

# Applying Timed Interval Calculus to Simulink Diagrams

Chunqing Chen and Jin Song Dong

School of Computing  
National University of Singapore  
{`chenchun`, `dongjs`}@`comp.nus.edu.sg`

**Abstract.** Simulink has been used widely as an industry tool to model and simulate embedded systems. With increasing usage of embedded systems in real-time safety-critical situations, Simulink is deficient to cope with the requirements of high-level assurance and timing analysis. In this paper, we present a systematic approach to translate Simulink diagrams to Timed Interval Calculus (TIC), a notation extending Z to support real-time system specification and verification. This work is based on the same angle chosen by Simulink and TIC where they model systems in terms of continuous time. Translated TIC specifications preserve the functional and timing aspects of the diagrams, and cover a wide range of Simulink blocks. After the translation, we can increase the design space by specifying important requirements, especially timing constraints exactly on the system or its components. Moreover, we can take advantage of TIC reasoning rules to formally verify systems with requirements, and hence elevate the design quality of Simulink.

**Keywords:** Simulink, Real-Time Specification, Z, Verification

## 1 Introduction

Simulink [18] is a graphical environment used widely for modelling and simulating embedded systems. A Simulink diagram is formed by connecting blocks with wires to represent a set of mathematical functions that model system behavior over time. Simulink adopts continuous-time semantics [12] to support discrete (multi-rate), continuous and hybrid systems. Its simulation facility assists designers to analyze system behavior visually under specific parameters, initial conditions and simulation period.

With increasing usage of embedded systems in real-time safety-critical situations, high-level assurance is required and timing analysis becomes necessary [21]. Simulink is deficient to cope with the complexity by two factors: one is that each simulation in Simulink reflects system behavior under a particular circumstance, and hence it is infeasible to examine the behavior of infinite values of parameters or an infinite simulation period; the other is that Simulink is difficult in specifying and analyzing timing constraints, since it adopts an idealized timing model for block execution and communication. Recently, formal methods receive

more attention to complement Simulink by their rigorous semantics and formal verification capability [25]. We propose to apply a real-time formal notation, i.e. Timed Interval Calculus (TIC) [8] to model Simulink diagrams, and thus to complement Simulink by TIC formal verification capability.

TIC [8] is set-theory based and extends the work of Mahony and Hayes [14]. It makes use of the continuous functions of time to model system, and defines interval brackets to concisely express properties in terms of intervals. TIC reuses Z [26] mathematical and schema notations. It has been applied to a number of real-time systems, including a mine-shaft pump system [14, 24], a speedometer [8] and a controller for barometric altimeter [6]. Dawson and Goré [5] have formalized TIC reasoning rules using generic theorem prover Isabelle [20] for machine-assisted proof. The similar formalism that can handle real-time systems is Duration Calculus (DC) [28]. It uses the integration of Boolean-valued states over closed and bounded intervals to specify critical duration constraints. Its extensions, Mean Value Calculus [27] adopts the mean value of states to express properties in point intervals, and Extended Duration Calculus [29] defines two functions to give the values of state at the endpoints of an interval. Because they describe system behavior without the explicit references to absolute time, they are limited to represent the constraints which restrict the values of the interval endpoints for specific intervals.

The approach is based on the same angle adopted by Simulink and TIC when modelling systems in terms of continuous time. It can cover a wide range of Simulink blocks of different categories. The behavior of Simulink library blocks is described informally in [17], so we focus on capturing their denotational semantics, i.e. the mathematical functions between their inputs and outputs over time. We thus define a set of TIC library functions to model the library blocks. Based on the TIC library, we develop a strategy to translate Simulink diagrams into TIC specifications by modelling the components and connections of the diagrams. The strategy can derive and keep the sample time of elementary blocks during the translation. Furthermore, Simulink conditionally executed subsystems are taken into account as well. Hence, the generated TIC specifications preserve the functional and timing aspects of the diagrams. A tool has been implemented using JAVA to experiment our strategy. After the translation, we can precisely and easily express important requirements such as timing related *safety* and *liveness* on a system or one of its components. Verification is a deduction to show that the system satisfies requirements with the use of TIC reasoning rules. Our approach even allows the analysis of open systems which are not checkable by simulation in Simulink. Therefore, using TIC can increase the design space and elevate the design quality of Simulink.

Recently, there are a number of work on translating Simulink into other formal notations or programming languages. Arthan et al [2], Adams and Clayton [1] translate Simulink diagrams into Z by capturing the functional behavior of one cycle. Cavalcanti et al [4] extend the work by using Circus to capture the functionality and concurrency of Simulink diagrams. Their approaches aim to verify that Simulink diagrams are correctly implemented in programming lan-

guage Ada, and that is different from ours. Our goal is to validate that Simulink diagrams satisfy different requirements. Moreover, they consider only single-rate discrete systems, and timing information is missing. Similarly, Caspi et al [3] focus on only discrete Simulink blocks though it supports multi-rate systems. Other approaches [22, 23, 9] take Simulink/ Stateflow <sup>1</sup> Models (SSMs) into account. Sims et al [22] verify SSMs with an invariant checker, and the translation from SSMs to the input language of the checker is performed by hand. Tiwari et al [23] reduce certain accuracy grade by discretizing differential equations represented by Simulink diagrams into difference equations denoting discrete transition systems. Gupta et al [9] develop a tool *CheckMate* that can automatically verify customized SSMs, but the type of constraints supported is limited to the linear inequality that allows one variable only. Jersak et al [13] report on translating Simulink diagrams to SPI models for timing analysis. However, the communication in SPI models is represented by FIFO queues, and that is different from wires in Simulink. In short, our approach covers more types of Simulink blocks and supports more complex requirements.

The rest of the paper is organized as follows. Section 2 introduces Simulink and TIC. Section 3 defines the set of TIC library functions for Simulink library blocks. Section 4 presents the translation strategy. Section 5 shows that the TIC verification capability can complement Simulink. Section 6 concludes the paper with possible future work.

## 2 Background

### 2.1 Simulink

A Simulink diagrams is formed by connecting blocks with wires to represent a set of mathematical functions which specify system behavior over time. Elementary blocks are units. Each denotes a primitive mathematical relationship over its inputs and outputs, for example, a summation of two inputs. An elementary block is an instance of a Simulink library block using parameterization technique, i.e., generated by assigning specific values to the parameters of the library block. Hence, given different values of parameters, a library block can create the elementary blocks with different functionalities. To support hierarchical modelling, a Simulink block can be a Simulink diagram as well to represent a subsystem. A wire is a directed edge to indicate dependency relationships between connected blocks. Namely, the source block (destination block) can write (read) the value to (from) the wire according to its sample time.

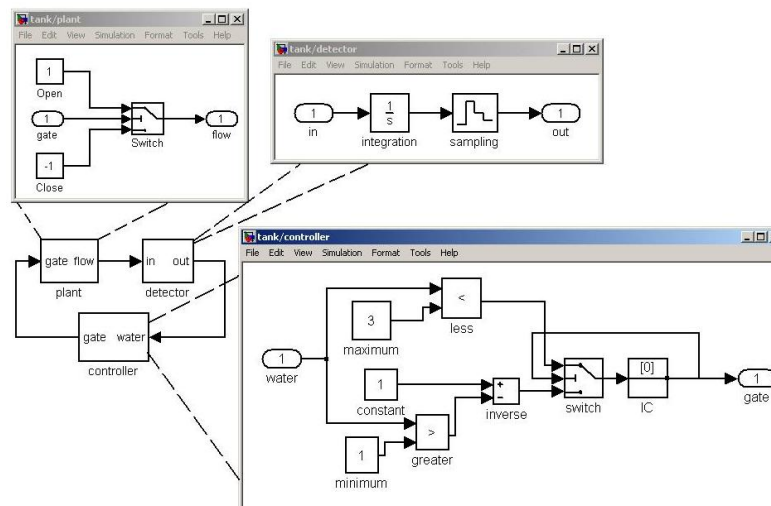
Every Simulink elementary block is assigned a sample time as the rate at which the block executes in simulation. A sample time of a block can be determined by parameter *SampleTime*, by *sample time propagation* rules, or by block type (e.g. blocks from continuous library always have a continuous sample time). Simulink supports various systems such as continuous, discrete and hybrid

---

<sup>1</sup> Stateflow [16] combines flow diagrams and statecharts for control logic and can be integrated into Simulink.

systems. It adopts continuous-time semantics as a unifying domain. Hence the discrete systems behave piecewise-constant continuously. Simulink also supports conditionally executed subsystems whose executions depend on the value of an input, i.e. control signal. For instances, an *enabled* subsystem is active when the control signal is positive, and a *triggered* subsystem is active when a trigger event in the control signal occurs.

*Example 1.* We introduce a water tank system as a running example to explain and illustrate our main ideas and results. The system consists of a tank, a drainage outlet, a gate, a detector sensing water volume every 1 second, and a controller for the gate based on the value from the detector. Initially, the tank is full of water as represented by value 4 and the gate is closed. When the gate closes, the water flow rate (denoted by *flow*) is  $-1 \leq \textit{flow} < 0$  where the negative sign indicates that the water volume decreases. When the gate opens, water flows into the tank with the range  $0 \leq \textit{flow} \leq 1$ . The objective is to prevent the tank from overflow or being empty. The Simulink diagram for the system with constant water flow rates is shown in Figure 1. In the diagram, each box is a Simulink block. Subsystems are used to provide a hierarchical structure of the system. Each ellipse inside a subsystem block represents an interface. To be specific, subsystem *plant* outputs different water flow rate (denoted by *flow*) according to the value from the input (denoted by *gate*); subsystem *detector* continuously integrates the water flow rate, as well as samples and holds the result every 1 second; subsystem *controller* implements the logic to control the gate. Namely, it opens the gate by outputting value 1 when the water volume (denoted by *water*) is not greater than value 1, and closes the gate by yielding value 0 when the water volume is not less than value 3.



**Fig. 1.** The *water tank* system with its subsystems in Simulink

A Simulink diagram is stored in a structured ASCII file which is called *model file*. The file describes the Simulink diagram by keywords and parameter-value pairs. The parameter-value pairs capture the contents of the Simulink diagram by assigning values to relevant parameters. The use of keywords followed by a pair of brackets arranges contents in the same hierarchical order of the Simulink diagram. For example, part of the contents of the block *integration* in the subsystem *detector* is given below.

```

System { Name                "detector"
        Block { BlockType    Integrator
                Name          "integration"
                InitialCondition "4" } }

```

In the above example, the parameter-value pair “*BlockType Integrator*” indicates that the block named *integration* executes an integration function. Note that the exact mathematical expression, i.e. the integration is not shown directly. As the contents of the block is within the pair of brackets following the keyword *System*, it shows that the block is a component of the system *detector*. Moreover, the model file contains more information which is not expressed graphically, for example, the initial value 4 is not shown in Figure 1. Thus, the model file is the input of our translation from Simulink to Timed Interval Calculus.

## 2.2 Timed Interval Calculus

TIC is set-theory based and reuses the  $\mathbb{Z}$  mathematical and schema notations. It adopts total functions of time to model system behavior, and defines interval brackets to concisely express properties in terms of intervals.

Time domain  $\mathbb{T}$  is nonnegative real numbers, i.e.,  $\mathbb{R}_+ \cup \{0\}$ . An interval is a contiguous range of time points. There are four types based on whether the endpoints are included, namely, left-open ( $($ ), left-closed ( $[$ ), right-open ( $)$ ) and right-closed ( $]$ ). For example, a left-closed, right-open interval is defined below.

$$\left| \begin{array}{l} [ \dots ) : \mathbb{T} \times \mathbb{T} \rightarrow \mathbb{P} \mathbb{T} \\ \forall x, y : \mathbb{R} \bullet [x \dots y) = \{z : \mathbb{T} \mid x \leq z < y\} \end{array} \right.$$

Note that a point interval can be depicted by a left-closed, right-closed interval, and the empty set is not an interval. Symbol  $\mathbb{I}$  denotes the set of all nonempty intervals. Operators  $\alpha, \omega$  and  $\delta$  return the infimum, supremum and length of an interval respectively.

TIC defines each system variable as a total function of time. For example, a water height can be represented by the function *height* :  $\mathbb{T} \rightarrow \mathbb{R}$ .

A TIC expression denotes a set of intervals during which a predicate holds everywhere. A predicate is a function from time to Boolean ( $\mathbb{B} ::= \text{true} \mid \text{false}$ ). Since the operators  $\alpha, \omega$  and  $\delta$  which are the functions of intervals can be applied in a predicate, the predicate is evaluated with respect to time points and intervals simultaneously by applying the *lifting* function [15]. The lifting function ( $\uparrow$ ) can

generalize operators of simple type to complex type. For example, the predicate,  $height(\alpha) \leq height$ , would be lifted to a lambda abstraction which assigns a time point and an interval to the appropriate place in the predicate at the same time.

$$(height(\alpha) \leq height) \uparrow^{\mathbb{I}, \mathbb{T}} = \lambda \Delta : \mathbb{I}; t : \mathbb{T} \bullet height(\alpha(\Delta)) \leq height(t)$$

The following specification defines a set of left-closed, right-open intervals of the predicate  $TP : \mathbb{T} \rightarrow \mathbb{B}$  with the use of lifting function. Note that there are other three types of sets of intervals and they can be defined in the similar way.

$$\left| \begin{array}{l} \{ \_ \} : (\mathbb{T} \rightarrow \mathbb{B}) \leftrightarrow \mathbb{I} \\ \hline \forall TP : \mathbb{T} \rightarrow \mathbb{B} \bullet \\ \{TP\} = \{x, y : \mathbb{T} \mid \forall t : \{x \dots y\} \bullet TP \uparrow^{\mathbb{I}, \mathbb{T}} (\{x \dots y\}, t) \bullet \{x \dots y\} \} \end{array} \right.$$

For example, applying the brackets to previous predicate,  $\{height(\alpha) \leq height\}$ , it returns all the left-closed, right-open intervals during which the water height is not less then the height at the beginning of the interval as expanded below.

$$\{ height(\alpha) \leq height \} = \{x, y : \mathbb{T} \mid \forall t : \{x \dots y\} \bullet (height(\alpha) \leq height) \uparrow^{\mathbb{I}, \mathbb{T}} (\{x \dots y\}, t) \bullet \{x \dots y\} \}$$

In general, a set of intervals with unspecified endpoints is defined:

$$\left| \begin{array}{l} \{ \_ \} : (\mathbb{T} \rightarrow \mathbb{B}) \leftrightarrow \mathbb{I} \\ \hline \forall TP : \mathbb{T} \rightarrow \mathbb{B} \bullet \{ \_ \} = (TP) \cup \{TP\} \cup \{TP\} \cup \{TP\} \end{array} \right.$$

Since TIC is based on the set theory, the set operators such as union ( $\cup$ ) and intersection ( $\cap$ ) can be used to construct new sets of intervals. In addition, to capture the sequences of behaviors, a concatenation operator ( $\curvearrowright$ ) is defined to connect consecutive intervals end-to-end.

$$\left| \begin{array}{l} \_ \curvearrowright \_ : \mathbb{I} \times \mathbb{I} \leftrightarrow \mathbb{I} \\ \hline \forall X, Y : \mathbb{I} \bullet X \curvearrowright Y = \\ \{x : X; y : Y; z : \mathbb{I} \mid z = x \cup y \wedge (\forall t1 : x; t2 : y \bullet t1 < t2) \bullet z\} \end{array} \right.$$

A TIC predicate is formed from the TIC expressions with logical operators ( $\neg$ ,  $\vee$  and so on). We transform system logical properties into real-time properties. Namely, they can be represented by constraints on different sets of intervals. For example, the timing constraint, “*within any closed interval which starts from a multiple of 5 seconds and lasts for 1 second, the critical property P must hold.*”, can be specified by the TIC predicate  $\{\exists k : \mathbb{N} \bullet \alpha = 5 * k \wedge \delta = 1\} \subseteq \{P\}$ .

To manage predicates in a structured way, we utilize the Z schema to group a list of variables in the declaration part and constrain the relationships among the variables in the predicate part. For example, a timing liveness property, that when the water height exceeds the maximum for more than 10 time units an alarm should be on and last till the end, can be modelled below.

<i>AlarmON</i>
$height : \mathbb{T} \rightarrow \mathbb{R}; maximum : \mathbb{R}; alarm : \mathbb{T} \rightarrow \mathbb{B}$
$\{height > maximum \wedge \delta > 10\} \subseteq \llbracket true \rrbracket \curvearrowright \llbracket alarm \rrbracket$

In the above predicate part, the expression  $\{height > maximum \wedge \delta > 10\}$  represents the intervals in which the variable *height* exceeds the constant *maximum* and whose length exceeds 10 time units; while another expression  $\llbracket true \rrbracket \curvearrowright \llbracket alarm \rrbracket$  denotes the intervals in which the variable *alarm* remains true from some time <sup>2</sup>. The relation between these two sets of intervals is hence constrained by the set comparison operator  $\subseteq$ . The specification is called a TIC schema which extends the conventional Z schema to support the TIC-defined operators. Hence in TIC, a system is modelled by a collection of TIC schemas.

TIC provides a rich set of reasoning rules [8, 6, 24]. They are derived from the set theory to capture the properties of sets of intervals and the interval concatenations. A typical verification is a deduction to show that system design implies requirements where they are specified in TIC. Due to the page limit, we list below some rules used in our later verification. Symbols *P*, *Q* and *R* denote predicates; *S*, *T*, *S'* and *T'* represent sets of intervals .

**Rule 1:** If the predicate that *P* implies *Q* holds all the time, then

$$(\forall t : \mathbb{T} \bullet P \uparrow^{\mathbb{T}} t \Rightarrow Q \uparrow^{\mathbb{T}} t) \Rightarrow \llbracket P \rrbracket \subseteq \llbracket Q \rrbracket. \text{ A derivative is: } \llbracket P \wedge Q \rrbracket \subseteq \llbracket P \rrbracket$$

**Rule 2:** Transitivity on sets of intervals :

$$(\llbracket P \rrbracket \subseteq \llbracket Q \rrbracket) \wedge (\llbracket Q \rrbracket \subseteq \llbracket R \rrbracket) \Rightarrow \llbracket P \rrbracket \subseteq \llbracket R \rrbracket$$

**Rule 3:** Monotonicity on concatenation:

$$(S \subseteq S' \wedge T \subseteq T') \Rightarrow (S \curvearrowright T \subseteq S' \curvearrowright T')$$

**Rule 4:** Concatenate duration: If  $\alpha$ ,  $\omega$  and  $\delta$  do not occur in *P*, and *r*, *s* are positive values of type  $\mathbb{T}$ , then  $\llbracket P \wedge \delta = r+s \rrbracket = \llbracket P \wedge \delta = r \rrbracket \curvearrowright \llbracket P \wedge \delta = s \rrbracket$

### 3 TIC Library Functions for Simulink Library Blocks

Simulink library blocks act as templates to produce elementary blocks by the parameterization technique. Simulink elementary blocks are the units to construct Simulink diagrams. In this section, we firstly describe the general structure of TIC schema for the elementary blocks, and then construct a set of TIC library functions for the library blocks.

#### 3.1 TIC schemas for Simulink elementary blocks

An elementary block denotes a mathematical function between its inputs and outputs at all points in time. In general, it can be characterized by a tuple  $\{Ins, Out, Ps, \mathcal{F}\}$  where *Ins* is a set of inputs, *Out* is an output, *Ps* is a set of parameters and  $\mathcal{F}$  is the mathematical function. The function computes output based on its inputs and parameters, i.e.  $\mathcal{F} : (Ins \times Ps) \rightarrow Out$ .

<sup>2</sup>  $\llbracket true \rrbracket$  denotes any nonempty interval in the form of  $\llbracket - \rrbracket$

We translate each elementary block to a TIC schema. The schema declares each input and output as a total function from time to real numbers. The assumption of the range type of the function is acceptable since different data types in Simulink only affect simulation efficiency. Furthermore, in a Simulink diagram, each elementary block is assigned a specific sample time value as the rate at which it is executed during simulation. To capture this timing aspect, a schema variable  $st : \mathbb{T}$  is declared in the translated schema. The basic schema structure of an elementary block can be modelled as below.

$$\boxed{\begin{array}{l} \textit{BasicBlk} \\ \textit{Ins} : \mathbb{P}(\mathbb{T} \rightarrow \mathbb{R}); \textit{Out} : \mathbb{T} \rightarrow \mathbb{R}; \textit{Ps} : \mathbb{P}\mathbb{R} \\ \mathcal{F} : (\mathbb{P}(\mathbb{T} \rightarrow \mathbb{R}) \times \mathbb{P}\mathbb{R}) \rightarrow (\mathbb{T} \rightarrow \mathbb{R}); \textit{st} : \mathbb{T} \end{array}}$$

Blocks having sample time value 0 are said to have continuous sample times. Such a block executes its function at every time point. Its output depends on its inputs either at current time point or through a period, for example, an integration requires calculation over an interval. Hence the block behavior is modelled in terms of intervals instead of time points by a TIC predicate  $\mathbb{I} = \llbracket \mathcal{F}(\textit{Ins}, \textit{Ps}) = \textit{Out} \rrbracket$ . The schema structure for a continuous block can be represented as below.

$$\textit{EleBlk}_C \hat{=} [\textit{BasicBlk} \mid \textit{st} = 0 \wedge \mathbb{I} = \llbracket \mathcal{F}(\textit{Ins}, \textit{Ps}) = \textit{Out} \rrbracket]$$

Blocks having positive sample time values are said to have discrete sample times. As Simulink adopts the continuous-time semantics, a discrete system behaves piecewise constantly continuously. Namely, blocks execute their functions only at sample time points, and remain constant between the sample time points. To capture this timing behavior, we decompose the time domain into a sequence of left-closed, right-open intervals, where the length is the sample time value. This feature is represented by the expression  $\{\exists k : \mathbb{N} \bullet \alpha = k * st \wedge \omega = (k + 1) * st\}$  where  $\mathbb{N}$  includes the value 0. Moreover, the update of the function is modelled by the expression  $\{\mathcal{F}(\textit{Ins}, \textit{Ps})(\alpha) = \textit{Out}\}$ . The expression restricts that all values of the output over an interval relies on the values of the inputs at the beginning of the interval. Hence, the schema structure for a discrete block can be defined as below.

$$\textit{EleBlk}_D \hat{=} [\textit{BasicBlk} \mid \textit{st} > 0 \wedge \{\exists k : \mathbb{N} \bullet \alpha = k * st \wedge \omega = (k + 1) * st\} = \{\mathcal{F}(\textit{Ins}, \textit{Ps})(\alpha) = \textit{Out}\}]$$

Every elementary block is either discrete or continuous and thus can be modelled by the following schema:  $\textit{EleBlk} \hat{=} \textit{EleBlk}_C \vee \textit{EleBlk}_D$ .

### 3.2 Construction of TIC library

Parameterization technique is the key for Simulink library blocks to create elementary blocks. As we focus on the mathematical function, the parameters for block visual appearance or simulation efficiency are deliberately ignored. For example, the parameter about a block font size is omitted. According to the effect

to the mathematical function, the remaining parameters are classified into three groups: operand parameters, sample times and operator parameters.

A library block is represented by a TIC library function. The function accepts a set of arguments which correspond to the operand parameters or sample time of the library block, and returns the TIC schema which specifies the behavior of generated elementary block. Based on the general structure of the TIC schemas of elementary blocks defined in the previous section, we model the general structure of the TIC library functions as follows.

- A *continuous* library block has the sample time value 0. The structure of its TIC function can be depicted by the function  $Lib\_C : \mathbb{P}\mathbb{R} \rightarrow \mathbb{P} EleBlk\_C$  that considers only the values of relevant operand parameters.
- A *discrete* library block has positive sample time values. The structure of its TIC library function can be depicted by the function  $Lib\_D : (\mathbb{T} \times \mathbb{P}\mathbb{R}) \rightarrow \mathbb{P} EleBlk\_D$  that takes into account both the sample time and the operand parameters of the block.
- Other library blocks can produce either discrete or continuous elementary blocks. Thus, their structure of the TIC library function can be depicted by the following function which covers both kinds of behavior with corresponding sample time constraint.

$$\left| \begin{array}{l} Lib\_I : (\mathbb{T} \times \mathbb{P}\mathbb{R}) \rightarrow \mathbb{P} EleBlk \\ \hline \forall t : \mathbb{T}; ps : \mathbb{P}\mathbb{R} \bullet (t = 0 \Rightarrow Lib\_I(t, ps) = Lib\_C(ps)) \\ \quad \wedge (t > 0 \Rightarrow Lib\_I(t, ps) = Lib\_D(t, ps)) \end{array} \right.$$

With the expressive power, TIC supports a wide range of library blocks. One advantage is that we can handle continuous library blocks. For example, the continuous library block *Integrator* adds an initial value and the integral of its input from time 0 to current time point. In the TIC library function *Integrator* shown below, the variables  $In_1$  and  $Out$  relate to the input and the output respectively, as well as the variable  $IniVal$  for the initial value. The parameter of the function denote the initial value from the Simulink when creating an elementary block. The product of the function is the TIC schema that specifies the behavior of the elementary block in terms of a higher level, i.e. intervals instead of time points. Note that the operator  $\int$  is defined in [7].

$$\left| \begin{array}{l} Integrator : \mathbb{R} \rightarrow \mathbb{P}[In_1, Out : \mathbb{T} \rightarrow \mathbb{R}; IniVal : \mathbb{R}; st : \mathbb{T}] \\ \hline \forall init : \mathbb{R} \bullet Integrator(init) = [In_1, Out : \mathbb{T} \rightarrow \mathbb{R}; IniVal : \mathbb{R}; st : \mathbb{T} | \\ \quad st = 0 \wedge IniVal = init \wedge Out(0) = IniVal \wedge \\ \quad \mathbb{I} = \llbracket Out(\omega) = Out(\alpha) + \int_{\alpha}^{\omega} In_1 \rrbracket \end{array} \right.$$

Another advantage is that the timing feature, i.e., the sample times of the elementary blocks can be preserved in the generated TIC schemas. For example, the discrete library block *Zero Order Hold* samples and holds its input for a specified sample time. As the TIC library function *ZOH* shown below, the returned schema which stores the sample time value by the variable  $st$ , and captures the discrete behavior by the TIC expression  $\{Out = In_1(\alpha)\}$ .

$$\left| \begin{array}{l} ZOH : \mathbb{T} \rightarrow \mathbb{P}[In_1, Out : \mathbb{T} \rightarrow \mathbb{R}; st : \mathbb{T}] \\ \hline \forall t : \mathbb{T} \bullet ZOH(t) = [In_1, Out : \mathbb{T} \rightarrow \mathbb{R}; st : \mathbb{T} \mid st > 0 \wedge st = t \wedge \\ \quad \{\exists k : \mathbb{N} \bullet \alpha = k * st \wedge \omega = (k + 1) * st\} = \{Out = In_1(\alpha)\}] \end{array} \right.$$

We have demonstrated how TIC library functions deal with the operand parameters and the sample times. Next we consider the analysis of the operator parameters. This kind of parameters is special since it allows a library block to generate elementary blocks with different functionalities. One example is the library block *Sum* which adds two inputs by default. While it can produce elementary blocks that can either execute a subtraction or an addition of three inputs according to the value of its parameter *Inputs*. To cope with the complexity, we adopt the overloading technique. Namely, we define multiple TIC library functions for a single type of library block, and each function models one kind of functionality of the library block. Regarding the previous example, three TIC library functions capture three different functions of the library block *Sum* respectively.

In this section, we showed the general structure of TIC schemas for Simulink elementary blocks and TIC library functions for Simulink library blocks. The arguments of a library function correspond to the operand parameters or the sample time of a library block. Overloading technique is applied to handle the operator parameters. The TIC library serves as a foundation for automatic translation. Currently, we formally defined 50 TIC library functions<sup>3</sup> for 25 often used library blocks from 8 categories including *continuous*, *discrete* and *discontinuities* libraries. We also identified that the library blocks from the *Ports and Subsystems* category are not suitable to be specified in this phase. They are usually used to construct subsystems. Hence, their functionalities are unpredictable until they are instantiated in specific Simulink diagrams. We will discuss their solution in the next section.

## 4 Translating Simulink Diagrams to TIC Specifications

A Simulink diagram represents a set of mathematical functions over time. In this section we will show how the translation from Simulink diagrams to TIC specifications can preserve the functional and timing aspects. As Simulink models systems in a hierarchical way, we illustrate the translation in the bottom-up order, namely, from elementary blocks, wires to diagrams. A discussion for handling conditionally executed subsystems is provided in the end.

### 4.1 Translating Elementary Blocks

A Simulink elementary block denotes a primitive mathematical function in a diagram. It is produced by a library block using the parameterization technique.

<sup>3</sup> They can be found at [www.comp.nus.edu.sg/~chenchun/TIC\\_Lib/](http://www.comp.nus.edu.sg/~chenchun/TIC_Lib/) and Appendix A shows part of them used for the water tank system.

Similarly, an elementary block is translated into a TIC schema by applying an appropriate TIC library function to relevant Simulink parameters. Two important factors are taken into account in the translation.

One is the criteria to choose a suitable TIC library function that produces a TTIC schema to correctly model the functional behavior of the elementary block. The primary criterion is the parameter *BlockType* which indicates the mathematical function implicitly. Recalling the *integration* example given in Section 2.1, the value *Integrator* of the parameter *BlockType* denotes that the block performs an integration function. Furthermore, some library blocks can generate different functionalities of their instances by different values assigned of their operator parameters. Thus, these relevant operator parameters are additional criteria as well. Taking the library block *Sum* example from 3.2, parameters *BlockType* and *Inputs* compose the criteria to select an appropriate TIC library function.

The other is the sample time which represents the rate at which a block is executed. A block sample time can be assigned explicitly by the parameter *SampleTime* with positive value; or implied by the type of its library block, for example, a continuous block always has sample time value 0; or derived from sample time propagation, which is a process to calculate the sample time of a block from the sample times of the inputs of the block. A method is developed below to derive the sample time of an elementary block based on the instructions in [18]. We assume that the sample time values of the block inputs (each is a function of time and denoted by  $Blk\_In == \mathbb{T} \rightarrow \mathbb{R}$ ) are known and represented by the function  $InST : Blk\_In \rightarrow \mathbb{T}$ .

1. If all inputs have the same sample time values, then the value is assigned to the sample time of the block. The following function returns the desired sample time value if it exists, value 0 otherwise.

$$\left| \begin{array}{l} AllEq : \mathbb{P} Blk\_In \rightarrow \mathbb{T} \\ \hline \forall ins : \mathbb{P} Blk\_In \bullet \exists res : \mathbb{T} \bullet \\ AllEq(ins) = \text{If } \forall in : ins \bullet InST(in) = res \text{ Then } res \text{ Else } 0 \end{array} \right.$$

2. Otherwise, if a sample time value of an input is the common integer divisor of other sample time values, then the value is the result. The following function specifies the computation and returns value 0 if no such a sample time exists.

$$\left| \begin{array}{l} ExiFast : \mathbb{P} Blk\_In \rightarrow \mathbb{T} \\ \hline \forall ins : \mathbb{P} Blk\_In \bullet \exists res : \mathbb{T} \bullet ExiFast(ins) = \\ \text{If } \exists in1 : ins \bullet \forall in2 : ins \mid in1 \neq in2 \bullet \\ \exists k : \mathbb{N} \mid k > 1 \bullet InST(in2) = InST(in1) * k \wedge InST(in1) = res \\ \text{Then } res \text{ Else } 0 \end{array} \right.$$

3. Otherwise, if the Simulink *variable-step* solver <sup>4</sup> is used, then the block is assigned the continuous sample time; if the Simulink *fixed-step* solver is used,

<sup>4</sup> Variable-step solvers vary the simulation step size in simulation, while fixed-step solvers keep the simulation step size constant.

and the greatest common integer divisor (GCD) of the sample time values of all inputs can be derived, then GCD is the result, otherwise, it is value 0. In the following function  $STP$ ,  $Solver == \{Fixed\_Step, Variable\_Step\}$ , and the function  $CalGCD$  returns GCD if it exists, otherwise value 0. The function  $STP$  checks whether any of the previous two conditions holds before it computes the sample time based on the solver type. Note that the previous two conditions are mutually exclusive.

$$\left| \begin{array}{l} STP : \mathbb{P} Blk\_In \times Solver \rightarrow \mathbb{T} \\ \hline \forall ins : \mathbb{P} Blk\_In; s : Solver \bullet STP(ins, s) = \\ \text{If } AllEq(ins) \neq 0 \vee ExiFast(ins) \neq 0 \text{ Then } AllEq(ins) + ExiFast(ins) \\ \text{Else (If } s = Variable\_Step \text{ Then } 0 \text{ Else } CalGCD(ins)) \end{array} \right.$$

Hence, after the translation, the functional aspect of an elementary block is captured by a TIC schema, and the timing information, i.e. the sample time value, is calculated and kept in the schema.

Taking the elementary block *less* in Figure 1 as an example, the selection criteria consists of the parameters *BlockType* and *Operator*, so their respective values *RelationalOperator* and “<” determine that the proper library function is *Relation\_l* shown in Appendix A. In addition, the sample time value is 0, which is derived by the sample time propagation method as the block *maximum* has the continuous sample time. Thus, the block is translated into the TIC schema:  $tank\_controller\_less \hat{=} Relation\_l(0)$ . We adopt a conventional naming manner to capture the hierarchical structure of the Simulink diagram. To be specific, a TIC schema name of a block is formed by appending the names of the block and systems along the structure path of the diagram using the symbol “\_”. Hence, the schema name *tank\_controller\_less* indicates that the block *less* is the component of the system *controller* which is the subsystem of the system *tank*.

## 4.2 Translating Wires

In Simulink, wires represent input and output relations between connected blocks. They have values at all points in time. Source (Destination) block writes (reads) value to (from) a wire according to its sample time. Hence, it supports the communication between blocks which have different sample time values. We convert a wire into an equation which consists of two variables denoting the output of the source block and the input of the destination block respectively. The equivalence remains the Simulink communication feature, i.e. the destination block receiving the same value that is produced by the source block at the same time. The general structure can thus be modelled in the schema.

$$Line \hat{=} [src, dst : \mathbb{T} \rightarrow \mathbb{R} \mid src = dst]$$

## 4.3 Translating Diagrams

A Simulink diagram is formed by connecting Simulink blocks with wires. Hence the formal specification of a diagram should capture the components and connection. Our method is similar to the way presented in [2]. A diagram into a TIC

schema after translating its components into corresponding TIC schemas. The schema declares each component to be a schema variable which is an instance of the TIC schema of the component. It hence depicts the connection by translating each wire into an equation using the variables in the declaration part. The structure of the schema can be modelled by the following mutually recursive free type definition [11].

$$\begin{aligned} \text{Diagram} & ::= \text{System} \langle\langle [InS, OutS : \mathbb{P}(\mathbb{T} \rightarrow \mathbb{R}); Blks : \mathbb{P}_1 \text{Block}; Ls : \mathbb{P}_1 \text{Line}] \rangle\rangle \\ & \& \\ \text{Block} & ::= \text{Subsystem} \langle\langle [subsys : \text{Diagram}] \rangle\rangle \mid \text{LibBlk} \langle\langle [blk : \text{EleBlk}] \rangle\rangle \end{aligned}$$

The above definition indicates that a Simulink block can be either an elementary block or a subsystem which is a diagram as well. Moreover, it restricts that a Simulink diagram must have at least one wire to connect two components.

#### 4.4 Additional Translation Issues

As mentioned at the end of Section 3, the library blocks from the *Ports and Subsystems* category would be analyzed during the translation. As they are mainly used to construct subsystems, we demonstrate the solution by considering the *plain* subsystems and *conditionally executed* subsystems below.

A plain subsystem reduces virtually the number of blocks displayed in a Simulink diagram without changing the system behavior. Hence, it is treated in the same way translating diagrams. In particular, the instances generated by the library blocks *Inport* and *Outport* are represented by functions from time to real numbers as they represent the interface of the subsystems.

A conditionally executed subsystem restricts its execution within special periods. Namely, the execution depends on the value of the *control signal*. In our approach, such a subsystem is translated into a TIC schema in the similar way for a plain subsystem, besides two additional TIC predicates constraining the relationship between the execution and the control signal in terms of intervals. To be specific, each predicate contains two TIC expressions. One expression (*TE1*) specifies the intervals when the system is (or is not) executable, the other expression (*TE2*) depicts the corresponding behavior. Hence, the predicate is generally in the format  $TE1 \subseteq TE2$ . As the execution of subsystems can be arbitrary, it is hard to predict the contents of the expression *TE2*, and we thus focus on the way to construct the expression *TE1* for two prominent conditionally executed subsystems respectively.

- An *enabled* subsystem is executed when the control signal is positive. The control signal is defined by the function  $enabled : \mathbb{T} \rightarrow \mathbb{R}$ . The set of intervals when the subsystem is enabled is thus expressed by the expression  $\llbracket enabled > 0 \rrbracket$ , while the expression  $\llbracket enabled \leq 0 \rrbracket$  represents the set of intervals when the subsystem is disabled.
- A *triggered* subsystem is executed when a trigger event in the control signal occurs. That is to say, it is active at single time points. The control signal

is defined by the function  $triggered : \mathbb{T} \rightarrow \mathbb{B}$  which returns true when an event occurs, and false otherwise. Thus, the set of intervals when the subsystem is active is specified by the expression  $[triggered]$ , while the expression  $(\neg triggered)$  denotes the set of intervals when the system is inactive.

We experiment our strategy by translating the water tank system displayed in Figure 1 into corresponding TIC schemas shown in Appendix B. For simplicity, we choose the subsystem *detector* which is plain as the example.

The subsystem contains four elementary blocks. Two of them, *integration* and *sampling* are translated to two TIC schemas below which capture the initial value and the sample time of the blocks respectively.

$$\begin{aligned} tank\_detector\_integration &\hat{=} Integrator(4) \\ tank\_detector\_sampling &\hat{=} ZOH(1) \end{aligned}$$

Other blocks, *in* and *out* denote the interface of the subsystem. They are declared as functions over time in the following TIC schema *tank\_detector*. The schema models the subsystem by declaring its components as instances of the translated TIC schemas, and confining the connections by three equations.

$\begin{aligned} &in : \mathbb{T} \rightarrow \mathbb{R}; \text{ sampling} : tank\_detector\_sampling \\ &integration : tank\_detector\_integration; \text{ out} : \mathbb{T} \rightarrow \mathbb{R} \end{aligned}$
$\begin{aligned} &sampling.Out = out \wedge in = integration.In_1 \\ &integration.Out = sampling.In_1 \end{aligned}$

We remark that in the subsystem *controller*, there is an algebraic loop made up by the blocks *switch* and *IC*. In practical, solving an algebraic loop is difficult, and unnecessary if it is not involved in analysis. Thus, in our approach, the structure of the loop, i.e. the components and the connections, is preserved after the translation, so the loop can be retrieved by relevant TIC schemas and equations when needed.

In this section, we presented a strategy to translate Simulink diagrams into TIC specifications in the bottom-up manner. The translation preserves the functional and timing aspect. We also discussed the solution to handle conditionally executed subsystems. Currently we have been implementing the strategy using JAVA so the translation can be accomplished automatically, for example, the TIC specifications of the water tank system are generated successfully by the translator.

## 5 Simulink Diagrams Verification in TIC

In TIC, verification is a deduction to show that a system implies requirements. This section will firstly describe the way to specify requirements based on the translated TIC specifications, and then show the benefits from utilizing the TIC verification capability by analyzing the water tank system.

## 5.1 Specification of Requirements

TIC models system behavior in terms of intervals. It supports various requirements specifications represented as TIC predicates. For example, a *safety* requirement that a predicate  $P$  holds always can be expressed in the TIC predicate  $\mathbb{I} = \llbracket P \rrbracket$  that restricts the requirement in a higher level, i.e., in any non-empty level. With the TIC-defined operators on interval endpoints and length, timing constraints that are difficult to be supported in Simulink, can be represented precisely and concisely in TIC. For example, a timing *liveness* requirement that for any interval lasting more than  $K$  time units and during which a predicate  $P$  holds, then a predicate  $Q$  should hold within  $K$  time units and last till the end of the interval, can be specified by the following TIC predicate  $\llbracket P \wedge \delta > K \rrbracket \subseteq \llbracket \delta \leq K \rrbracket \curvearrowright \llbracket Q \rrbracket$ .

In our approach, requirements are formed based on the TIC schemas translated from Simulink diagrams. We can specify them over the whole system or some of its components. For example, in the water tank system, the requirement of the subsystem *detector* is “for any period lasting more than 1 second and during which the water volume in a tank is not greater than 1, then the gate must open within 1 second (including 1) till the end.”. From the translated TIC specifications shown in Appendix B, the variable *gate* of the schema *tank\_controller* denotes the gate status, and the output of the schema *tank\_detector\_sampling* represents the water volume sensed in the tank. Moreover, the requirement is about timing-related liveness, so it can be specified easily in the similar format of the previous TIC predicate. Namely, the requirement is modelled as below:

$$\forall sys : tank \bullet \{sys.detector.sampling.Out \leq 1 \wedge \delta > 1\} \subseteq \{ \delta \leq 1 \} \curvearrowright \{sys.controller.gate = 1\}$$

We remark that Simulink Verification and Validation [19] provides a function to link requirements documents (e.g. a Word or Excel file) with Simulink components. The function aims to assist users to quickly look over requirements in the modelling phase, and it is different from ours in that we can formally verify systems against requirements directly.

## 5.2 Checking beyond Simulink

As Simulink diagrams are constructed in a hierarchical manner, we adopt a common approach [10] to analyze system behavior in a bottom-up order. We start with checking requirements of subsystems, so the proved requirements act as lemmas for the analysis of higher-level system. During the verification, the translated TIC specifications serve as assumptions to depict the blocks behavior and the connections in the diagram. Each deductive step is reached by formally applying assumptions, reasoning rules, common mathematical theories, or lemmas. We take the verification of the requirement mentioned above as an example and give manually developed deduction outlines by necessity:

1. Start with the premise  $\{sys.detector.sampling.Out \leq 1 \wedge \delta > 1\}$ . Let  $r = 1 > 0$  and  $s = \delta - 1 > 0$ , and based on *rule 4* and *rule 3*, we can obtain a concatenation of two sets of intervals. Two sets are named by  $E1$  and  $E2$ .

$$\begin{aligned} & \{sys.detector.sampling.Out \leq 1 \wedge \delta > 1\} \\ & = \{sys.detector.sampling.Out \leq 1 \wedge \delta = 1\} && [E1] \\ & \curvearrowright \{sys.detector.sampling.Out \leq 1\} && [E2] \end{aligned}$$

2. In  $E1$ , since  $\delta = 1 \Rightarrow \delta \leq 1$ , we applied *rule 1* and *rule 2* in turn to obtain  $E1 \subseteq \{\delta = 1\} \subseteq \{\delta \leq 1\}$ .
3. As  $sampling.Out = out$  in the schema  $tank\_detector$ , and in the schema  $tank$ ,  $detector.out = controller.water$ , after two substitutions of the equations in turn, we have

$$\begin{aligned} E2 & = \{sys.detector.out \leq 1\} \\ & = \{sys.controller.water \leq 1\} \end{aligned}$$

4. A proved requirement of the *controller* subsystem is used as a lemma below:  
**Lemma** :  $\forall con : tank\_controller \bullet \llbracket con.water \leq 1 \rrbracket \subseteq \llbracket con.gate = 1 \rrbracket$ . It states that whenever the controller detects that the input water volume is not lager then 1, it opens the gate. Thus, after applying *rule 1* and *2* several times, we have:

$$E2 \subseteq \{sys.controller.gate = 1\}$$

5. From step 2 and 4, after applying *rule 3*, we can conclude that

$$\begin{aligned} & \{sys.detector.sampling.Out \leq 1 \wedge \delta > 1\} \\ & \subseteq \{\delta \leq 1\} \curvearrowright \{sys.controller.gate = 1\} \quad \square \end{aligned}$$

Besides the above proof, we have successfully shown that the water tank system satisfied other important requirements<sup>5</sup> including the safety requirement that the tank would be neither empty nor overflow always. In the verification, mathematical analysis, e.g. integral calculus, is applied freely in the TIC logic, and hence it provides a flexible interface to conventional control theory. Furthermore, one advantage is that open systems can be analyzed if certain constraints of their input functions are given. Simulink can only simulate closed systems, and it is often impractical to know the exact input functions. This limitation can be solved in our approach by specifying the constraints in TIC and treating them as assumptions in the verification.

The proof so far is achieved by hand. The calculations however are simple and stereotypic, so there is a reasonable hope for machine-assisted proof. Currently we are exploring the way to reuse the existent work [5] which formalized several reasoning rules in the generic theorem prover Isabelle [20]. Based on the real numbers and set theories available in Isabelle/HOL, intervals can be implemented as connected sets of real numbers, and TIC specifications can be encoded into Isabelle theorems to be checked by the validated reasoning rules.

<sup>5</sup> Details of other requirements verification are available in the technical report at [www.comp.nus.edu.sg/~chenchun/water\\_tank/](http://www.comp.nus.edu.sg/~chenchun/water_tank/)

## 6 Conclusion

In this paper, we propose to apply Timed Interval Calculus (TIC), a set-theoretic notation, to formally model and verify Simulink diagrams. The work is based on the same angle adopted by Simulink and TIC where they specify systems in terms of continuous time. We defined a set of TIC library functions to model Simulink library blocks and cover a wide range of categories such as *continuous*, *discrete* and *discontinuous* libraries. Moreover, the TIC schemas produced by their library functions can capture the functional behavior over time of the Simulink elementary blocks.

We presented a strategy to translate Simulink diagrams to TIC specifications in the bottom-up order. The timing information can be derived and kept in the generated TIC schemas. Hence, the translation preserves the functional and timing aspects of the diagrams. Moreover, we discussed the way to handle conditionally executed subsystems, such as *enabled* and *triggered* subsystems. A translator has been implemented in JAVA to experiment our strategy.

With the expressive power of TIC, we can precisely and concisely specify requirements, especially the timing constraints, over a system or its components after the translation. This way yields a larger design space. Using TIC reasoning rules, we can formally verify systems against requirements beyond Simulink, for example, a safety requirement needs a possible infinite simulation period. During the verification, mathematical analysis, e.g., control theory, can be applied freely in TIC logic. Furthermore, open systems which are not checkable by simulation in Simulink can be analyzed in our approach. Thus, using TIC can elevate the design quality in Simulink.

We are enhancing the capability of the translator with more complex Simulink diagrams. In the future, we plan to extend the TIC library functions to support Stateflow. Embedding TIC into Isabelle/HOL for machine-assisted proof is one of our goals as well.

## 7 Acknowledgements

We thank Brendan Mahony for providing materials about TIC notation. We also thank Anders P. Ravn and Zhou Chaochen and Mark Adams for their insightful discussion on the related work. This work is supported by the A\*Star research Grants “Formal Design Techniques for Reactive Embedded Systems”.

## References

1. M. M. Adams and P. B. Clayton. Clawz: Cost-effective formal verification for control systems. In *ICFEM 2005*, pages 465–479, 2005.
2. R. D. Arthan, P. Caseley, C. O’Halloran, and A. Smith. ClawZ: Control laws in Z. In *ICFEM 2000*, pages 169–176. IEEE Press, 2000.
3. P. Caspi, A. Curic, A. Maignan, C. Sofronis, and S. Tripakis. Translating discrete-time Simulink to Lustre. In *EMSOFT 2003*, Philadelphia, PA, USA, 2003.

4. A. Cavalcanti, P. Clayton, and C. O'Halloran. Control law diagrams in Circus. In *FM 2005*, University of Newcastle upon Tyne, UK, July 2005.
5. J. E. Dawson and R. Goré. Machine-checking the timed interval calculus. In *Australian Joint Conference on Artificial Intelligence*, pages 95–106, 2002.
6. C. J. Fidge. Modelling discrete behaviour in a continuous-time formalism. In *IFM 1999, York, UK*, pages 170–188. Springer-Verlag, June 1999.
7. C. J. Fidge, I. J. Hayes, and B. P. Mahony. Defining differentiation and integration in Z. In *ICFEM 1998*. IEEE Computer Society, 1998.
8. C. J. Fidge, I. J. Hayes, A. P. Martin, and A. Wabenhorst. A set-theoretic model for real-time specification and reasoning. In *MPC 1998*, pages 188–206, 1998.
9. S. Gupta, B. H. Krogh, and R. A. Rutenbar. Towards formal verification of analog designs. In *ICCAD 2004*, pages 210 – 217, 2004.
10. J. Hooman. *Specification and Compositional Verification of Real-Time Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1991.
11. ISO/IEC. *Information Technology — Z Formal Specification Notation — Syntax, Type System and Semantics*, 1st edition, July 2002. 13568.
12. A. Jantsch and I. Sander. Models of computation and languages for embedded system design. *IEE Pro. on Comp. and Dig. Tech.*, 152(2):114–129, March 2005.
13. M. Jersak, Y. Cai, D. Ziegenbein, and R. Ernst. A transformational approach to constraint relaxation of a time-driven simulation model. In *the 13th international symposium on System synthesis*, pages 137–142. IEEE Computer Society, 2000.
14. B. P. Mahony and I. J. Hayes. A case-study in timed refinement: A mine pump. *IEEE Transactions on Software Engineering*, 18(9):817–826, 1992.
15. A. Martin and C. J. Fidge. Lifting in Z. *Electronic Notes in Theoretical Computer Science*, 42, 2001.
16. The MathWorks. *Stateflow and Stateflow coder - For Complex Logic and State Diagram Modeling*, 2003.
17. The MathWorks. *Simulink - Simulation and Model-based Design - Simulink Reference Version 6*, 2004.
18. The MathWorks. *Simulink - Simulation and Model-based Design - Using Simulink Version 6*, 2004.
19. The MathWorks. *Simulink Verification and Validation User's Guide*, March 2006.
20. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Springer, 2002.
21. A. Pnueli. Embedded systems: Challenges in specification and verification. In *EMSOFT 2002*, pages 1–14, London, UK, 2002. Springer-Verlag.
22. S. Sims, R. Cleaveland, K. Butts, and S. Ranville. Automated validation of software models. In *ASE 2001*, pages 91–96. IEEE Computer Society, 2001.
23. A. Tiwari, N. Shankar, and J. Rushby. Invisible formal methods for embedded control systems. *Proceedings of the IEEE*, 91(1):29–39, Jan. 2003.
24. A. Wabenhorst. Induction in the timed interval calculus. *Theoretical Computer Science*, 300(1-3):181–207, 2003.
25. F. Wang. Formal verification of timed systems: A survey and perspective. *Proceedings of the IEEE*, 92(8):1283–1307, August 2004.
26. J. Woodcock and J. Davies. *Using Z: Specification, Refinement and Proof*. Prentice Hall International, 1996.
27. C. C. Zhao and X. S. Li. A mean value calculus of durations. In *A classical mind: essays in honour of C. A. R. Hoare*, pages 431–451. Prentice-Hall International, 1994.
28. C. C. Zhou and M. R. Hansen. *Duration Calculus: A Formal Approach to Real-Time Systems*. Springer Verlag, 2004.
29. C. C. Zhou, A. P. Ravn, and M. R. Hansen. An extended duration calculus for hybrid real-time systems. In *Hybrid Systems*, pages 36–59. Springer-Verlag, 1993.

## A TIC library functions of the water tank system

$ZOH : \mathbb{T} \rightarrow \mathbb{P}[In_1, Out : \mathbb{T} \rightarrow \mathbb{R}; st : \mathbb{T}]$
$\forall t : \mathbb{T} \bullet ZOH(t) = [In_1, Out : \mathbb{T} \rightarrow \mathbb{R}; st : \mathbb{T} \mid$ $st > 0 \wedge st = t \wedge \{Out = In_1(\alpha)\} =$ $\{\exists k : \mathbb{N} \bullet \alpha = k * st \wedge \omega = (k + 1) * st\}]$
$Integrator : \mathbb{R} \rightarrow$ $\mathbb{P}[In_1, Out : \mathbb{T} \rightarrow \mathbb{R}; IniVal : \mathbb{R}; st : \mathbb{T}]$
$\forall init : \mathbb{R} \bullet Integrator(init) =$ $[In_1, Out : \mathbb{T} \rightarrow \mathbb{R}; IniVal : \mathbb{R}; st : \mathbb{T} \mid st = 0 \wedge$ $init = IniVal \wedge Out(0) = IniVal \wedge$ $\mathbb{I} = \llbracket Out(\omega) = Out(\alpha) + \int_{\alpha}^{\omega} In_1 \rrbracket]$
$Switch : (\mathbb{T} \times \mathbb{R}) \rightarrow \mathbb{P}[TH : \mathbb{R}; st : \mathbb{T};$ $In_1, In_2, In_3, Out : \mathbb{T} \rightarrow \mathbb{R}]$
$\forall t : \mathbb{T}; th : \mathbb{R} \bullet$ $(t = 0 \Rightarrow Switch(t, th) = [TH : \mathbb{R}; st : \mathbb{T};$ $In_1, In_2, In_3, Out : \mathbb{T} \rightarrow \mathbb{R} \mid$ $st = 0 \wedge th = TH \wedge$ $\llbracket In_2 > TH \rrbracket = \llbracket Out = In_1 \rrbracket \wedge$ $\llbracket In_2 \leq TH \rrbracket = \llbracket Out = In_3 \rrbracket]) \wedge$ $(t > 0 \Rightarrow Switch(t, th) = [TH : \mathbb{R}; st : \mathbb{T};$ $In_1, In_2, In_3, Out : \mathbb{T} \rightarrow \mathbb{R} \mid$ $t = st \wedge st > 0 \wedge th = TH \wedge$ $\{\exists k : \mathbb{N} \bullet \alpha = k * st \wedge \omega = (k + 1) * st\} =$ $\{(In_2(\alpha) > TH \Rightarrow Out = In_1(\alpha)) \wedge$ $(In_2(\alpha) \leq TH \Rightarrow Out = In_3(\alpha))\})$
$Sum\_PM : \mathbb{T} \rightarrow$ $\mathbb{P}[In_1, In_2, Out : \mathbb{T} \rightarrow \mathbb{R}; st : \mathbb{T}]$
$\forall t : \mathbb{T} \bullet (t = 0 \Rightarrow Sum\_PM(t) =$ $[In_1, In_2, Out : \mathbb{T} \rightarrow \mathbb{R}; st : \mathbb{T} \mid$ $st = 0 \wedge \mathbb{I} = \llbracket Out = In_1 - In_2 \rrbracket]) \wedge$ $(t > 0 \Rightarrow Sum\_PM(t) =$ $[In_1, In_2, Out : \mathbb{T} \rightarrow \mathbb{R}; st : \mathbb{T} \mid$ $t = st \wedge st > 0 \wedge$ $\{Out = In_1(\alpha) - In_2(\alpha)\} =$ $\{\exists k : \mathbb{N} \bullet \alpha = k * st \wedge \omega = (k + 1) * st\})$

$Relation\_l : \mathbb{T} \rightarrow \mathbb{P}[In_1, In_2 : \mathbb{T} \rightarrow \mathbb{R};$ $Out : \mathbb{T} \rightarrow \{0, 1\}; st : \mathbb{T}]$
$\forall t : \mathbb{T} \bullet$ $(t = 0 \Rightarrow Relation\_l(t) = [In_1, In_2 : \mathbb{T} \rightarrow \mathbb{R};$ $Out : \mathbb{T} \rightarrow \{0, 1\}; st : \mathbb{T} \mid st = 0 \wedge$ $\llbracket In_1 < In_2 \rrbracket = \llbracket Out = 1 \rrbracket \wedge$ $\llbracket In_1 \geq In_2 \rrbracket = \llbracket Out = 0 \rrbracket]) \wedge$ $(t > 0 \Rightarrow Relation\_l(t) = [In_1, In_2 : \mathbb{T} \rightarrow \mathbb{R};$ $Out : \mathbb{T} \rightarrow \{0, 1\}; st : \mathbb{T} \mid$ $t = st \wedge st > 0 \wedge$ $\{\exists k : \mathbb{N} \bullet \alpha = k * st \wedge \omega = (k + 1) * st\} =$ $\{(In_1(\alpha) < In_2(\alpha) \Rightarrow Out = 1) \wedge$ $(In_1(\alpha) \geq In_2(\alpha) \Rightarrow Out = 0)\})$
$Relation\_g : \mathbb{T} \rightarrow \mathbb{P}[In_1, In_2 : \mathbb{T} \rightarrow \mathbb{R};$ $Out : \mathbb{T} \rightarrow \{0, 1\}; st : \mathbb{T}]$
$\forall t : \mathbb{T} \bullet$ $(t = 0 \Rightarrow Relation\_g(t) = [In_1, In_2 : \mathbb{T} \rightarrow \mathbb{R};$ $Out : \mathbb{T} \rightarrow \{0, 1\}; st : \mathbb{T} \mid st = 0 \wedge$ $\llbracket In_1 > In_2 \rrbracket = \llbracket Out = 1 \rrbracket \wedge$ $\llbracket In_1 \leq In_2 \rrbracket = \llbracket Out = 0 \rrbracket]) \wedge$ $(t > 0 \Rightarrow Relation\_g(t) = [In_1, In_2 : \mathbb{T} \rightarrow \mathbb{R};$ $Out : \mathbb{T} \rightarrow \{0, 1\}; st : \mathbb{T} \mid$ $t = st \wedge st > 0 \wedge$ $\{\exists k : \mathbb{N} \bullet \alpha = k * st \wedge \omega = (k + 1) * st\} =$ $\{(In_1(\alpha) > In_2(\alpha) \Rightarrow Out = 1) \wedge$ $(In_1(\alpha) \leq In_2(\alpha) \Rightarrow Out = 0)\})$
$InitCond : (\mathbb{T} \times \mathbb{R}) \rightarrow$ $\mathbb{P}[In_1, Out : \mathbb{T} \rightarrow \mathbb{R}; IniVal : \mathbb{R}; st : \mathbb{T}]$
$\forall t : \mathbb{T}; init : \mathbb{R} \bullet$ $(t = 0 \Rightarrow InitCond(t, init) =$ $[In_1, Out : \mathbb{T} \rightarrow \mathbb{R}; IniVal : \mathbb{R}; st : \mathbb{T} \mid$ $init = IniVal \wedge Out(0) = IniVal \wedge$ $st = 0 \wedge \llbracket 0 < \alpha \rrbracket = \llbracket Out = In_1 \rrbracket]) \wedge$ $(t > 0 \Rightarrow InitCond(t, init) =$ $[In_1, Out : \mathbb{T} \rightarrow \mathbb{R}; IniVal : \mathbb{R}; st : \mathbb{T} \mid$ $t = st \wedge Out(0) = IniVal \wedge$ $st > 0 \wedge init = IniVal \wedge$ $\{\alpha = 0 \wedge \omega = st\} = \{Out = IniVal\} \wedge$ $\{Out = In_1(\alpha)\} =$ $\{\exists k : \mathbb{N}_1 \bullet \alpha = k * st \wedge \omega = (k + 1) * st\})$

$Constant : \mathbb{R} \rightarrow \mathbb{P}[Out : \mathbb{T} \rightarrow \mathbb{R}; IniVal : \mathbb{R}]$ $\forall cv : \mathbb{R} \bullet Constant(cv) =$ $[Out : \mathbb{T} \rightarrow \mathbb{R}; IniVal : \mathbb{R}  $ $cv = IniVal \wedge \mathbb{I} = \llbracket Out = IniVal \rrbracket]$
--

$tank\_detector$ $in : \mathbb{T} \rightarrow \mathbb{R}$ $integration : tank\_detector\_integration$ $sampling : tank\_detector\_sampling$ $out : \mathbb{T} \rightarrow \mathbb{R}$
$sampling.Out = out \wedge in = integration.In_1$ $integration.Out = sampling.In_1$

## B TIC specifications of the water tank Simulink diagram

### Controller Subsystem:

$tank\_controller\_maximum \hat{=} Constant(3)$   
 $tank\_controller\_less \hat{=} Relation\_l(0)$   
 $tank\_controller\_minimum \hat{=} Constant(1)$   
 $tank\_controller\_greater \hat{=} Relation\_g(0)$   
 $tank\_controller\_inverse \hat{=} Sum\_PM(0)$   
 $tank\_controller\_switch \hat{=} Switch(0,0)$   
 $tank\_controller\_IC \hat{=} InitCond(0,0)$   
 $tank\_controller\_constant \hat{=} Constant(1)$

$tank\_controller$ $water : \mathbb{T} \rightarrow \mathbb{R}$ $maximum : tank\_controller\_maximum$ $minimum : tank\_controller\_minimum$ $less : tank\_controller\_less$ $greater : tank\_controller\_greater$ $constant : tank\_controller\_constant$ $inverse : tank\_controller\_inverse$ $switch : tank\_controller\_switch$ $IC : tank\_controller\_IC$ $gate : \mathbb{T} \rightarrow \mathbb{R}$
$water = less.In_1 \wedge inverse.Out = switch.In_3$ $constant.Out = inverse.In_1 \wedge IC.Out = gate$ $less.Out = switch.In_1 \wedge water = greater.In_1$ $switch.Out = IC.In_1 \wedge IC.Out = switch.In_2$ $greater.Out = inverse.In_2$ $maximum.Out = less.In_2$ $minimum.Out = greater.In_2$

### Plant Subsystem:

$tank\_plant\_switch \hat{=} Switch(0,0)$   
 $tank\_plant\_Open \hat{=} Constant(1)$   
 $tank\_plant\_Close \hat{=} Constant(-1)$

$tank\_plant$ $gate : \mathbb{T} \rightarrow \mathbb{R}$ $switch : tank\_plant\_switch$ $Open : tank\_plant\_Open$ $Close : tank\_plant\_Close$ $flow : \mathbb{T} \rightarrow \mathbb{R}$
$Open.Out = switch.In_1 \wedge gate = switch.In_2$ $Close.Out = switch.In_3 \wedge switch.Out = flow$

### Water Tank System:

$tank$ $plant : tank\_plant$ $detector : tank\_detector$ $controller : tank\_controller$
$plant.flow = detector.in$ $detector.out = controller.water$ $controller.gate = plant.gate$

### Detector Subsystem:

$tank\_detector\_integration \hat{=} Integrator(4)$   
 $tank\_detector\_sampling \hat{=} ZOH(1)$