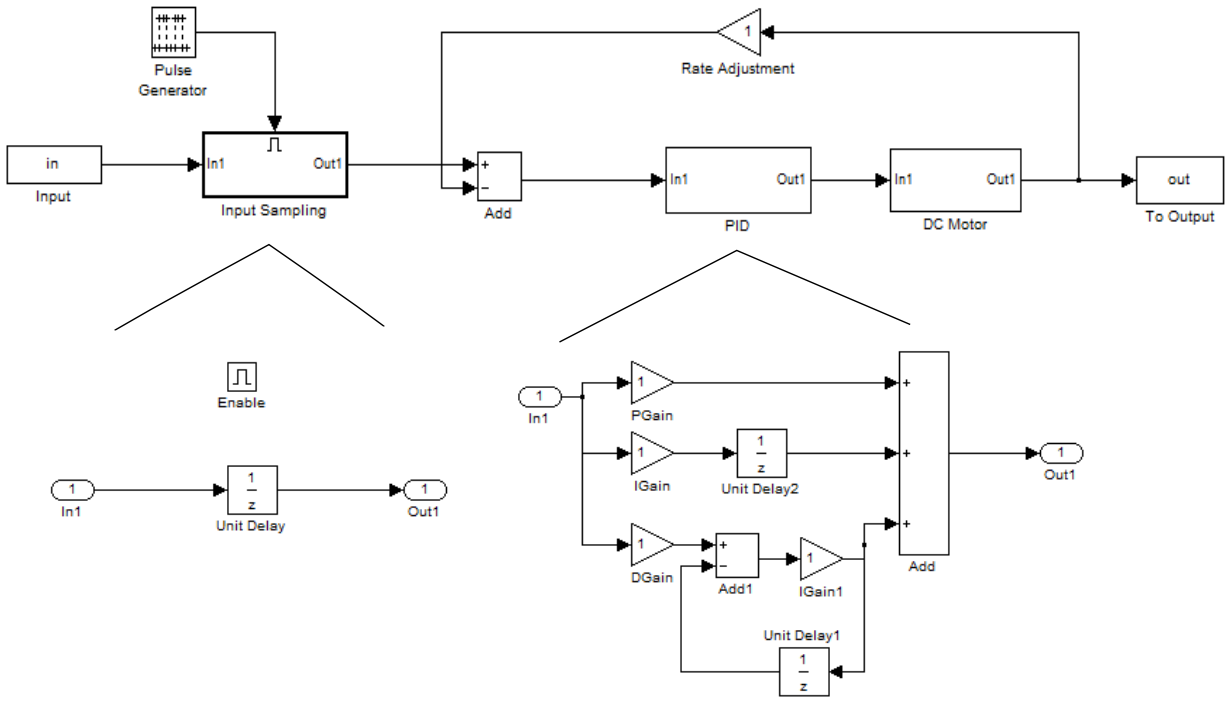


Figure 5.3: Case Study 3: Discretized DC-Motor Closed Loop Controller

The flow equivalence is validated by comparing the traces generated from the SIMULINK model and its corresponding SIGNAL translation.

Chapter 6

Conclusion

6.1 Summary

In this thesis, we developed a tool that translates a discrete time subset of SIMULINK into the polychronous formal language SIGNAL. The motivation behind this work, lies in the fact that SIMULINK, despite of being the most popular tool in embedded software design, is, however, not based on formal semantics (Section 2.2). 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, but they have clear and precise semantics, that allow the application of powerful design methodologies, like Model Based Design (Section 2.1). In Section 3, previous attempts of giving SIMULINK Formal semantics or translating it into a formal model of computation like formal languages or hybrid automata are presented. The choice of SIGNAl as a target language for this work, is justified by its multi-rate nature that allows for signal streams to be computed asynchronously to one another which fits very easily to a multi-threaded environment. Besides, the translation will allows engineers to gain access to the SIGNAL formal toolset for code generation and design verification, which will minimize the testing costs and time. The translation follows three major steps: type inference, clock inference and hierarchical top-down translation (Section 4.2). In the type inference step (Section 2), the type of each signal in SIMULINK is inferred and translated into its corresponding type in SIGNAL. The clock inference's (Section 4.5) goal is deriving the phase and period of every SIMULINK block. From these information affine clock relations between each block's input and output clock are constructed. This bridges the gap between synchronous and polychronous models of computation and allows the generation of flow equivalent SIGNAL programs from the SIMULINK models (Section 4.6). After the type and clock inference steps, 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 (Section 4.8). The tool is applied to three case studies with increasing complexity, in order to better illustrate the translation flow (Section 5).

6.2 Conclusion

Translating SIMULINK (almost-synchronous) to SIGNAL (polychronous) is challenging because of the different timing and typing semantics. The flow equivalence between SIMULINK models and their corresponding SIGNAl programs was achieved by inferring the type of each SIMULINK signal, mapping it to its corresponding SIGNAL type and generating affine clock relations between each block inputs and outputs. These clock relations are then implemented during the translation. 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.

6.3 Limitations

The main drawback of this tool is its dependency on SIMULINK semantics, which keeps changing from one version to another. Even for this version, we assumed some parameters to be set to fixed options (For example, fixed step simulation and not variable step simulation). 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. This is the case of the Enabled SubSystem. SIMULINK does not impose the period of the blocks in an Enabled SubSystem to be inherited, as in the case of Triggered SubSystems. The enclosed blocks can have different sampling times. This results in a complicated behavior of the Enabled SubSystem. Figure 6.1 shows an Enabled Subsystem that performs a substraction when triggered. The enable signal is created using a Pulse Generator with an amplitude equal to 1, a pulse width equal to 3 and a sampling time equal to 1 (see Figure 6.2a). The sample time of the enclosed Unit Delay block is set to 2. We expect the output signal y to be either equal to the sample time of the enable signal e or to the sample time of the Adder block, which is the greatest common divisor of the period of the Unit Delay block and the one of the input of the Enabled Subsystem. Hence using the GCD_{rule}, the Adder Block has a period equal to 1. In both cases, the sample time of y is expected to be equal to 1. However, as it is shown in Figure 6.2b, every time a trigger is received, it take two sample times to obtain a new output y. A clear timing behavior in this case can not be captured and modeled. In order to be able to translate the Enabled SubSystem block, we had to assume that its semantic is similar to the one of the triggered subsystem, namely, the period of its enclosed blocks are all equal to the one of the enable signal.

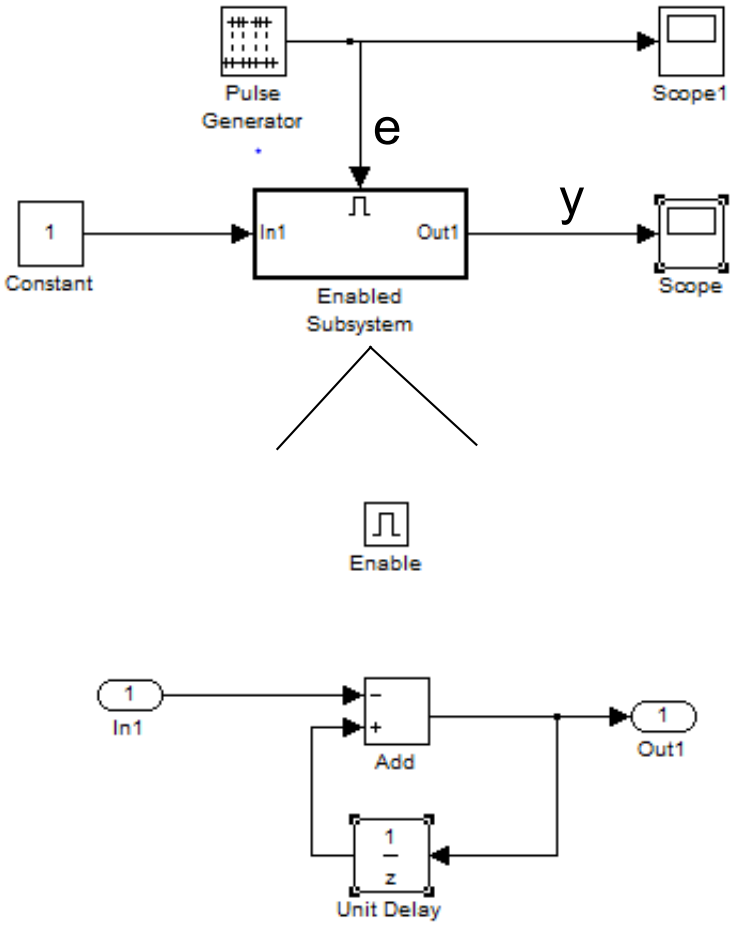
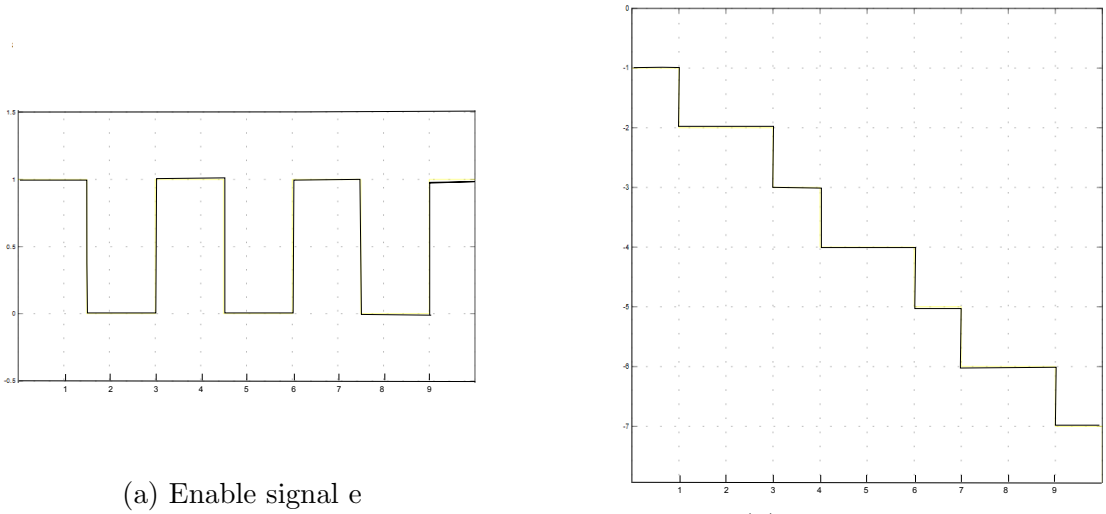


Figure 6.1: Example of an Enabled SubSystem

6.4 Future Work

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.
Besides, 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. Finally, flow equivalence between the SIGNAL program and the SIMULINK model can be proven, by formalizing the semantics of the SIMULINK blocks as operational semantic rules,



(a) Enable signal e

(b) Output signal y

Figure 6.2: Ambiguous behavior of the enabled SubSystem

then defining a notion of behavior of the SIMULINK model in terms of traces. After that, it should be shown, that the such semantic rules, uniquely determine the behavior of the SIMULINK model. It remains to prove that the behavior computed by the SIMULINK model matches the behavior of the SIGNAL program that we claim to be equivalent to the original SIMULINK model.

Bibliography

[1] Mbdv. URL `http://www.engineering.com/DesignSoftware`.

[2] Edward A Lee and Alberto Sangiovanni-Vincentelli. The tagged signal model-a preliminary version of a denotational framework for comparing models of computation.

[3] Gérard Berry and Laurent Cosserat. The esterel synchronous programming language and its mathematical semantics. In *Seminar on Concurrency, Carnegie-Mellon University*, pages 389–448, London, UK, UK, 1985. Springer-Verlag. ISBN 3-540-15670-4. URL `http://dl.acm.org/citation.cfm?id=646723.702721`.

[4] 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. ISSN 0018-9219. doi: 10.1109/5.97300.

[5] metropolis. URL `http://embedded.eecs.berkeley.edu/metropolis/metamodel.html`.

[6] *Simulink: User's Guide*. The Mathworks.

[7] Olivier Bouissou and Alexandre Chapoutot. An operational semantics for simulink's simulation engine. *ACM SIGPLAN Notices*, 47(5):129–138, 2012.

[8] 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.

[9] 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.

[10] 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.

[11] Abdoulaye Gamati. *Designing Embedded Systems with the SIGNAL Programming Language: Synchronous, Reactive Specification*. Springer Publishing Company, Incorporated, 1st edition, 2009. ISBN 1441909400, 9781441909404.

[12] Irisa. URL `http://www.irisa.fr/`.

[13] Signali. URL `http://www.irisa.fr/vertecs/Logiciels/sigali.html`.

[14] Sme. URL `http://www.irisa.fr/espresso/Polychrony/index.php`.

[15] Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[16] Todor Stefanov, Claudiu Zissulescu, Alexandru Turjan, Bart Kienhuis, and Ed Deprette. System design using khan process networks: the compaan/laura approach. In *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings*, volume 1, pages 340–345. IEEE, 2004.

[17] 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.

[18] Bijoy Antony Jose. Formal model driven software synthesis for embedded systems, 2011.

[19] Julien Ouy, Jean-Pierre Talpin, Loïc Besnard, and Paul Le Guernic. Separate compilation of polychronous specifications. *Electronic Notes in Theoretical Computer Science*, 200(1):51–70, 2008.

[20] N. Halbwachs, F. Lagnier, and C. Ratel. Programming and verifying real-time systems by means of the synchronous data-flow programming language lustre. *IEEE Transactions on Software Engineering, Special Issue on the Specification and Analysis of Real-Time Systems*, September 1992.

[21] Pascal Raymond, Xavier Nicollin, Nicolas Halbwachs, and Daniel Weber. Automatic testing of reactive systems. In *Real-Time Systems Symposium, 1998. Proceedings., The 19th IEEE*, pages 200–209. IEEE, 1998.

[22] Parosh Aziz Abdulla, Johann Deneux, Gunnar Stålmarck, Herman Ågren, and Ove Åkerlund. Designing safe, reliable systems using scade. In *Leveraging Applications of Formal Methods*, pages 115–129. Springer, 2006.

[23] Marius Dorel Bozga, Vassiliki Sfyrla, and Joseph Sifakis. Modeling synchronous systems in bip. In *Proceedings of the seventh ACM international conference on Embedded software*, pages 77–86. ACM, 2009.

[24] Thomas A Henzinger. *The theory of hybrid automata*. Springer, 2000.

Appendix A

Case Study 1

```
process Sim2Sig =
{real GainL1_GAIN;
integer UnitDelayL1_NB_DELAY;
[UnitDelayL1_NB_DELAY] real UnitDelayL1_INIT_VALUE;
real controllerL1_INIT_VAL;
integer plantL1_NB_COEFF_N;
integer plantL1_NB_COEFF_D;
[plantL1_NB_COEFF_D] real plantL1_INIT_VAL;
[plantL1_NB_COEFF_N] real plantL1_VAL_COEFF_N;
[plantL1_NB_COEFF_D] real plantL1_VAL_COEFF_D;
integer BUF}
(? real AddL1_in1;
! real plantL1_out1;
)
(|AddL1_out1 := AddL1(AddL1_in1,AddL1_in2)
|GainL1_out1 := GainL1GainL1_GAIN(GainL1_in1)
|UnitDelayL1_out1 :=UnitDelayL1UnitDelayL1_NB_DELAY,UnitDelayL1_INIT_VALUE
(UnitDelayL1_in1)
|controllerL1_out1 :=controllerL1controllerL1_INIT_VAL(controllerL1_in1)
|plantL1_out1 := plantL1plantL1_NB_COEFF_N,plantL1_NB_COEFF_D,plantL1_INIT_VAL,
plantL1_VAL_COEFF_N,plantL1_VAL_COEFF_D(plantL1_in1)
|GainL1_in1 := AddL1_out1
|controllerL1_in1 := GainL1_out1
|AddL1_in2 := UnitDelayL1_out1
|plantL1_in1 := controllerL1_out1
|UnitDelayL1_in1 := plantL1_out1
|)
where
```

```
real GainL1_in1;
real AddL1_out1;
real controllerL1_in1;
real GainL1_out1;
real AddL1_in2;
real UnitDelayL1_out1;
real plantL1_in1;
real controllerL1_out1;
real UnitDelayL1_in1;
process AddL1=
(? real input0;
real input1;
! real output;
)
(|tmp:=+input0 - input1
|output := tmp
|)
where
real tmp;
end;
process GainL1=
real GAIN;

(? real input;
! real output;
)
(|output := input * GAIN
|)
where
end;
process UnitDelayL1=
integer NB_DELAY;
[NB_DELAY] real INIT_VALUE;

(? real input;
! real output;
)
(|output := input $ NB_DELAY init INIT_VALUE—)
where
end;
process controllerL1=
real INIT_VAL;
```

```
(? real input;
! real output;
)
(|output := input - tmp1
|tmp1 := input $ init INIT_VAL
—)
where
real tmp1;
end;
process plantL1=
integer NB_COEFF_N;
integer NB_COEFF_D;
[NB_COEFF_D] real INIT_VAL;
[NB_COEFF_N] real VAL_COEFF_N;
[NB_COEFF_D] real VAL_COEFF_D;
(? real input;
! real output;)
(|output := (input * VAL_COEFF_N[0]+tmp0)/VAL_COEFF_D[0]
|tmp0:=(- VAL_COEFF_D[1] * output ) $ 1 init INIT_VAL
|)
where
real tmp0;
end;
end;
```

Appendix B

Case Study 2

```
process Sim2Sig =
{integer Integrator3L2_NB_DELAY;
[Integrator3L2_NB_DELAY] real Integrator3L2_INIT_VALUE;
real KeL2_GAIN;
real KtL2_GAIN;
real InductanceL3_GAIN;
integer Integrator1L3_NB_DELAY;
[Integrator1L3_NB_DELAY] real Integrator1L3_INIT_VALUE;
real resistanceL3_GAIN;
real InertiaL3_GAIN;
integer Integrator2L3_NB_DELAY;
[Integrator2L3_NB_DELAY] real Integrator2L3_INIT_VALUE;
real resistance1L3_GAIN;
integer BUF}
(? real DCMotorL1_in1;
! real DCMotorL1_out1;
)
(|DCMotorL1_out1:=DCMotorL1{Integrator3L2_NB_DELAY,Integrator3L2_INIT_VALUE,
KeL2_GAIN,KtL2_GAIN,InductanceL3_GAIN,Integrator1L3_NB_DELAY,
Integrator1L3_INIT_VALUE,resistanceL3_GAIN,InertiaL3_GAIN,
Integrator2L3_NB_DELAY,Integrator2L3_INIT_VALUE,resistance1L3_GAIN,BUF}
(DCMotorL1_in1)|) where
process DCMotorL1=
{integer Integrator3L2_NB_DELAY;
[Integrator3L2_NB_DELAY] real Integrator3L2_INIT_VALUE;
real KeL2_GAIN;
real KtL2_GAIN;
real InductanceL3_GAIN;
```

```
integer Integrator1L3_NB_DELAY;
[Integrator1L3_NB_DELAY] real Integrator1L3_INIT_VALUE;
real resistanceL3_GAIN;
real InertiaL3_GAIN;
integer Integrator2L3_NB_DELAY;
[Integrator2L3_NB_DELAY] real Integrator2L3_INIT_VALUE;
real resistance1L3_GAIN;
integer BUF}
(? real in1;
!real out1;
)
(|Integrator3L2_out1
:=Integrator3L2{Integrator3L2_NB_DELAY,Integrator3L2_INIT_VALUE}(Integrator3L2_in1)
|KeL2_out1 := KeL2{KeL2_GAIN}(KeL2_in1)
|KtL2_out1 := KtL2{KtL2_GAIN}(KtL2_in1)
|S1L2_out1:=S1L2{InductanceL3_GAIN,Integrator1L3_NB_DELAY,Integrator1L3_INIT_VALUE,
resistanceL3_GAIN,BUF}(S1L2_in1,S1L2_in2)
|S2L2_out1:=S2L2{InertiaL3_GAIN,Integrator2L3_NB_DELAY,Integrator2L3_INIT_VALUE,
resistance1L3_GAIN,BUF}(S2L2_in1)
|S1L2_in1 := in1
|out1 := Integrator3L2_out1
|S1L2_in2 := KeL2_out1
|S2L2_in1 := KtL2_out1
|KtL2_in1 := S1L2_out1
|Integrator3L2_in1 := S2L2_out1
|KeL2_in1 := S2L2_out1
|)
where
real S1L2_in1;
real Integrator3L2_out1;
real S1L2_in2;
real KeL2_out1;
real S2L2_in1;
real KtL2_out1;
real KtL2_in1;
real S1L2_out1;
real Integrator3L2_in1;
real S2L2_out1;
real KeL2_in1;
process Integrator3L2=
{integer NB_DELAY;
[NB_DELAY] real INIT_VALUE;
```

```
}
(? real input;
! real output;
)
(|output := input $ NB_DELAY init INIT_VALUE |)
where
end;
process KeL2=
{real GAIN;}
(? real input;
! real output;
)
(|output := input * GAIN |)
where
end;
process KtL2=
{real GAIN;}
(? real input;
! real output;
) (|output := input * GAIN |)
where
end;
process S1L2=
{real InductanceL3_GAIN;
integer Integrator1L3_NB_DELAY;
[Integrator1L3_NB_DELAY] real Integrator1L3_INIT_VALUE;
real resistanceL3_GAIN;
integer BUF}
(? real in1;
real in2;
!real out1;
)
(|AddL3_out1 := AddL3(AddL3_in1,AddL3_in2,AddL3_in3)
|InductanceL3_out1 := InductanceL3{InductanceL3_GAIN}(InductanceL3_in1)
|Integrator1L3_out1 :=Integrator1L3{Integrator1L3_NB_DELAY,Integrator1L3_INIT_VALUE}
(Integrator1L3_in1)
|resistanceL3_out1 := resistanceL3{resistanceL3_GAIN}(resistanceL3_in1)
|AddL3_in2 := in1
|AddL3_in3 := in2
|InductanceL3_in1 := AddL3_out1
|Integrator1L3_in1 := InductanceL3_out1
|resistanceL3_in1 := Integrator1L3_out1
```

```
|out1 := Integrator1L3_out1
|AddL3_in1 := resistanceL3_out1
|)
where
real AddL3_in2;
real AddL3_in3;
real InductanceL3_in1;
real AddL3_out1;
real Integrator1L3_in1;
real InductanceL3_out1;
real resistanceL3_in1;
real Integrator1L3_out1;
real AddL3_in1;
real resistanceL3_out1;
process AddL3=
(? real input0;
real input1;
real input2;
! real output;
)
(|tmp:=-input0 +input1 - input2
|output := tmp |)
where
real tmp;
end;
process InductanceL3=
{real GAIN;
}
(? real input;
! real output;
)
(|output := input * GAIN |)
where
end;
process Integrator1L3=
{integer NB_DELAY;
[NB_DELAY] real INIT_VALUE;
}
(? real input;
! real output;
)
(|output := input $ NB_DELAY init INIT_VALUE |)
```

```
where
end;
process resistanceL3=
{real GAIN;
}
(? real input;
! real output;
)
(|output := input * GAIN
|)
where
end;
end;
process S2L2=
{real InertiaL3_GAIN;
integer Integrator2L3_NB_DELAY;
[Integrator2L3_NB_DELAY] real Integrator2L3_INIT_VALUE;
real resistance1L3_GAIN;
integer BUF}
(? real in1;
!real out1;
)
(|Add1L3_out1 := Add1L3(Add1L3_in1,Add1L3_in2)
|InertiaL3_out1 := InertiaL3{InertiaL3_GAIN}(InertiaL3_in1)
|Integrator2L3_out1 :=Integrator2L3{Integrator2L3_NB_DELAY,Integrator2L3_INIT_VALUE}
(Integrator2L3_in1)
|resistance1L3_out1 := resistance1L3{resistance1L3_GAIN}(resistance1L3_in1)
|Add1L3_in1 := in1
|InertiaL3_in1 := Add1L3_out1
|Integrator2L3_in1 := InertiaL3_out1
|resistance1L3_in1 := Integrator2L3_out1
|out1 := Integrator2L3_out1
|Add1L3_in2 := resistance1L3_out1
|)
where
real Add1L3_in1;
real InertiaL3_in1;
real Add1L3_out1;
real Integrator2L3_in1;
real InertiaL3_out1;
real resistance1L3_in1;
real Integrator2L3_out1;
```

```
real Add1L3_in2;
real resistance1L3_out1;
process Add1L3=
(? real input0;
real input1;
! real output;
)
(|tmp:=+input0 - input1
|output := tmp
|)
where
real tmp;
end;
process InertiaL3=
{real GAIN;
}
(? real input;
! real output;
)
(|output := input * GAIN |)
where
end;
process Integrator2L3=
{integer NB_DELAY;
[NB_DELAY] real INIT_VALUE;
}
(? real input;
! real output;
)
(|output := input $ NB_DELAY init INIT_VALUE |)
where
end;
process resistance1L3=
{real GAIN;}
(? real input;
! real output;)
(|output := input * GAIN |)
where
end;
end;
end;
end;
```

Appendix C

Case Study 3

process Sim2Sig =
{integer Integrator3L2_NB_DELAY;
[Integrator3L2_NB_DELAY] real Integrator3L2_INIT_VALUE;
real KeL2_GAIN;
real KtL2_GAIN;
real InductanceL3_GAIN;
integer Integrator1L3_NB_DELAY;
[Integrator1L3_NB_DELAY] real Integrator1L3_INIT_VALUE;
real resistanceL3_GAIN;
real InertiaL3_GAIN;
integer Integrator2L3_NB_DELAY;
[Integrator2L3_NB_DELAY] real Integrator2L3_INIT_VALUE;
real resistance1L3_GAIN;
integer UnitDelayL2_NB_DELAY;
[UnitDelayL2_NB_DELAY] real UnitDelayL2_INIT_VALUE;
integer AddL2_PERIODE;
integer AddL2_PHASE;
real DGainL2_GAIN;
real IGainL2_GAIN;
real IGain1L2_GAIN;
real PGainL2_GAIN;
integer UnitDelay1L2_NB_DELAY;
[UnitDelay1L2_NB_DELAY] real UnitDelay1L2_INIT_VALUE;
integer UnitDelay2L2_NB_DELAY;
[UnitDelay2L2_NB_DELAY] real UnitDelay2L2_INIT_VALUE;
real PulseGeneratorL1_Amplitude;
real RateAdjustmentL1_GAIN;
integer RateAdjustmentL1_PERIODE;

```
integer RateAdjustmentL1_PHASE;
integer BUF}
(? real InputSamplingL1_in1;
! real DCMotorL1_out1;
)
(|AddL1_out1 := AddL1(AddL1_in1,AddL1_in2)
|DCMotorL1_out1:=DCMotorL1{Integrator3L2_NB_DELAY,Integrator3L2_INIT_VALUE,
KeL2_GAIN,KtL2_GAIN,InductanceL3_GAIN,Integrator1L3_NB_DELAY,
Integrator1L3_INIT_VALUE,resistanceL3_GAIN,InertiaL3_GAIN,
Integrator2L3_NB_DELAY,Integrator2L3_INIT_VALUE,resistance1L3_GAIN,BUF}
(DCMotorL1_in1)
|InputSamplingL1_out1:=InputSamplingL1{UnitDelayL2_NB_DELAY,
UnitDelayL2_INIT_VALUE,BUF}(InputSamplingL1_in1 when InputSamplingL1_en)
cell ^InputSamplingL1_in1 | InputSamplingL1_en := EnableL2(InputSamplingL1_in_en)
|PIDL1_out1:=PIDL1{AddL2_PERIODE,AddL2_PHASE,DGainL2_GAIN,IGainL2_GAIN,
IGain1L2_GAIN,PGainL2_GAIN,UnitDelay1L2_NB_DELAY,UnitDelay1L2_INIT_VALUE,
UnitDelay2L2_NB_DELAY,UnitDelay2L2_INIT_VALUE,BUF}(PIDL1_in1)
|PulseGeneratorL1_out1:=PulseGeneratorL1{PulseGeneratorL1_Amplitude}()
|RateAdjustmentL1_out1 := RateAdjustmentL1{RateAdjustmentL1_GAIN,
RateAdjustmentL1_PERIODE,RateAdjustmentL1_PHASE}(RateAdjustmentL1_in1)
|PIDL1_in1 := AddL1_out1
|RateAdjustmentL1_in1 := DCMotorL1_out1
|AddL1_in1 := InputSamplingL1_out1
|DCMotorL1_in1 := PIDL1_out1
|InputSamplingL1_in_en := PulseGeneratorL1_out1
|AddL1_in2 := RateAdjustmentL1_out1
|)
where
real PIDL1_in1;
real AddL1_out1;
real RateAdjustmentL1_in1;
real DCMotorL1_out1;
real AddL1_in1;
real InputSamplingL1_out1;
real PIDL1_out1;
real DCMotorL1_in1;
real InputSamplingL1_in_en,PulseGeneratorL1_out1;
boolean InputSamplingL1_en;
real AddL1_in2;
real RateAdjustmentL1_out1;
process AddL1=
(? real input0;
```

```
real input1;
! real output;
)
(|tmp:=+input0 - input1
|output := tmp
|)
where
real tmp;
end;
process DCMotorL1=
{integer Integrator3L2_NB_DELAY;
[Integrator3L2_NB_DELAY] real Integrator3L2_INIT_VALUE;
real KeL2_GAIN;
real KtL2_GAIN;
real InductanceL3_GAIN;
integer Integrator1L3_NB_DELAY;
[Integrator1L3_NB_DELAY] real Integrator1L3_INIT_VALUE;
real resistanceL3_GAIN;
real InertiaL3_GAIN;
integer Integrator2L3_NB_DELAY;
[Integrator2L3_NB_DELAY] real Integrator2L3_INIT_VALUE;
real resistance1L3_GAIN;
integer BUF}
(? real in1;
!real out1;
)
(|Integrator3L2_out1 :=Integrator3L2{Integrator3L2_NB_DELAY,Integrator3L2_INIT_VALUE}
(Integrator3L2_in1)
|KeL2_out1 := KeL2{KeL2_GAIN}(KeL2_in1)
|KtL2_out1 := KtL2{KtL2_GAIN}(KtL2_in1)
|S1L2_out1:=S1L2{InductanceL3_GAIN,Integrator1L3_NB_DELAY,
Integrator1L3_INIT_VALUE,resistanceL3_GAIN,BUF}(S1L2_in1,S1L2_in2)
|S2L2_out1:=S2L2{InertiaL3_GAIN,Integrator2L3_NB_DELAY,
Integrator2L3_INIT_VALUE,resistance1L3_GAIN,BUF}(S2L2_in1)
|S1L2_in1 := in1
|out1 := Integrator3L2_out1
|S1L2_in2 := KeL2_out1
|S2L2_in1 := KtL2_out1
|KtL2_in1 := S1L2_out1
|Integrator3L2_in1 := S2L2_out1
|KeL2_in1 := S2L2_out1
|)
```

```
where
real S1L2_in1;
real Integrator3L2_out1;
real S1L2_in2;
real KeL2_out1;
real S2L2_in1;
real KtL2_out1;
real KtL2_in1;
real S1L2_out1;
real Integrator3L2_in1;
real S2L2_out1;
real KeL2_in1;
process Integrator3L2=
{integer NB_DELAY;
[NB_DELAY] real INIT_VALUE;
}
(? real input;
! real output;
)
(|output := input $ NB_DELAY init INIT_VALUE
|)
where
end;
process KeL2=
{real GAIN;}
(? real input;
! real output;
)
(|output := input * GAIN
|)
where
end;
process KtL2=
{real GAIN;
}
(? real input;
! real output;
)
(|output := input * GAIN
|)
where
end;
```

```
process S1L2=
{real InductanceL3_GAIN;
integer Integrator1L3_NB_DELAY;
[Integrator1L3_NB_DELAY] real Integrator1L3_INIT_VALUE;
real resistanceL3_GAIN;
integer BUF}
(? real in1;
real in2;
!real out1;
)
(|AddL3_out1 := AddL3(AddL3_in1,AddL3_in2,AddL3_in3)
|InductanceL3_out1 := InductanceL3{InductanceL3_GAIN}(InductanceL3_in1)
|Integrator1L3_out1 :=Integrator1L3{Integrator1L3_NB_DELAY,Integrator1L3_INIT_VALUE}
(Integrator1L3_in1)
|resistanceL3_out1 := resistanceL3{resistanceL3_GAIN}(resistanceL3_in1)
|AddL3_in2 := in1
|AddL3_in3 := in2
|InductanceL3_in1 := AddL3_out1
|Integrator1L3_in1 := InductanceL3_out1
|resistanceL3_in1 := Integrator1L3_out1
|out1 := Integrator1L3_out1
|AddL3_in1 := resistanceL3_out1
|)
where
real AddL3_in2;
real AddL3_in3;
real InductanceL3_in1;
real AddL3_out1;
real Integrator1L3_in1;
real InductanceL3_out1;
real resistanceL3_in1;
real Integrator1L3_out1;
real AddL3_in1;
real resistanceL3_out1;
process AddL3=
(? real input0;
real input1;
real input2;
! real output;
)
(|tmp:=-input0 + input1 - input2
|output := tmp
```

```
|)
where
real tmp;
end;
process InductanceL3=
{real GAIN;}
(? real input;
! real output;
)
(|output := input * GAIN|)
where
end;
process Integrator1L3=
{integer NB_DELAY;
[NB_DELAY] real INIT_VALUE;
}
(? real input;
! real output;
)
(|output := input $ NB_DELAY init INIT_VALUE
|)
where
end;
process resistanceL3=
{real GAIN;}
(? real input;
! real output;
)
(|output := input * GAIN
|)
where
end;
end;
process S2L2=
{real InertiaL3_GAIN;
integer Integrator2L3_NB_DELAY;
[Integrator2L3_NB_DELAY] real Integrator2L3_INIT_VALUE;
real resistance1L3_GAIN;
integer BUF}
(? real in1;
!real out1;
)
```

```
(|Add1L3_out1 := Add1L3(Add1L3_in1,Add1L3_in2)
|InertiaL3_out1 := InertiaL3{InertiaL3_GAIN}(InertiaL3_in1)
|Integrator2L3_out1 :=Integrator2L3{Integrator2L3_NB_DELAY,Integrator2L3_INIT_VALUE}
(Integrator2L3_in1)
|resistance1L3_out1 := resistance1L3{resistance1L3_GAIN}(resistance1L3_in1)
|Add1L3_in1 := in1
|InertiaL3_in1 := Add1L3_out1
|Integrator2L3_in1 := InertiaL3_out1
|resistance1L3_in1 := Integrator2L3_out1
|out1 := Integrator2L3_out1
|Add1L3_in2 := resistance1L3_out1
|)
where
real Add1L3_in1;
real InertiaL3_in1;
real Add1L3_out1;
real Integrator2L3_in1;
real InertiaL3_out1;
real resistance1L3_in1;
real Integrator2L3_out1;
real Add1L3_in2;
real resistance1L3_out1;
process Add1L3=
(? real input0;
real input1;
! real output;
)
(|tmp:=+input0 - input1
|output := tmp
|)
where
real tmp;
end;
process InertiaL3=
{real GAIN;}
(? real input;
! real output;
)
(|output := input * GAIN
|)
where
end;
```

```
process Integrator2L3=
{integer NB_DELAY;
[NB_DELAY] real INIT_VALUE;
}
(? real input;
! real output;
)
(|output := input $ NB_DELAY init INIT_VALUE|)
where
end;
process resistance1L3=
{real GAIN;}
(? real input;
! real output;)
(|output := input * GAIN|)
where
end;
end;
end;
process InputSamplingL1=
{integer UnitDelayL2_NB_DELAY;
[UnitDelayL2_NB_DELAY] real UnitDelayL2_INIT_VALUE;
integer BUF}
(? real in1;
!real out1;
)
(|UnitDelayL2_out1 :=UnitDelayL2{UnitDelayL2_NB_DELAY,UnitDelayL2_INIT_VALUE}
(UnitDelayL2_in1)
|UnitDelayL2_in1 := in1
|out1 := UnitDelayL2_out1
|)
where
real UnitDelayL2_in1;
real UnitDelayL2_out1;
process UnitDelayL2=
{integer NB_DELAY;
[NB_DELAY] real INIT_VALUE;
}
(? real input;
! real output;
)
(|output := input $ NB_DELAY init INIT_VALUE|)
```

```
where
end;
end;
process EnableL2=
( ? real input;
! boolean en;)
(|en := (true when (input¿real(0))) default (false when (input¡=real(0)))|)
where
end;
process PIDL1=
{integer AddL2_PERIODE;
integer AddL2_PHASE;
real DGainL2_GAIN;
real IGainL2_GAIN;
real IGain1L2_GAIN;
real PGainL2_GAIN;
integer UnitDelay1L2_NB_DELAY;
[UnitDelay1L2_NB_DELAY] real UnitDelay1L2_INIT_VALUE;
integer UnitDelay2L2_NB_DELAY;
[UnitDelay2L2_NB_DELAY] real UnitDelay2L2_INIT_VALUE;
integer BUF}
(? real in1;
!real out1;
)
(|AddL2_out1 := AddL2{AddL2_PERIODE,AddL2_PHASE}(AddL2_in1,AddL2_in2,AddL2_in3)
|Add1L2_out1 := Add1L2(Add1L2_in1,Add1L2_in2)
|DGainL2_out1 := DGainL2{DGainL2_GAIN}(DGainL2_in1)
|IGainL2_out1 := IGainL2{IGainL2_GAIN}(IGainL2_in1)
|IGain1L2_out1 := IGain1L2{IGain1L2_GAIN}(IGain1L2_in1)
|PGainL2_out1 := PGainL2{PGainL2_GAIN}(PGainL2_in1)
|UnitDelay1L2_out1 :=UnitDelay1L2{UnitDelay1L2_NB_DELAY,
UnitDelay1L2_INIT_VALUE}
(UnitDelay1L2_in1)
|UnitDelay2L2_out1 :=UnitDelay2L2{UnitDelay2L2_NB_DELAY,UnitDelay2L2_INIT_VALUE}
(UnitDelay2L2_in1)
|DGainL2_in1 := in1
|IGainL2_in1 := in1
|PGainL2_in1 := in1
|out1 := AddL2_out1
|IGain1L2_in1 := Add1L2_out1
|Add1L2_in1 := DGainL2_out1
|UnitDelay2L2_in1 := IGainL2_out1
```

```
|AddL2_in3 := IGain1L2_out1
|UnitDelay1L2_in1 := IGain1L2_out1
|AddL2_in1 := PGainL2_out1
|Add1L2_in2 := UnitDelay1L2_out1
|AddL2_in2 := UnitDelay2L2_out1
|)
where
real DGainL2_in1;
real IGainL2_in1;
real PGainL2_in1;
real AddL2_out1;
real IGain1L2_in1;
real Add1L2_out1;
real Add1L2_in1;
real DGainL2_out1;
real UnitDelay2L2_in1;
real IGainL2_out1;
real AddL2_in3;
real IGain1L2_out1;
real UnitDelay1L2_in1;
real AddL2_in1;
real PGainL2_out1;
real Add1L2_in2;
real UnitDelay1L2_out1;
real AddL2_in2;
real UnitDelay2L2_out1;
process AddL2=
{integer PERIODE;
integer PHASE;}
(? real input0;
real input1;
real input2;
! real output;
)
(|tmp:=+input0 + input1 + input2
|cnt := (cnt+1) $ init PHASE
|cnt2 := cnt modulo PERIODE
|cnt ^ = input1
|output := tmp when (cnt2=0)
|)
where
real tmp;
```

```
integer cnt, cnt2;
end;
process Add1L2=
(? real input0;
real input1;
! real output;
)
(|tmp:=+input0 - input1
|output := tmp
|)
where
real tmp;
end;
process DGainL2=
{real GAIN;}
(? real input;
! real output;
)
(|output := input * GAIN
|)
where
end;
process IGainL2=
{real GAIN;}
(? real input;
! real output;
)
(|output := input * GAIN|)
where
end;
process IGain1L2=
{real GAIN;}
(? real input;
! real output;
)
(|output := input * GAIN|)
where
end;
process PGainL2=
{real GAIN;}
(? real input;
! real output;
```

```
)
(|output := input * GAIN|)
where
end;
process UnitDelay1L2=
{integer NB_DELAY;
[NB_DELAY] real INIT_VALUE;
}
(? real input;
! real output;
)
(|output := input $ NB_DELAY init INIT_VALUE|)
where
end;
process UnitDelay2L2=
{integer NB_DELAY;
[NB_DELAY] real INIT_VALUE;
}
(? real input;
! real output;
)
(|output := input $ NB_DELAY init INIT_VALUE|)
where
end;
end;
process PulseGeneratorL1=
{real AMPLITUDE}
(? !real out;)
(| dpg1:= dpg2 $ 1 init AMPLITUDE
| dpg2:= dpg3 $ 1 init AMPLITUDE
| dpg3:= dpg4 $ 1 init AMPLITUDE
| dpg4:= dpg5 $ 1 init AMPLITUDE
| dpg5:= dpg6 $ 1 init AMPLITUDE
| dpg6:= dpg7 $ 1 init real(0)
| dpg7:= dpg8 $ 1 init real(0)
| dpg8:= dpg9 $ 1 init real(0)
| dpg9:= dpg10 $ 1 init real(0)
| dpg10:= dpg1 $ 1 init real(0)
| out := dpg1
|)
where
real dpg1,dpg2,dpg3,dpg4,dpg5,dpg6,dpg7,dpg8,dpg9,dpg10;
```

```
end;
process RateAdjustmentL1=
{real GAIN;
integer PERIODE;
integer PHASE;
}
(? real input;
! real output;
)
(|cnt := (cnt+1) $ init PHASE
|cnt2 := cnt modulo PERIODE
|cnt ^= output
|input ^= when (cnt2=0)
|tmp := input * GAIN
|output := tmp cell ^ output|)
where
integer cnt, cnt2;
real tmp;
end;
end;
```