5-2009

A formal framework for modeling and validating Simulink diagrams

Chunqing CHEN

Jin Song DONG

Jun SUN
Singapore Management University, junsun@smu.edu.sg

DOI 10.1007/s00165-009-0108-9
BCS © 2009
Formal Aspects of Computing (2009) 21: 451–483

Formal Aspects of Computing

A formal framework for modeling and validating Simulink diagrams

Chunqing Chen, Jin Song Dong, Jun Sun

Computer Science, School of Computing, National University of Singapore, 21 Lower Kent Ridge Road, Singapore 119077, Singapore. E-mail: chenchun@comp.nus.edu.sg; dongjs@comp.nus.edu.sg; sunj@comp.nus.edu.sg

Abstract. Simulink has been widely used in industry to model and simulate embedded systems. With the increasing usage of embedded systems in real-time safety-critical situations, Simulink becomes deficient to analyze (timing) requirements with high-level assurance. In this article, we apply Timed Interval Calculus (TIC), a real-time specification language, to complement Simulink with TIC formal verification capability. We elaborately construct TIC library functions to model Simulink library blocks which are used to compose Simulink diagrams. Next, Simulink diagrams are automatically transformed into TIC models which preserve functional and timing aspects. Important requirements such as timing bounded liveness can be precisely specified in TIC for whole diagrams or some components. Lastly, validation of TIC models can be rigorously conducted with a high degree of automation using a generic theorem prover. Our framework can enlarge the design space by representing environment properties to open systems, and handle complex diagrams as the analysis of continuous and discrete behavior is supported.

Keywords: Simulink, Real-Time Specification, Z Language, Formal Verification

1. Introduction

Simulink [Mat08b] is a graphical environment used widely to model and simulate embedded systems. A Simulink diagram is formed by connecting blocks with wires, and represents a set of mathematical relationships which model system behavior over time. Simulink adopts continuous-time semantics [JS05] to support dynamic systems such as hybrid control systems. Its simulation facility allows system behavior to be visually observed for specific parameter values over specific simulation periods. However, simulations are deficient in checking system behavior for infinite parameter values or over infinite simulation periods. In addition, open systems whose exact input functions are usually unknown are unanalyzable in Simulink because simulations are inapplicable to these systems. Moreover, Simulink lacks timing analysis which becomes necessary due to an increasing usage of embedded systems in real-time safety-critical situations [Pnu02].

Recently, formal methods have received more attention for improving the development of embedded real-time systems by their rigorous semantics and formal verification capability [Wan04, HS06]. In this article, we apply a real-time specification notation, Timed Interval Calculus (TIC) [MH92, FHMW98], to complement Simulink: functional and timing aspects of Simulink diagrams are formally captured in TIC; important (timing)

Correspondence and offprint requests to: C. Chen, E-mail: chenchun@comp.nus.edu.sg

requirements are rigorously validated by well-defined TIC reasoning rules and the strong support of mathematical analysis in TIC. The significant and novel point of our approach is the use of TIC modeling and reasoning features to support validation beyond Simulink.

The first contribution is that we formally model Simulink diagrams in TIC. Existing work [ACOS00, AC05, CCO05, TSCC05, MBR06] that interprets Simulink diagrams in other formal notations or programming languages usually focuses on discrete behavior. TIC is different from those notations and languages in that it is based on a continuous-time domain [MH92], and can concisely express properties at an interval level [FHMW98]. Furthermore, TIC supports calculus (for example, integral calculus [FHM98]) which is often used in control engineering. To the best of our knowledge, our work is the first attempt to model Simulink diagrams in terms of continuous time. Currently we support a wide range of Simulink library blocks which are frequently used to compose Simulink diagrams, specifically, 51 library blocks of 10 categories covering *continuous*, *discontinuous*, *signal routing* and so on. By elaborately constructing and rigorously validating TIC library functions which model these library blocks, we have discovered incomplete semantics and a bug in the original description [Mat08a] of these library blocks.

The second contribution is that we develop a strategy to automatically transform Simulink diagrams into TIC models. The transformation preserves functional and timing aspects of Simulink elementary blocks and the hierarchical structure of diagrams. Moreover, it can deal with unspecified sample times in (complex) diagrams. We also consider two important types of *conditionally executed subsystems* whose execution depends on their *control inputs*, particularly, *enabled* subsystems and *triggered* subsystems. Their conditional behavior is captured by carefully modeling the way of assigning values to their inputs under different circumstances, such as an enabled subsystem with a discrete control input and continuous inputs.

The third contribution is that we extend our previous work [CDS08], which developed a generic verification system based on Prototype Verification System (PVS) [ORS92] for TIC, to automatically translate axiomatic definitions [WD96] which are the type of the TIC library functions into PVS functions. Furthermore, we define supplementary rules dedicated to Simulink modeling features. With the extended verification system, we can rigorously validate Simulink diagrams against important requirements beyond Simulink with a high degree of automation, for example, checking bounded timing response requirements.

In summary, we aim to construct a formal framework to model and validate Simulink diagrams. We take a step-by-step approach. Firstly, a set of TIC library functions are defined to model Simulink library blocks in terms of intervals. Unfortunately, library blocks are informally and sometimes partially described in their original documentation [Mat08a]. We hence focus on their denotational semantics, the time-dependent mathematical relationships between their inputs and outputs. Secondly, a translator has been implemented in Java to automatically transform Simulink diagrams into TIC models in a bottom-up order. Last but not least, we specify important requirements based on the transformed TIC models which represent whole diagrams or some components. Moreover, properties of open systems, such as the bounds of an environment variable, can be precisely specified using TIC. We can rigorously verify whether the TIC models of Simulink diagrams fulfill the TIC models of requirements, by applying TIC reasoning rules, mathematical laws and common proof methods.

1.1. Related works

Recently, there have been a number of works on transforming Simulink into other formal notations or programming languages. Arthan et al. [ACOS00], Adams and Clayton [AC05] transformed Simulink diagrams to the Z notation [WD96] by capturing the functional behavior of one cycle. Cavalcanti et al. [CCO05] extended that work by applying Circus [CSW03] to model the functionality and concurrency of diagrams. Their approaches aim to verify that Simulink diagrams are correctly implemented in the programming language Ada, which is different from ours. We aim to validate that Simulink diagrams fulfill various requirements. Moreover, they consider only single-rate discrete systems, and timing information is missing. In contrast, we can handle multi-rate discrete and hybrid systems, and the timing information is retained as well.[1]

Meenakshi et al. [MBR06] used the model checker *NuSMV* to analyze single-rate discrete Simulink diagrams. Tripakis et al. [TSCC05] applied the synchronous programming language *Lustre* to support multi-rate discrete Simulink diagrams. Tiwari et al. [TSR03] converted differential equations denoted in Simulink to difference

[1] Initially, we tried to apply TCOZ [MD98, MD00] to Simulink. The attempt was abandoned as TCOZ faced the drawback of lacking support of the continuous-time semantics which is used by Simulink.

equations for constructing discrete transition systems. However, discretization of infinite transition systems can decrease accuracy when checking properties of continuous dynamics [MCDB03]. Our approach is different in that we can directly represent and analyze continuous Simulink diagrams.

There are also approaches [SCBR01, GKR04] which take into account Simulink/Stateflow[2] Models (SSMs). Sims et al. [SCBR01] verified SSMs using an invariant checker, and the transformation from SSMs to the input language of the checker was performed by hand. Gupta et al. [GKR04] developed a tool which increased Simulink modeling capability by defining some customized Simulink. That tool was designed to check functional behavior, and it lacked support of timing analysis. Jersak et al. [JCZE00] transformed Simulink diagrams into the SPI models for timing analysis, while the transformation abstracted functional behavior. On the other hand, we support the validation for both functional and timing behavior.

There are other formal notations of real-time systems. One is Duration Calculus (DC) [ZHR91] which accumulates Boolean-valued states over closed intervals to specify critical duration constraints. One of its extensions, Mean Value Calculus [ZL94] adopts the mean value of states to express properties at point intervals, and the other is Extended Duration Calculus [ZRH93] which defines two functions to represent the state values at interval endpoints. Since DC and its extensions model system behavior in terms of intervals only, they are limited to specify the constraints which are relevant to specific values of interval endpoints. For example, the behavior of the Simulink library block *Unit Delay* as presented in Sect. 3.2 relies on particular interval endpoints.

1.2. Structure

This article is a revised and extended version of our preliminary work [CD06, CDS07b], which has investigated the feasibility and benefit of applying TIC to complement Simulink with machine-assisted proof support. Besides the presentation of the TIC library functions and transformation has been improved significantly, the article extends [CD06] in several aspects: Sect. 3.3 discusses the way of constructing and validating the TIC library functions with the demonstration of discoveries; Sect. 4.4 presents an algorithm to compute unspecified sample times in Simulink diagrams; Sect. 4.5 illustrates more deeply the way to analyze conditionally executed subsystems in different conditions. Supporting more Simulink library blocks (from 32 to 51) using TIC enables our framework to handle more complex systems. Moreover, the article improves [CDS07b] by automatically translating the axiomatic definitions which are the type of the TIC library functions to PVS functions in Sect. 5.1. Note that our previous work [CDS08] has not considered the axiomatic definitions. We also apply our framework to new complex Simulink diagrams.

The remainder of this article is organized as follows: Sect. 2 briefly introduces the relevant features of Simulink, TIC, and PVS. Section 3 defines a set of TIC library functions to model Simulink library blocks. Section 4 presents a systematic way of transforming Simulink diagrams to TIC models. Section 5 describes how to enhance the verification system to ease the validation of Simulink diagrams. Section 6 illustrates an application of our framework with a hybrid control system. Section 7 concludes the article.

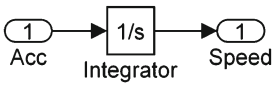
2. Background

In this section, we introduce Simulink [Mat08b], Timed Interval Calculus (TIC) [FHMW98], and Prototype Verification System (PVS) [ORS92]. We highlight their relevant features which are used in this article, and readers who are interested to know more may refer to their respective references.

2.1. Simulink

A Simulink diagram made up of blocks and wires represents a set of time-dependent mathematical relationships which model a dynamic system. A block can be an *elementary block*, which is the basic structure unit of Simulink diagrams and denotes a primitive mathematical relationship between its inputs and outputs, such as outputting the sum of two inputs. An elementary block is created by assigning specific values to the parameters of a Simulink *library block*. This parameterization technique enables a library block to create elementary blocks with

[2] Stateflow combines flow diagrams and statecharts to specify control logic and can be integrated with Simulink.




```
1  System{ Name             "calculation"
2    Block{ BlockType       Inport
3           Name            "Acc"       }
4    Block{ BlockType       Integrator
5           Name            "Integrator"
6           InitialCondition "4"        }
7    Line{ SrcBlock         "Integrator"
8           DstBlock        "Speed"   }
9    Line{ SrcBlock         "Acc"
10          DstBlock        "Integrator" }}
```

Fig. 1. A system *calculation* in Simulink graphical and textual formats

different functionalities. A block can also be a diagram composed of other sub-diagrams and elementary blocks. A *wire* is a directed edge which indicates a dependent relationship between two connected blocks; the input of the destination block depends on the output of the source block.

Every elementary block in a diagram is considered to have a *sample time* as its execution rate. A sample time of an elementary block can be explicitly specified via the *SampleTime* block parameter, determined by the block type (for example, elementary blocks generated by the *Integrator* library block have continuous sample time), or derived from the blocks which connect to the block inputs. Simulink adopts continuous-time semantics, and discrete systems are treated to be a special case of continuous systems: their behavior is piecewise constantly continuous. Moreover, Simulink supports *conditionally executed subsystems* whose behavior depends on their control input values instead of time. For example, an *enabled* subsystem is active when its control input is positive.

A Simulink diagram is represented textually in a model file [Mat08a], which denotes diagrams by *keywords* and *parameter-value pairs*. Parameter-value pairs describe the contents of diagrams such as block sample times by associating particular values with relevant parameters. Keywords followed by a pair of brackets models components at the same hierarchical layer of diagrams. Taking Fig. 1 as an example, the left part graphically depicts a simple system which outputs speed as the integration of the acceleration from input port *Acc* to output port *Speed*, and the right part shows the corresponding textual representation. Note that in the context (from lines 4 to 6) of elementary block *Integrator*, its mathematical function *integration* is not explicitly specified but indicated by the value of the *BlockType* block parameter. Moreover, the initial value 4 of *Integrator* is not visually available in the diagram. In other words, model files contain all information of systems modeled in Simulink, and they are the source of our approach which captures the (mathematical) functional and timing (namely, sample times) aspects and the structure of Simulink diagrams.

2.2. Timed interval calculus

Timed Interval Calculus (TIC) is based on the set theory and reuses the Z notation [WD96]. It adopts total functions of continuous time to model system behavior [MH92], and defines *interval brackets* for concisely expressing properties in terms of intervals [FHMW98]. Interval endpoints can be explicitly accessed. In the following, we present the essential modeling features of TIC and special symbols used in this article.

The time domain $\mathbb{T}$ is the set of all non-negative real numbers. An interval is a range of continuous time points. Intervals are classified into four basic types based on the inclusion of interval endpoints. For example, the operator $[\_ \ldots \_)$ defined below denotes a *left-closed*, *right-open* interval, where the expression $\mathbb{P}\,\mathbb{T}$ denotes a set of time points. Other three operators, $(\_ \ldots \_)$, $(\_ \ldots \_]$, and $[\_ \ldots \_]$ for *left and right-open* intervals, *left-open*, *right-closed* intervals, and *left and right-closed* intervals are defined similarly.

$$[\_ \ldots \_) : \mathbb{T} \times \mathbb{T} \to \mathbb{P}\,\mathbb{T}$$

$$\forall x, y : \mathbb{R} \mid x < y \bullet [x \ldots y) = \{z : \mathbb{T} \mid x \leq z < y\}$$

There are three types of elements which compose TIC specifications as shown below.

- *Constants*. For example, a maximum water volume $MaxV$ is a real number, namely, $MaxV : \mathbb{R}$.
- *Timed traces*. They represent variables which are functions of time. Specifically, a timed trace is a *total* function from the time domain to the type of a variable, and the type can be either continuous or discrete. For instance, water volume in a tank can be modeled by a timed trace V with real numbers $\mathbb{R}$ ($V : \mathbb{T} \to \mathbb{R}$), while an alarm signal can be depicted by a timed trace *alarm* whose range consists of two values, 0 and 1 ($alarm : \mathbb{T} \to \{0, 1\}$).

- *Interval operators*. These are functions of intervals. In TIC, α, ω, and δ are the primitive interval operators, and they return the starting point, ending point, and length of a given interval respectively.

A key construction of TIC is interval brackets. A pair of interval brackets with a predicate enclosed by the pair forms a *TIC expression* which denotes a set of intervals where the predicate holds everywhere. A predicate usually contains timed traces and interval operators, and the references to the time domain and intervals can be elided. For example, the TIC expression, $(\!| V(\alpha) \leq V |\!)$ where $(\!| \; |\!)$ is a pair of interval brackets,[3] represents a set of *left-closed*, *right-open* intervals, and in each interval the water volume V is not less than its value at the starting point of the interval. Without using $(\!| \; |\!)$, we need to explicitly associate the timed trace V and the interval operator α with their corresponding domains, as shown in the following expanded set construction of the TIC expression.

$$(\!| V(\alpha) \leq V |\!) = \{x, y : \mathbb{T} \mid \forall t : [x \ldots y) \bullet V(\alpha([x \ldots y))) \leq V(t) \bullet [x \ldots y)\}$$

Conventional set operators such as $\cup$ and $\cap$ can be applied to compose new TIC expressions. To depict sequential behavior over intervals, the concatenation operator $\frown$ is defined below to connect intervals end-to-end, where the symbol $\mathbb{I}$ represents the set of all non-empty intervals. Specifically, $\frown$ takes two sets of intervals X and Y as arguments, and returns a set of intervals each of which is composed by an interval x from the left-hand set X and an interval y from the right-hand set Y, provided (1) x occurs strictly before y and (2) x and y meet exactly, no overlap and no gap. Note that this operator is a partial function indicated by the symbol $\nrightarrow$ as the concatenation of any two sets of intervals may return the empty set.

$$\_ \frown \_ : \mathbb{P}\mathbb{I} \times \mathbb{P}\mathbb{I} \nrightarrow \mathbb{P}\mathbb{I}$$

$$\forall X, Y : \mathbb{P}\mathbb{I} \bullet X \frown Y = \{z : \mathbb{I} \mid (\exists x : X;\ y : Y \mid \omega(x) = \alpha(y) \wedge (\forall t1 : x;\ t2 : y \bullet t1 < t2) \bullet z = x \cup y)\}$$

TIC predicates model system properties and requirements at the interval level by imposing constraints on TIC expressions. For instance, the TIC predicate $[\!(V > MaxV)\!] \subseteq [\!(alarm = 1)\!]$, where the interval brackets $[\!(\;)\!]$ are the union of the four basic types of interval brackets,[4] specifies a response property that for an interval during which V exceeds $MaxV$, an alarm is on as indicated by the value 1 during that interval. To manage TIC specifications in a structural manner, the Z schema structure is adopted to define variables in the declaration part and group constraints over these variables in the predicate part. Taking the following schema *Detector* as an example, it contains the response property mentioned earlier and the variables used in the predicate. In addition, the declaration $V : \mathbb{T} \twoheadrightarrow \mathbb{R}$ captures the continuity feature that V is continuous in any non-empty interval by the symbol $\twoheadrightarrow$ from [FHM98].

Detector

$V : \mathbb{T} \twoheadrightarrow \mathbb{R};\ alarm : \mathbb{T} \to \{0, 1\};\ MaxV : \mathbb{R}$	[declaration]
$[\!(V > MaxV)\!] \subseteq [\!(alarm = 1)\!]$	[predicate]

TIC defines a collection of reasoning rules for capturing timing properties of sets of intervals and their concatenations. For example, given a predicate which is independent on interval operators, the following rule can decompose a non-point interval in which the predicate holds into two concatenated intervals, which both satisfy the predicate. Note that an implicitly necessary condition is that the time domain is continuous, as intervals over the discrete-time domain, such as an interval whose endpoints are a pair of consecutive discrete time points, cannot be decomposed.

if a predicate P is independent on α, ω, and δ, then $[\!(P \wedge \delta > 0)\!] = [\!(P)\!] \frown [\!(P)\!]$

Specifying both system designs and requirements in TIC, we can rigorously prove that designs imply requirements by deduction. In general, a proof is divided into several sub-proofs, and each sub-proof concentrates on a simple requirement of a subsystem. Each deductive step in a proof is reached by applying a hypothesis, a TIC reasoning rule, a mathematic law, or a pre-proved requirement.

In our approach, we take advantage of the expressive power of TIC to model Simulink diagrams. System designs denoted in Simulink and (timing) requirements are represented at the interval level. The formal verifica-

[3] Other three basic types of interval brackets are $(\!| \; |\!]$, $[\!| \; |\!]$, and $[\!| \; |\!]$ for *left and right-open* intervals, *left-open*, *right-closed* intervals, and *left and right-closed* intervals.

[4] $[\!(P)\!] == (\!|P|\!) \cup (\!|P|\!] \cup [\!|P|\!) \cup [\!|P|\!]$ where P is a predicate.

tion capability of TIC can increase the design confidence by rigorously validating designs against requirements, and some validations are beyond Simulink.

2.3. Prototype verification system

Prototype Verification System (PVS) is an integrated environment for formal specification and rigorous verification. An important feature of PVS is its synergistic integration of an expressive specification language and powerful theorem-proving capabilities. A recently developed NASA PVS library [But04] supports the analysis of differential and integral calculus. Previously, we developed a verification system based on PVS for TIC [CDS08]. The system encoded TIC symbols such as the interval brackets and concatenation operator in PVS, formalized and validated the TIC reasoning rules, and supported the analysis of continuous dynamics. In our framework, the verification system is used for the machine-assisted proofs of Simulink diagrams. As we emphasize our extended work (in Sect. 5), in particular representing TIC axiomatic definitions in PVS, we introduce below relevant modeling features of PVS.

The specification language of PVS is based on a classic typed, higher-order logic. Built-in types include *Boolean* (`bool`), *real numbers* (`real`), *natural numbers* (`nat`) and so on. Common logic and arithmetic operators such as conjunction (`AND`), implication (`=>`), and addition (`+`) are also defined.

Entities of PVS are introduced by means of *declarations*, which are the main constituent of PVS specifications. Declarations are used to define variables, constants, formulas, and so on. *Variable declarations* introduce new variables with their associated types. In addition, variables are local when they are defined in binding expressions which may involve keywords including `FORALL` for the universal quantifier ∀ or `LAMBDA` for the symbol λ in lambda expressions. *Constant declarations* introduce new constants which can be functions, relations or the usual (0-ary) constants. For example, the declaration `f: [nat -> nat]` defines a total function `f` (by the symbol `->`) whose domain and range are natural numbers. *Formula declarations* can introduce axioms using the keyword `AXIOM` and theorems using the keyword `LEMMA`. Axioms can be referenced by the command `lemma` during proofs. Moreover, PVS supports the *name overloading* technique which allows declaration identifiers to have the same names, even the declarations are of different types. For instance, we can declare a function `g` and an axiom `g` although constants and formulas are of different types.

In PVS, new types can be defined from the built-in types using *type constructors*. Three frequently used constructors are *subtypes*, *function types* and *record types*. For example, a record type `r` that consists of a Boolean variable `x` and a real number `y` can be specified in the expression `[# x: bool, y: real #]`. A component of a record type can be accessed by the accessor name; the x-component is accessed by `r‘x`.

PVS contains many built-in theories about logics, sets, numbers, etc. These theories support specifications and verification of various systems. For instance, the PVS set theory provides common definitions such as `emptyset` denoting ∅ and `subset?` meaning a subset relation.

3. Constructing TIC library functions for Simulink library blocks

Simulink library blocks are templates to produce elementary blocks of Simulink diagrams. These library blocks are classified into various categories: continuous, discrete, discontinuous, mathematical functions and so on. Unfortunately, their semantics is *informally*, and even partially, described in the original documentation [Mat08a]. It is thus necessary and important to formally model these blocks using TIC.

In this section, we firstly present the basic structure of TIC schemas that denote elementary blocks. The structure captures the time-dependent relationships of elementary blocks. Next, we construct TIC library functions and highlight their features with examples. Lastly, we extend our prior work [CD06] by discussing the construction of the TIC library functions and their validation. These TIC functions can accompany the original documentation [Mat08a] as they model accurate and thorough semantics of the library blocks.

3.1. TIC schemas for Simulink elementary blocks

An elementary block denotes a time-dependent mathematical relationship between its inputs and outputs. In Simulink, inputs and outputs always have values at any time point. Namely, they are total functions of time. We consider here a common type of elementary blocks which have *multiple inputs and single output*. Other types such

as multiple inputs and outputs can be handled similarly. Furthermore, each elementary block has its sample time as its execution rate.

An elementary block is modeled by a TIC schema: essential attributes including the block inputs, output, parameters and sample time are captured in the declaration part, and the block behavior is specified in terms of intervals in the predicate part. As sample times can be either continuous or discrete, we classify TIC schemas of elementary blocks in two groups according to their sample times.

When the sample time of an elementary block is equal to the value 0, the block executes continuously. Its output at a time point relies on inputs either at the same time point (for example, a summation) or through an interval (for example, an integral operation). We specify continuous behavior at the interval level rather than the time point level.

Definition 1 (Continuous basic block) A TIC schema for a continuous elementary block is a 6-tuple $(Ins, Out, Ps, st, \mathcal{F}, \mathcal{V})$, where Ins denotes a set of inputs and each input is a timed trace of the type $\mathbb{T} \to \mathbb{R}$, Out is the output which is also a timed trace, Ps denotes a set of block parameters of the type $\mathbb{R}$, st is the sample time, $\mathcal{F}$ denotes the mathematical relationship which depends on the inputs and block parameters, namely, $\mathcal{F} : Ins \times Ps \to Out$, and $\mathcal{V}$ is a mapping assigning real numbers to block parameters. These elements satisfy two constraints which capture continuous behavior of the block: one is $\mathbb{I} = [\![\mathcal{F}(Ins, Ps) = Out]\!]$ for indicating that $\mathcal{F}$ holds for all non-empty intervals; the other is $st = 0$ to limit the sample time value.

When the sample time of an elementary block is positive, the block executes discretely. To be specific, the block changes its output at *sample time hits* which are integer multiples of the sample time, and keeps its output constant between any two consecutive sample time hits. This discrete behavior is captured by modeling the behavior for each *sample time interval* which is *left-closed*, *right-open* and whose endpoints are a pair of consecutive sample time hits.

Definition 2 (Discrete basic block) A TIC schema for a discrete elementary block is also a 6-tuple $(Ins, Out, Ps, st, \mathcal{F}, \mathcal{V})$, where the types of the elements are the same as those in Definition 1. However, the constraints that these elements satisfy are different from Definition 1 as the behavior is discrete here. Specifically, $st > 0$ restricts the value of the sample time, and the other constraint specifies the discrete behavior for all sample time intervals which are denoted by $[\exists k : \mathbb{N} \bullet \alpha = k * st \wedge \omega = (k + 1) * st)$ where $\mathbb{N}$ represents the set of all natural numbers.

For many discrete elementary blocks in practice, predicates that model their behavior can be expressed in the form: $[\exists k : \mathbb{N} \bullet \alpha = k * st \wedge \omega = (k + 1) * st) \subseteq [\mathcal{F}(Ins(\alpha), Ps) = Out)$. Namely, the output Out during a sample time interval is dependent on the input(s) Ins at the starting point of the sample time interval, such as elementary blocks of the *Sum* library block with a discrete sample time as mentioned in Sect. 3.2. Nevertheless, the predicates may be in a variant form for certain types of elementary blocks, for instance, elementary blocks of the *Unit Delay* library block as illustrated in the following section.

The above two types of general structure of TIC schemas (Definitions 1 and 2) serve as a guideline to construct TIC library functions, where the mathematical relationship $\mathcal{F}$ will be explicitly specified with respect to a particular Simulink library block. Note that the range of the above timed traces is real numbers for representing the data type *double* in Simulink. This is acceptable since different data types in Simulink only affect simulation efficiency. Our approach can be extended to support multi-dimensional values. For example, vectors of values can be represented as *sequences* of values in TIC.

3.2. TIC library functions for Simulink library blocks

In Simulink, an elementary block is generated by assigning particular values to the parameters of a library block. This parameterization technique is also adopted by our TIC library functions which model Simulink library blocks. Specifically, these library functions return TIC schemas which represent elementary blocks.

As we focus on the mathematical relationships denoted by elementary blocks, irrelevant block parameters are ignored such as parameters for the graphical appearance of blocks. We divide the remainder of block parameters into three groups, *operands*, *sample times* and *operators*, according to their effects on the mathematical relationships. We present below the general structure of TIC library functions which involve the first two groups of block parameters. In the end, the way to handle the last group is given.

A continuous library block always produces continuous elementary blocks whose sample times are 0, and we thus consider only its operand parameters.

Definition 3 (Continuous library block) A TIC library function for a continuous library block takes a set of arguments which denote the operand parameters of the library block and returns a TIC schema which conforms to Definition 1 with respect to its elements and the constraints over the elements.

For example, the *Integrator* library block is a continuous library block, and its output value at the ending point of an arbitrary interval is equal to the sum of its output value at the starting point of the interval and the integration of its input over the interval. In addition, the output value at the time point 0 is stored via the *InitialCondition* block parameter, which is represented by the variable *IniVal* in the following TIC library function for the *Integrator* library block.

$$
\begin{array}{|l}
Integrator : \mathbb{R} \to \mathbb{P}[In_1 : \mathbb{T} \to \mathbb{R};\ Out : \mathbb{T} \nrightarrow \mathbb{R};\ IniVal : \mathbb{R};\ st : \mathbb{T}] \\
\hline
\forall init : \mathbb{R} \bullet Integrator(init) = [In_1 : \mathbb{T} \to \mathbb{R};\ Out : \mathbb{T} \nrightarrow \mathbb{R};\ IniVal : \mathbb{R};\ st : \mathbb{T} \mid \\
\quad st = 0 \land IniVal = init \land Out(0) = IniVal \land \mathbb{I} = [\![Out(\omega) = Out(\alpha) + \int_{\alpha}^{\omega} In_1]\!]]
\end{array}
$$

In a schema returned by the function *Integrator*, namely, *Integrator*(*init*), the predicate $IniVal = init$ which associates the argument *init* with *IniVal* corresponds to the mapping $\mathcal{V}$ in Definition 1. Moreover, the predicate $\mathbb{I} = [\![Out(\omega) = Out(\alpha) + \int_{\alpha}^{\omega} In_1]\!]$, which explicitly specifies the mathematical relationship $\mathcal{F}$ in terms of intervals, conforms to the first constraint in Definition 1. Note that we indicate the continuity feature of the block output *Out* by $\nrightarrow$.

A discrete library block always creates discrete elementary blocks whose sample times are positive, and we take into account its sample time and operand parameters.

Definition 4 (Discrete library block) A TIC library function for a discrete library block takes a set of arguments denoting the sample time and operand parameters, and returns a TIC schema which conforms to Definition 2 with respect to its elements and the constraints over the elements.

For instance, the *Unit Delay* library block is a discrete library block, and its output values during a sample time interval are equal to the input value which is sampled at the most recent sample time hit of the sample time interval. The initially output value is specified via the *X0* block parameter, and the sample time is determined from the *SampleTime* block parameter. These parameters are denoted by variables *IniVal* and *st* in the following TIC library function for the *Unit Delay* library block.

$$
\begin{array}{|l}
UnitDelay : \mathbb{T} \times \mathbb{R} \to \mathbb{P}[In_1, Out : \mathbb{T} \to \mathbb{R};\ IniVal : \mathbb{R};\ st : \mathbb{T}] \\
\hline
\forall t : \mathbb{T};\ init : \mathbb{R} \bullet UnitDelay(t, init) = [In_1, Out : \mathbb{T} \to \mathbb{R};\ IniVal : \mathbb{R};\ st : \mathbb{T} \mid \\
\quad st = t \land st > 0 \land IniVal = init \land [\![\alpha = 0 \land \omega = st)\!] \subseteq [\![Out = IniVal)\!] \land \\
\quad [\![\exists k : \mathbb{N}_1 \bullet \alpha = k * st \land \omega = (k+1) * st)\!] \subseteq [\![Out = In_1(\alpha - st))\!]]
\end{array}
$$

In a schema returned by *UnitDelay*, predicates $st = t$ and $IniVal = init$ correspond to $\mathcal{V}$ in Definition 2. Moreover, the delay behavior as the discrete behavior of an elementary block of the *Unit Delay* library block is modeled for all sample time intervals, particularly, by the last two predicates. One predicate is $[\![\alpha = 0 \land \omega = st)\!] \subseteq [\![Out = IniVal)\!]$ constraining the output values through the *initial* sample time interval which starts with the time point 0. The other predicate restricts the output values during a *non-initial* sample time interval i, where $i \in [\![\exists k : \mathbb{N}_1 \bullet \alpha = k * st \land \omega = (k+1) * st)\!]$ and $\mathbb{N}_1$ represents the set of all positive natural numbers. To be specific, the output values are equal to the input value at the last sample time hit before i, namely, $i \in [\![Out = In_1(\alpha - st))\!]$.

Other library blocks can generate either continuous or discrete elementary blocks. A TIC library function for such a library block captures both type of behavior by returning different TIC schemas according to the sample time assigned to the library block. Namely, the structure of a returned schema conforms to Definition 1 when the sample time is 0, and conforms to Definition 2 otherwise.

Taking the *Sum* library block as an example, it adds two inputs by default. If its sample time is specified by the value 0, it continuously outputs the sum. Otherwise, the addition is discrete: the output values during a sample time interval depend on the input values at the starting point of the sample time interval. The above relation between the types of behavior and the sample time is modeled by two conjunctive implications in the following TIC library function.

$$
\begin{array}{|l}
Sum\_PP : \mathbb{T} \to \mathbb{P}[In_1, In_2, Out : \mathbb{T} \to \mathbb{R};\ st : \mathbb{T}] \\
\hline
\forall t : \mathbb{T} \bullet (t = 0 \Rightarrow Sum\_PP(t) = [In_1, In_2, Out : \mathbb{T} \to \mathbb{R};\ st : \mathbb{T} \mid st = 0 \wedge \mathbb{I} = \llbracket Out = In_1 + In_2 \rrbracket]) \\
\quad \wedge\ (t > 0 \Rightarrow Sum\_PP(t) = [In_1, In_2, Out : \mathbb{T} \to \mathbb{R};\ st : \mathbb{T} \mid st = t \wedge st > 0 \wedge \\
\qquad \{\exists k : \mathbb{N} \bullet \alpha = k * st \wedge \omega = (k+1) * st\} \subseteq \{Out = In_1(\alpha) + In_2(\alpha)\}])
\end{array}
$$

So far we have presented the general structure of TIC library functions which involves the sample times and operand parameters of library blocks. Ideally, we shall establish a one-to-one mapping from block types to TIC library functions, such as the function *Integrator* for the *Integrator* library block. However, this kind of mapping is inapplicable to operator parameters, because these parameters can cause a library block to produce elementary blocks with different functionalities. We thus construct multiple TIC library functions for one library block which has operator parameters; each library function represents a particular functionality.

Reusing the *Sum* library block as an instance, its *Inputs* block parameter is an operator parameter. The default value of *Inputs* is "++" which indicates the default functionality of generated elementary blocks, a summation of two inputs (as modeled by *Sum_PP*). However, when the value of *Inputs* is "+−", the functionality becomes a subtraction of two inputs. This subtraction behavior is modeled by another TIC library function *Sum_PM* as shown below.

$$
\begin{array}{|l}
Sum\_PM : \mathbb{T} \to \mathbb{P}[In_1, In_2, Out : \mathbb{T} \to \mathbb{R};\ st : \mathbb{T}] \\
\hline
\forall t : \mathbb{T} \bullet (t = 0 \Rightarrow Sum\_PM(t) = [In_1, In_2, Out : \mathbb{T} \to \mathbb{R};\ st : \mathbb{T} \mid st = 0 \wedge \mathbb{I} = \llbracket Out = In_1 - In_2 \rrbracket]) \\
\quad \wedge\ (t > 0 \Rightarrow Sum\_PM(t) = [In_1, In_2, Out : \mathbb{T} \to \mathbb{R};\ st : \mathbb{T} \mid st = t \wedge st > 0 \wedge \\
\qquad \{\exists k : \mathbb{N} \bullet \alpha = k * st \wedge \omega = (k+1) * st\} \subseteq \{Out = In_1(\alpha) - In_2(\alpha)\}])
\end{array}
$$

3.3. Discussions and discoveries

Up to now we have described the structure of TIC library functions which capture various behavior of Simulink library block at the interval level. In this section, we discuss the way of constructing and validating these library functions with some discoveries.[5]

We aim to capture the time-dependent mathematical relationships denoted by library blocks as their intrinsic semantics. Unfortunately, the original Simulink documentation [Mat08a] specifies library blocks in a narrative and sometimes partial manner. This can cause ambiguous interpretation of library blocks and further obstruct the proper usage of Simulink.

For example, the *Relay* library block switches its output according to its relay status, either *on* or *off*. Its original description states that: when the relay is on, the block remains on until its input drops below the value of the *Switch off point* block parameter; when the relay is off, it remains off until its input exceeds the value of the *Switch on point* block parameter. In addition, the *Switch on point* value must be greater than or equal to the *Switch off point* value. However, the description misses the specification of the initial behavior, namely, the relay status at the time point 0. It is thus necessary to clearly capture this initial behavior to avoid confusion of the initial relay state.

To formally model a library block based on its informal and particularly incomplete description, we investigate the behavior by means of simulations under all possible circumstances where the library block may be used in practice. Specifically, we assign different values to the operand parameters of a library block and feed its inputs with various types, continuous or discrete, to simulate all possible situations in which the block may be applied. Reusing the *Relay* library block as an example, we consider the relationship of two parameters and the relationship between the initial input value and two parameters. As shown in Table 1, there are eight cases to observe the initial relay state, where OnV denotes the *Switch on point* value, $OffV$ denotes the *Switch off point* value, and InV denotes the input value at the time point 0.

We also exploit simulations to validate TIC library functions. Namely, we check if the simulation results of a library block conform to the TIC library function of that library block. When an inconsistency occurs, we carefully analyze the original description of the block again and consult with our partners of Simulink, to identify the problem and refine the library function if needed. This not only elevates the confidence of our TIC library functions but also leads to the discovery of a bug of a library block as shown below.

[5] These discoveries have been confirmed by senior application engineers of the MathWorks.

Table 1. The initial relay state of the *Relay* library block in different cases

Case	Relation	Relay
1	$OnV > OffV \wedge InV = OffV$	Off
2	$OnV > OffV \wedge OnV > InV > OffV$	Off
3	$OnV > OffV \wedge InV = OnV$	On
4	$OnV > OffV \wedge InV < OffV$	Off
5	$OnV > OffV \wedge InV > OnV$	On
6	$OnV = OffV \wedge InV = OffV$	On
7	$OnV = OffV \wedge InV < OffV$	Off
8	$OnV = OffV \wedge InV > OffV$	On

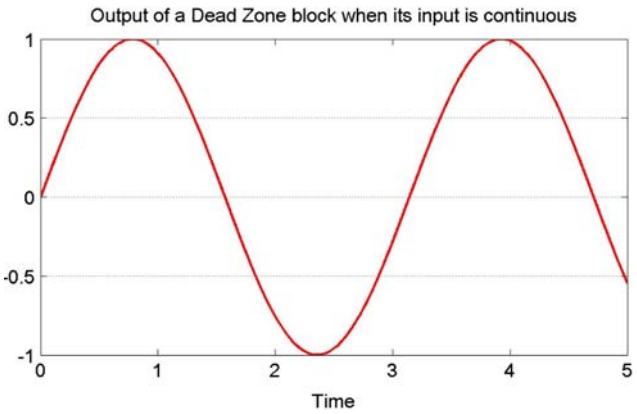


Fig. 2. An incorrect simulation result of the *Dead Zone* library block

The output of the *Dead Zone* library block depends on the relation between its input and a region constrained by a lower limit and an upper limit: (1) the output is zero if the input is within the region, (2) the output is the input minus the upper limit if the input is greater than or equal to the upper limit, (3) the output is the input minus the lower limit if the input is less than or equal to the lower limit.

Unfortunately, the above original description from [Mat08a] is inconsistent with the case where the input is continuous and the upper limit equals the lower limit. Figure 2 depicts a particular simulation result of an elementary block of the *Dead Zone* library block: its output is identical to its input, where both limits are equal to the value 0.5 and the input is a sine wave. However, the input and output should be different according to the description, as the input is greater than or less than 0.5 most of the time.

In this section, we have presented the general structure of TIC library functions which formally model the time-dependent mathematical relationships denoted by Simulink library blocks. We have further discussed the way to elaborately construct and validate TIC library functions. These functions can serve as an accurate and thorough documentation to accompany the original Simulink reference [Mat08a] for the library blocks, and form the foundation used for the automatic transformation from Simulink diagrams to TIC models.

Currently, we have modeled 44 Simulink library blocks of 9 categories including *continuous*, *logic operations*, *math operations*, and so on. These block names are given in Appendix A.1 (where names in the *italic* format are new from our previous work [CD06]), and their TIC library functions are available online [CDS07a]. The main reason for modeling these library blocks is their frequent usage in practice. For example, all 22 library blocks of the *Commonly Used* category defined in [Mat08a] are supported. Moreover, these library blocks have been used in 17 Simulink demos which cover areas of aerospace and automobile. Note that the library blocks of the *Ports and Subsystems* category are improper to be captured here, because their functionalities are usually unpredictable until they are instantiated in specific Simulink diagrams. We will present how to handle the library blocks of this category in the next section.

4. Transforming Simulink diagrams into TIC models

A Simulink diagram is a connected block diagram and is constructed as a tree. We present a strategy in this section to transform Simulink diagrams into TIC models. The transformation is in a bottom-up order, from elementary blocks and wires to diagrams. The transformed TIC models capture the functional and timing aspects of each elementary block, and retain the hierarchical structure of diagrams. At the end of this section, we extend our previous work [CD06] to deal with blocks whose sample times are unspecified, and to handle blocks from the *Ports and Subsystems* category including conditionally executed subsystems. The transformation strategy has been implemented in Java for automation.

4.1. Transforming elementary blocks

A Simulink elementary block is produced by a library block by using the parameterization technique. Similarly, a TIC schema which represents an elementary block is produced by applying relevant argument values to a TIC library function which models the corresponding library block. During the transformation, two aspects are taken into account.

One is the criteria for selecting an appropriate TIC library function for an elementary block. The primary criterion is the *BlockType* block parameter which indicates the functionality of a block. Nevertheless, this parameter is inadequate to distinguish special library blocks which contain operator parameters and can produce elementary blocks with different functionalities. Thus, operator parameters are also considered as additional criteria. Recalling the *Sum* library block in Sect. 3.2, it has an operator parameter *Inputs* which determines the functionality of its produced elementary blocks. Hence, the criteria for choosing a TIC library function for *Sum* consist of two block parameters, *BlockType* and *Inputs*.

The other aspect is about sample times. A sample time of an elementary block is determined by one of the following ways: by the *SampleTime* block parameter, by the library block type (for example, elementary blocks of a continuous library block always have continuous sample times), or by blocks which connect to the block inputs. In addition, the last way relies on the assumption that all sample times of the blocks are specified. We formalize below the last way based on the rules from [Mat08b]. We will also present a simple but effective algorithm in Sect. 4.4 to deal with the case when the assumption is invalid.

Let Blk_In denote the blocks connecting to the inputs of an elementary block, and $InST$ be a function of the type $InST : Blk\_In \to \mathbb{T}$ which returns the sample time of a block in Blk_In.

- We firstly check whether sample times of all blocks in Blk_In are identical. If so, we assign the identical value to be the sample time of the elementary block. Otherwise, we return the value -1 as modeled by the following function $AllEq$ to indicate that the sample time is unspecified.

$$AllEq : \mathbb{P}\, Blk\_In \to (\mathbb{T} \cup \{-1\})$$
$$\forall\, ins : \mathbb{P}\, Blk\_In \bullet \exists res : \mathbb{T} \bullet AllEq(ins) = \mathsf{If}\, \forall\, in : ins \bullet InST(in) = res\ \mathsf{Then}\ res\ \mathsf{Else}\ -1$$

- Next, we check whether there is a sample time of a block in Blk_In which is the greatest common integer divisor (GCD) of the sample times of other blocks in Blk_In. If so, we assign the GCD to be the sample time of the elementary block. Otherwise, we return the value -1 as modeled by the following function $ExiFast$ to indicate that the sample time is unspecified.

$$ExiFast : \mathbb{P}\, Blk\_In \to (\mathbb{T} \cup \{-1\})$$
$$\begin{aligned}&\forall\, ins : \mathbb{P}\, Blk\_In \bullet \exists res : \mathbb{T} \bullet ExiFast(ins) = \\ &\quad \mathsf{If}\, \exists\, in1 : ins \bullet \forall\, in2 : ins \mid in1 \neq in2 \bullet \\ &\qquad \exists k : \mathbb{N} \mid k > 1 \bullet InST(in2) = InST(in1) * k \wedge InST(in1) = res \\ &\quad \mathsf{Then}\ res\ \mathsf{Else}\ -1\end{aligned}$$

- Lastly, when $AllEq$ and $ExiFast$ return the value -1, we derive the sample time according to the solver[6] used in the diagram which contains the elementary block. As modeled by the following function STP, if a

[6] There are two types of solvers for simulations in Simulink: variable-step solvers vary the simulation step size, while fixed-step solvers keep the simulation step size constant. $Solver == \{Variable\_Step, Fixed\_Step\}$.

variable-step solver is used, the sample time is continuous, namely, equaling the value 0. Otherwise, the sample time is the result of function *CalGCD* which returns the GCD of sample times of *Blk_In* if such a GCD exists or the value 0 otherwise.

$$STP : \mathbb{P}\, Blk\_In \times Solver \to \mathbb{T}$$

$$\forall\, ins : \mathbb{P}\, Blk\_In;\ s : Solver \bullet STP(ins, s) =$$
$$\quad \text{If } AllEq(ins) < 0 \text{ Then } AllEq(ins) \text{ Else (If } ExiFast(ins) < 0 \text{ Then } ExiFast(ins)$$
$$\quad\quad \text{Else (If } s = Variable\_Step \text{ Then } 0 \text{ Else } CalGCD(ins)))$$

The above two aspects are important to capture the functional and timing properties of an elementary block. We here reuse the elementary block *Integrator* in Fig. 1 as an example: (1) the selection criterion is the *BlockType* block parameter whose value is *Integrator*, and hence the TIC library function *Integrator* in Sect. 3.2 is chosen, (2) the sample time is 0 since the type of the *Integrator* library block is continuous. This elementary block is modeled by the following schema *calculation_Integrator* which is the result of assigning the value 4 as the initial value to the library function. The expanded form of the schema is also given.

$$calculation\_Integrator \mathrel{\widehat{=}} Integrator(4)$$

calculation_Integrator

$$In_1 : \mathbb{T} \to \mathbb{R};\ Out : \mathbb{T} \nrightarrow \mathbb{R};\ IniVal : \mathbb{R};\ st : \mathbb{T}$$

$$st = 0 \land IniVal = 4 \land Out(0) = IniVal \land \mathbb{I} = [\![Out(\omega) = Out(\alpha) + \textstyle\int_\alpha^\omega In_1]\!]$$

To preserve the hierarchical structure of a Simulink diagram, a transformed TIC schema is named in a conventional manner which composes the name of the path in the diagram. For example, the schema name *calculation_Integrator* is formed by appending the block name *Integrator* to the system name *calculation* with the symbol "_". This naming manner is also adopted by other approaches [ACOS00, CCO05, TSCC05].

4.2. Transforming wires

In Simulink, wires represent input and output relations between connected blocks, and the communication is infinitely fast. Namely, a destination block can receive a value which is produced by a source block at the same time point. As Simulink adopts the continuous-time domain, wires have values at all time points. In other words, a source (or destination) block can write (or read) value to (or from) a wire according to its own execution rate, its sample time. This feature hence enables Simulink to support multi-rate discrete systems as well as hybrid systems which contain blocks with different sample times.

Each wire is converted into an equation at the interval level. The equation consists of two timed traces *src* and *dst*, where *src* denotes the output of a source block ($src : \mathbb{T} \to \mathbb{R}$) and *dst* denotes the input of a destination block ($dst : \mathbb{T} \to \mathbb{R}$). The equivalence of these timed traces is valid for all non-empty intervals, specifically, $\mathbb{I} = [\![src = dst]\!]$. We remark that the way as depicted here is applicable to normal (sub)systems. However, a different way for handling conditionally executed subsystems will be demonstrated in Sect. 4.5.

4.3. Transforming diagrams

A Simulink diagram is made up by linking blocks with wires. It is thus necessary to retain the components and connections of diagrams in transformed TIC models. Our method is similar to Arthan et al's [ACOS00]: the transformation of a diagram is performed after the components of the diagram are transformed into TIC schemas. A diagram is modeled by a TIC schema in the following way.

- Each component is denoted by a schema variable. If a component is a block, the type of the variable is the TIC schema which models the block. Otherwise, the component is an interface such as an input port, and the variable is declared to be a total function from time to real numbers (more details are in Sect. 4.5).
- Each wire is represented by a TIC predicate in the way described in Sect. 4.2, where the variables used in a predicate are the schema variables from the schema declaration.

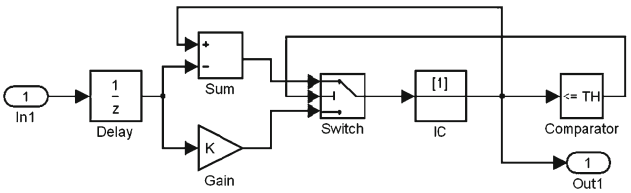


Fig. 3. A Simulink diagram where only blocks *Delay* and *IC* have specified sample times

A Simulink block can be either an elementary block or a diagram itself representing a subsystem. The way to handle elementary blocks is illustrated in Sects. 4.1 and 4.5 shows how to deal with subsystems.

4.4. Computing unspecified sample times

In Sect. 4.1, we described the way to determine sample times of elementary blocks. However, this way is often inapplicable in practice, as users usually assign explicitly sample times to some blocks and leave other blocks with unspecified sample times which are equal to the value -1.

Taking Fig. 3 as a running example in this section, the diagram consists of six elementary blocks, one input port (*In1*) and one output port (*Out1*), and only the sample times of blocks *Delay* and *IC* are specified with values 1 and 2 respectively. We remark that in the beginning the sample time of the elementary block *Switch* is unknown by using the method in Sect. 4.1 since its block type is not continuous and the sample times of blocks *Sum*, *Comparator* and *Gain* which connect to its inputs are unspecified.

It is thus necessary to develop an algorithm to handle the unspecified sample times which cannot be directly computed in the previous way. Our algorithm can automatically compute *derivable* sample times which are a subset of unspecified sample times. The unspecified sample time of an elementary block is derivable provided one of the following conditions holds.

1. One of the blocks connecting to the elementary block has continuous sample time;
2. All sample times of the blocks connecting to the elementary block are specified;
3. All sample times of the blocks connecting to the elementary block are either specified or derivable.

When the elementary block fulfills condition 1, its sample time is equal to the value 0 [Mat08b]. When condition 2 is satisfied, the sample time can be computed by applying the function *STP* defined in Sect. 4.1. When condition 3 holds, the sample time can be derived after finishing the computation of the derivable sample times of the blocks connecting to the elementary block.

According to the above conditions, we can deduce that the sample times of *Sum*, *Gain* and *Comparator* in Fig. 3 are derivable by condition 2 and the sample time of *Switch* is also derivable by condition 3.

Based on the computability property of derivable sample times, we have developed a simple and effective algorithm as shown below to handle unspecified sample times. The algorithm starts with a non-empty list *BLKS* of elementary blocks whose sample times are unspecified, and *repeatedly* modifies the list *until* its termination condition at line 15 holds. To be specific, the algorithm terminates when either all sample times are specified or none of the unspecified sample times is derivable. The first case is examined by method *Empty* which checks if the list is empty, and the second case is examined by method *CheckEq* which checks if there is a change of the list after executing the *for* loop from lines 3 to 14 once.

An element b in *BLKS* is analyzed with respect to three cases (lines 5–12).

- Lines 5–7 correspond to condition 1. Method *ExistsContinuous* checks whether there exists a block whose sample time is continuous and the block connects to b (namely, the block is in the result returned by method *GetInBLKSST*). If *ExistsContinuous* returns true, the sample time of b is equal to the value 0, and then b is deleted from *BLKS* by method *Delete*.
- Lines 8–10 handle condition 2. Method *AllSTKnown* checks if all sample times of the blocks connecting to b are specified. If *AllSTKnown* returns true, we apply method *CallSTP* which implements the function *STP* as defined in Sect. 4.1 to compute the sample time of b, and then delete b from *BLKS*.
- Line 11 indicates that there is inadequate information to calculate the sample time of b. We simply do nothing to finish the analysis of b.

Algorithm 1: Deal with all unspecified sample times

Require: A non-empty list of elementary blocks *BLKS* whose sample times are unspecified

```
repeat
    iniBLKS ← BLKS
    for all i = 1 to BLKS.length do
        b ← BLKS[i]
        if ExistsContinuous(b.GetInBLKSST()) then
            b.st ← 0
            BLKS ← Delete(BLKS, b)
        else if AllSTKnown(b.GetInBLKSST()) then
            b.st ← CallSTP(BLKS, b)
            BLKS ← Delete(BLKS, b)
        else skip
        end if
        i ← i + 1
    end for
until Empty(BLKS) or CheckEq(BLKS, iniBLKS)
```

We illustrate how the algorithm can systematically compute the unspecified sample times in Fig. 3. Initially, *BLKS* consists of four blocks, *Sum*, *Gain*, *Switch* and *Comparator*. After the first time the *for* loop is executed, *Gain*, *Sum* and *Comparator* are deleted from *BLKS* with specified sample times which are 1, 1 and 2 respectively. After the second execution of the *for* loop, *Switch* is deleted with its sample time which is 1. The algorithm hence terminates since *BLKS* becomes empty.

We remark that Fig. 3 contains two loops. One is formed by *Sum*, *Switch* and *IC*, and the other comprises *Switch*, *IC* and *Comparator*. Loops are frequently used in Simulink to graphically model differential equations or feedback control systems. Our algorithm can calculate derivable sample times of blocks which are in loops, and can thus transform diagrams containing loop structures with the preservation of timing information. On the other hand, [TSCC05] lacks the support for deriving block sample times in loops.

4.5. Dealing with the *Ports and Subsystems* category

The transformation presented in Sect. 4.1 handles the elementary blocks whose library blocks are modeled by the TIC library functions defined in Sect. 3. However, it is difficult and impracticable to define TIC library functions to model the library blocks of the *Ports and Subsystems* category, because the behavior of their instances as produced elementary blocks in particular diagrams is usually unpredictable. We hence directly model their instances in TIC during the transformation. We demonstrate below how to deal with these library blocks based on their usage, specifically, either denoting interfaces or creating subsystems. Appendix A.2 lists the library block names of this category supported so far.

- Library blocks *Inport*, *Outport*, *Enable*, and *Trigger* are designed to create (sub)system interfaces. For example, instances of the *Outport* library block represent outputs of (sub)systems. An instance of one of these four library blocks in a (sub)system is transformed to a total function of time in a TIC schema which models the (sub)system. Furthermore, an instance of the *Inport* or *Outport* library block in a *plain* or *enabled* subsystem (which will be explained more in the following context) is declared to be continuous provided the block which connects to it is continuous.
  Reusing the system *calculation* in Fig. 1 as an example: its input port *Acc* is an instance of the *Inport* library block, and its output port *Speed* is an instance of the *Outport* library block. These ports are represented by two functions declared in the following schema *calculation* which models the system. These functions are in turn used in the predicates of *calculation*. Moreover, *Speed* is continuous as indicated by $\twoheadrightarrow$ because the elementary block *Integrator* outputs continuously. Note that the schema *calculation_Integrator* which models *Integrator* is presented in Sect. 4.1.

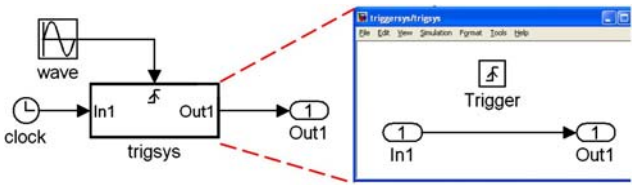


Fig. 4. A triggered subsystem with a continuous control input

calculation

$Acc : \mathbb{T} \to \mathbb{R};\ Integrator : calculation\_Integrator;\ Speed : \mathbb{T} \nrightarrow \mathbb{R}$

$\mathbb{I} = [\![Acc = Integrator.In_1]\!] \land \mathbb{I} = [\![Integrator.Out = Speed]\!]$

- The *Subsystem* library block is applied to form plain subsystems which virtually reduce the number of blocks displayed in Simulink diagrams and form the hierarchical structure of Simulink diagrams. The way to translate plain subsystems is the same as the one for diagrams described in Sect. 4.3.
  Library blocks *Enabled Subsystem* and *Triggered Subsystem* are used to construct enabled subsystems and triggered subsystems respectively which are *conditionally executed* subsystems. Here we expand our previously primitive work in [CD06] by deeply analyzing these subsystems and illustrating our solutions with examples in the following two subsections.

4.5.1. Triggered subsystems

A *triggered* subsystem executes each time a trigger event occurs. A trigger event is determined by a control input which is an instance of the *Trigger* library block. There are three types of trigger events, *rising*, *falling*, and *either*, according to the direction the control input crosses the value 0. For instance, a *rising* trigger event occurs, when the control input rises from a negative or zero value to a positive value.

When no events occur, triggered subsystems always hold their outputs at the last value between trigger events [Mat08b]. In addition, Simulink constrains the sample times of components in a triggered subsystem in the following way: all blocks and interface ports such as an input have the same sample times of its control input. To model a triggered subsystem, we focus on modeling the way which assigns values to system inputs under different circumstances of the system control input, whether a trigger event occurs or not. This is because the subsystem outputs and the behavior of its components are dependent on the subsystem inputs.

Triggered subsystem can be classified into two groups in terms of their control inputs, which can be either continuous or discrete. We present here how to handle triggered subsystems whose control inputs are continuous. Appendix C.1 shows the way to support the other group where control inputs are discrete.

When the control input of a triggered subsystem is continuous, a trigger event occurs only at a point-interval where the starting point equals the ending point. We specify the subsystem behavior in three situations: the presence of trigger events, the absence of trigger events in non-initial intervals which start with positive time points, and the absence of trigger events in initial intervals which start with the time point 0. We remark that the last situation is not documented in [Mat08b, Mat08a].

For a better illustration, we use the simple system in Fig. 4 as an example. Triggered subsystem *trigsys* outputs the moments when its continuous control input *Trigger* rises from a negative or zero value to a positive value. In the following schema *sys_trigsys*, which denotes the triggered subsystem, *Trigger* is defined by a total function whose range consists of two values, 1 and 0, where the value 1 indicates the presence of a trigger event and the value 0 indicates the absence. Since we aim to model the behavior of triggered subsystems with respect to trigger events, we represent control inputs at the above abstract level without specifying how to detect trigger events. In addition, *sys_trigsys* captures a timing feature that trigger events occur only at point-intervals by the predicate $[Trigger = 1] \subseteq [\alpha = \omega]$ where $[\ ]$ indicates a set of *left and right-closed* intervals.

sys_trigsys

$Trigger : \mathbb{T} \to \{0, 1\};\ In1, Out1 : \mathbb{T} \to \mathbb{R}$

$[Trigger = 1] \subseteq [\alpha = \omega] \land \mathbb{I} = [\![In1 = Out1]\!]$

sys

$clock : sys\_clock;\ trigsys : sys\_trigsys;\ \ldots$

$\ldots$

$[\![trigsys.Trigger = 1]\!] \subseteq [\![clock.Out = trigsys.In1]\!]$ [Predicate1]

$[\![trigsys.Trigger = 0 \wedge \alpha > 0]\!] \subseteq [\![trigsys.In1(\alpha) = trigsys.In1]\!]$ [Predicate2]

$[\![trigsys.Trigger = 0 \wedge \alpha = 0]\!] \subseteq [\![trigsys.In1 = 0]\!]$ [Predicate3]

In the above schema *sys* which includes the subsystem *trigsys* by the declaration $trigsys : sys\_trigsys$, the conditionally executed behavior of *trigsys* is modeled by three predicates which constrain the wire between the elementary block *clock* and the subsystem input $trigsys.In1$. Predicate1 states that $trigsys.In1$ equals the output of *clock* when a trigger event happens. Predicate2 constrains that the values of $trigsys.In1$ in a non-initial interval (indicated by $\alpha > 0$) are equal to the $trigsys.In1$ value at the starting point of the non-initial interval when no trigger event occurs. Predicate3 depicts that when no trigger event happens in an initial interval (by $\alpha = 0$), the values of $trigsys.In1$ over the initial interval are 0 by default. Note that the default value 0 is obtained from our experiments, although it is unspecified in the Simulink documentation.

We informally explain below the reason to represent the last value between trigger events by using α in Predicate2. When a trigger event occurs at a time point t, we have $clock.Out(t) = trigsys.In1(t)$ according to Predicate1. Based on Predicate2, we can deduce that the set of intervals as denoted by $[\![trigsys.Trigger = 0 \wedge \alpha > 0]\!]$ contains a *left-open* interval i such that (1) its starting point is t, namely, $\alpha(i) = t$, and (2) the values of $trigsys.In1$ in i are equal to $trigsys.In1(\alpha(i))$, namely, $trigsys.In1(t)$. Furthermore, we can imply that in any interval which starts at t and ends before a time point t' at which the next trigger event happens, the values of $trigsys.In1$ are the same as $trigsys.In1(t)$. For instance, for a point-interval $[x \ldots x]$ where $t < x < t'$, we can find a *left-open*, *right-closed* interval $(t \ldots x]$ such that $trigsys.In1(t) = trigsys.In1(x)$ because of the TIC expression $[\![trigsys.In1(\alpha) = trigsys.In1]\!]$.

4.5.2. Enabled subsystems

An *enabled* subsystem executes when the value of its control input which is an instance of the *Enabled* library block is positive. Namely, an enabled subsystem starts its execution from the moment when its control input value crosses zero from a negative value and continues its execution in the interval in which the control input values remain positive. When an enabled subsystem is disabled, it can output either its last values or its initial output values. We demonstrate here how to model the enabled subsystems which output their last values when they are disabled. Nevertheless, the enabled subsystems which output their initial output values can be handled similarly with auxiliary variables for storing the initial output values.

We concentrate on specifying the behavior which associates the subsystem inputs with their control inputs. Unlike triggered subsystems which restrict all components of a triggered subsystem to have the same sample time, enabled subsystems in Simulink can include components with different sample times [Mat08b]. This loose restriction results in the difficulty of representing the last values of enabled subsystems, especially when their control inputs are discrete. We present below our solution to model the enabled subsystems whose control inputs are discrete. Appendix C.2 shows how to support the enabled subsystems whose control inputs are continuous.

When the control input of an enabled subsystem is discrete, we can represent the behavior of its control input by specifying its value at every sample time hits. Nevertheless, the logic that Simulink uses to update enabled subsystems with different sample times of their components is complicated to be identified, as discussed by Tripakis et al. [TSCC05]. Currently, we restrict ourselves to cope with a subset of enabled subsystems: all sample times of components within an enabled subsystem are inherited by the inputs of the subsystem, and all subsystem inputs have the same sample times. These restrictions are weaker than those of Tripakis et al., as we allow the sample time of the control input to be different from other sample times.

Taking the enabled subsystem *open* in Fig. 5 as an example, the enabled subsystem (used in our case study in Sect. 6.1) multiplies its continuous input *volumeIn* by the constant -0.1 when it is enabled; otherwise, it outputs the last value. The control input *Enable* is linked by elementary block *inverse* whose sample time is 1. In the following TIC schemas, $tank\_plant\_open\_K$ represents block K and $tank\_plant\_open$ represents *open*. The input *volumeIn* of *open* is connected by a continuous block (refer to Sect. 6.1 for the connection), so it is declared to be a continuous function in $tank\_plant\_open$. Therefore, the sample time of K is 0, and the output *flow* is also continuous based on the discussion at the beginning of Sect. 4.5.

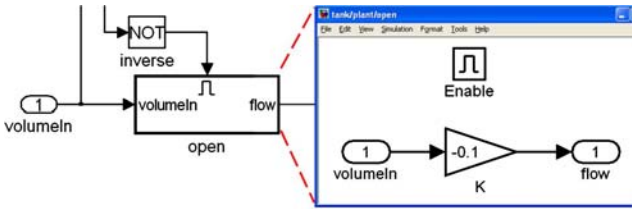


Fig. 5. An enable subsystem *open* in a subsystem *plant* of a system *tank*

$$tank\_plant\_open\_K \mathrel{\widehat{=}} Gain(0, -0.1)$$

tank_plant_open

$Enable : \mathbb{T} \to \mathbb{R};\ volumeIn, flow : \mathbb{T} \nrightarrow \mathbb{R};\ K : tank\_plant\_open\_K$

$\mathbb{I} = [\![volumeIn = K.In_1]\!] \wedge \mathbb{I} = [\![K.Out = flow]\!]$

Note that *tank_plant_open* captures the components and wires in the enabled subsystem *open*. The conditionally executed behavior of *open* is specified in the following schema *tank_plant* which represents the subsystem *plant* that contains *open*. To be specific, we model the relation between the assignment of the *open* input *open.volumeIn* and the *open* control input *open.Enable* in every sample time interval. We remark that a sample time interval is *left-closed*, *right-open* (in Definition 2). Moreover, discrete systems in Simulink execute only at sample time hits; particularly, elementary block *inverse* outputs values to *open.Enable* every 1 time unit. As *open* can be either enabled or disabled at two endpoints of a sample time interval, there are four cases about the *open* status at the endpoints. The sample time intervals during which *open* is disabled need to be further distinguished in two groups based on their starting points, because when an enabled subsystem is disabled in the initial sample time interval, its last value is 0 by default. The default value is known from our experiments, although it is missed in the Simulink documentation [Mat08b].

tank_plant

$volumeIn : \mathbb{T} \nrightarrow \mathbb{R};\ open : tank\_plant\_open;\ \ldots$

$\ldots$

$[open.Enable \leq 0 \wedge open.Enable(\omega) > 0 \wedge \alpha = 0 \wedge \omega = 1) \subseteq [open.volumeIn = 0)$ [Predicate1]

$[open.Enable \leq 0 \wedge open.Enable(\omega) \leq 0 \wedge \alpha = 0 \wedge \omega = 1)$
$\quad \subseteq [open.volumeIn = 0 \wedge open.volumeIn(\omega) = 0)$ [Predicate2]

$[open.Enable \leq 0 \wedge open.Enable(\omega) > 0 \wedge \exists k : \mathbb{N}_1 \bullet \alpha = k \wedge \omega = k + 1)$
$\quad \subseteq [open.volumeIn(\alpha) = open.volumeIn)$ [Predicate3]

$[open.Enable \leq 0 \wedge open.Enable(\omega) \leq 0 \wedge \exists k : \mathbb{N}_1 \bullet \alpha = k \wedge \omega = k + 1)$
$\quad \subseteq [open.volumeIn(\alpha) = open.volumeIn \wedge open.volumeIn(\alpha) = open.volumeIn(\omega))$ [Predicate4]

$[open.Enable > 0 \wedge open.Enable(\omega) > 0 \wedge \exists k : \mathbb{N} \bullet \alpha = k \wedge \omega = k + 1)$
$\quad \subseteq [volumeIn = open.volumeIn)$ [Predicate5]

$[open.Enable > 0 \wedge open.Enable(\omega) \leq 0 \wedge \exists k : \mathbb{N} \bullet \alpha = k \wedge \omega = k + 1)$
$\quad \subseteq [volumeIn = open.volumeIn \wedge volumeIn(\omega) = open.volumeIn(\omega))$ [Predicate6]

The first four predicates depict the behavior when *open* is disabled in a sample time interval, indicated by the expression $open.Enable \leq 0$. Predicate1 and Predicate2 are concerned with the initial sample time interval which starts with the time point 0: the values of *open.volumeIn* are equal to 0 in the interval. In addition, if *open* is still disabled at the ending point, namely, $open.Enable(\omega) \leq 0$, then the value of *open.volumeIn* is 0 at the ending point (in Predicate2). Predicate3 and Predicate4 deal with the non-initial sample time intervals ($[\exists k : \mathbb{N}_1 \bullet \alpha = k \wedge \omega = k + 1)$), where the last value is the *open.volumeIn* value at the starting point ($open.volumeIn(\alpha)$). Furthermore, if *open* is still disabled at the ending point, we assign the last value to be the *open.volumeIn* value at the ending point (in Predicate4). The above way for representing the last value is similar to the one handling triggered subsystems in Sect. 4.5.1.

Predicate5 and Predicate6 model the behavior when *open* is enabled in a sample time interval. Specifically, *open.volumenIn* equals the input *volumeIn* of *plant* at the same time point in the interval. Moreover, if *open*

becomes disabled at the ending point, the *volumeIn* value at the ending point is the last value for *open.volumeIn* (namely, $volumeIn(\omega) = open.volumeIn(\omega)$ in Predicate6), based on mathematical theories about limits of continuous functions, *volumeIn* here. In the case that *volumeIn* is discrete, we can specify the last value as the value of *volumeIn* at the most recent sample time hit when *volumeIn* updates its value before the ending point.

Up to now, we have described the way of transforming Simulink diagrams to TIC models, and the transformation can preserve the functional and timing aspects as well as diagram structure. We have also presented the algorithm to compute unspecified sample times in (complex) Simulink diagrams. We have further discussed how to handle the library blocks of the *Ports and Subsystems* category, in particular enabled subsystems and triggered subsystems. In this section, we support 7 library blocks of this category, and hence our approach covers totally 51 library blocks of 10 categories (other 44 library blocks are modeled in Sect. 3). Based on transformed TIC models, we can specify requirements over diagrams or some components, and rigorously reason about their validity with a high grade of automation, as we will show in the following sections.

5. Machine-assisted proof support for TIC models of Simulink diagrams

We have described so far how TIC can formally model Simulink diagrams. The formal verification capability of TIC can be exploited for rigorously validating Simulink diagrams against requirements. In addition, validation is non-trivial as it usually involves continuous dynamics and arbitrary (infinite) intervals. When diagrams are complex, manual proving is inadequate as it is difficult to ensure the correctness of each proof step and to manage all proof details. It is thus necessary and important to develop the machine-assisted proof support for TIC models of Simulink diagrams.

In the previous work [CDS08], we developed a generic verification system based on PVS for TIC. However, we considered only the translation of two types of TIC models: TIC schemas which represent system properties and TIC predicates which denote requirements. In another previous work [CDS07b], we constructed a set of PVS library types for the TIC library functions. Nevertheless, the construction was manual. In this section, we extend our previous work by supporting automatic transformation of axiomatic definitions which are the type of the TIC library functions. We also compare the PVS functions transformed from the TIC library functions with our previous work. At the end of this section, we define and validate a collection of supplementary rules dedicated to Simulink modeling features.

5.1. Translating TIC library functions

In Sect. 3, we defined a set of TIC library functions to model Simulink library blocks. These library functions return TIC schemas which capture the time-dependent relationships denoted by Simulink blocks.

Our previous work [CDS07b] represented a TIC library function by a PVS parameterized record type, which returned a record type to denote a returned TIC schema of the library function. In addition, each schema predicate was used to constrain the type of a record accessor which represented a schema variable.

However, this work possesses some disadvantages when translating TIC to PVS and conducting proofs in PVS. Firstly, as a schema predicate may involve several schema variables, it is difficult to automatically and correctly associate the predicate with the corresponding schema variable. Previously, we manually constructed the association and in turn the parameterized record types for TIC library functions. Secondly, when applying a property of an elementary block in a proof in PVS, users have to remember all record accessors of the record type which models the schema of the elementary block. This is inconvenient. Last but not least, the work is only suitable for TIC library functions, and is deficient to support axiomatic definitions [WD96] which are the type of the library functions.

We start illustrating our current approach with the analysis of general axiomatic definitions, followed by an application to a specific TIC library function. An axiomatic definition usually introduces a global variable and specifies constraints of the declared variable. Such a definition is said to be axiomatic, as constraints are assumed to hold whenever the variable is used. A general form can be depicted below, where *Predicate* denotes constraints on a variable *Decl_Name*, and *Expression* specifies the type of the variable. A specific axiomatic definition *square* which returns a square of a natural number is also given below.

$$\begin{array}{|l} Decl\_Name : Expression \\ \hline Predicate \end{array} \qquad \begin{array}{|l} square : \mathbb{N} \to \mathbb{N} \\ \hline \forall n : \mathbb{N} \bullet square(n) = n * n \end{array}$$

In our approach, an axiomatic definition is translated to a PVS function which is a constant declaration. We remark that PVS constant declarations introduce new constants such as functions (as explained in Sect. 2.3). To be specific, for an axiomatic definition, *Decl_Name* is used as the identifier of a PVS function, *Expression* is converted into PVS specifications, and *Predicate* is translated to a PVS axiom whose identifier is also *Decl_Name*. These mappings are sketched below. We present the corresponding PVS function of the function *square* as well.

```
1  Decl_Name : Expression;
2  Decl_Name : AXIOM Predicate;
```

```
1  square : [nat -> nat];
2  square : AXIOM FORALL (n: nat):
3                    square(n) = n * n;
```

We remark that the name overloading technique is applied to construct PVS specifications of axiomatic definitions. In the above PVS specifications of *square*, line 1 declares a constant named `square` which is a total function from natural numbers to natural numbers, and lines 2 and 3 model the function property as an axiom whose identifier is also named `square`.

A TIC library function is an axiomatic definition, and is thus translated to a PVS function which returns a set of PVS records. Parameters of a translated PVS function correspond to the arguments of the corresponding TIC library function, and the set of returned records represents a schema returned by the TIC library function. Taking the TIC library function *Integrator* in Sect. 3.2 as an example, it is translated to the following PVS function `Integrator` where keywords such as `Trace`, `Time`, `AllS`, `ALPHA`, `OMEGA`, and `LIFT` encode TIC semantics in PVS (the detailed encoding of these keywords was available in [CDS08]).

$$Integrator : \mathbb{R} \to \mathbb{P}[In_1 : \mathbb{T} \to \mathbb{R};\ Out : \mathbb{T} \nrightarrow \mathbb{R};\ IniVal : \mathbb{R};\ st : \mathbb{T}]$$

$$\forall init : \mathbb{R} \bullet Integrator(init) = [In_1 : \mathbb{T} \to \mathbb{R};\ Out : \mathbb{T} \nrightarrow \mathbb{R};\ IniVal : \mathbb{R};\ st : \mathbb{T} \mid st = 0 \land IniVal = init \land Out(0) = IniVal \land \mathbb{I} = [\![Out(\omega) = Out(\alpha) + \int_{\alpha}^{\omega} In_1]\!]]$$

```
1  Integrator: [real -> setof[[# In1: Trace, Out: Trace, IniVal: real, st: Time #]]];
2  Integrator: AXIOM  FORALL (init: real): Integrator(init) =
3      { temp: [# In1: Trace, Out: Trace, IniVal: real, st: Time #] |
4        temp`st = 0 AND temp`IniVal = init AND temp`Out(0) = temp`IniVal AND
5        fullset = AllS((LIFT(temp`Out) o LIFT(OMEGA)) = (LIFT(temp`Out) o LIFT(ALPHA)) +
6                              TICIntegral(LIFT(ALPHA), LIFT(OMEGA), temp`In1)) AND
7        continuous(temp`Out)};
```

- Line 1 declares the type of `Integrator`, namely, from real numbers to a set of records (denoted by `setof[[# ... #]]`). Each record comprises four accessors, and each accessor is associated with its corresponding type. For example, the type of the accessor `In1` is a timed trace encoded by `Trace`.
- The `AXIOM` specification from lines 2 to 7 models the properties of `Integrator`. An auxiliary variable *temp* defined at line 3 is used for easily referring to record accessors by the PVS projection function (`‘`). For instance, `temp‘st` indicates the accessor `st`. Line 4 captures the timing feature of the sample time, and relationships between record accessors and PVS function parameters. The predicate at lines 5 and 6 represents the integration at the interval level in PVS, where the PVS keyword `fullset` denotes all non-empty intervals $\mathbb{I}$, the function `AllS` encodes the interval brackets $[\![\ ]\!]$, the PVS operator `o` is for the function composition, and the function `LIFT` unifies different types of basic TIC elements to the same type of functions which take both time points and intervals as parameters and return real numbers.
  Note that continuous behavior is supported based on the NASA PVS library [But04]. The function `TICIntegral` at line 6 encodes an integral operation of the timed trace `In1`. Furthermore, the continuity feature of the output `Out` is captured at line 7 by the function `continuous`.

The above PVS function `Integrator` follows closely the TIC library function *Integrator* in terms of the structure. Schema predicates in *Integrator* are converted to the constraints in `Integrator` for *all* record accessors. This is more straightforward than our previous work which had to identify the association between predicates and record accessors. For example, both predicates $IniVal = init$ and $Out(0) = IniVal$ contain the variable *IniVal*, and the former restricts the value of *IniVal*, while the latter restricts the value of the timed trace *Out* at the time point 0.

Moreover, the structure of `Integrator` can facilitate the reasoning process in PVS, in particular, when applying the property of the *Integrator* library block. If using our previous approach, we have to enter the proof command `typepred` with an accessor name four times because there are four accessors. In contrast, we only

enter the proof command `lemma` with the function name once. It is also easier in practice for users to recall the functions names which are identical to the library block types than the record accessors.

We have described the way of systematically handling axiomatic definitions. Comparison with our previous work has been provided as well. We have implemented our approach using Java and hence extend our existing work [CDS08] to automatically translate axiomatic definitions to PVS functions.

5.2. Facilitating TIC validation of Simulink diagrams in PVS

We define a collection of supplementary rules dedicated to Simulink modeling features to increase the efficiency of TIC validation in PVS. We categorize these rules into two groups: one group deals with connections in Simulink diagrams, and the other handles discrete systems modeled in Simulink.

Wires in Simulink diagrams are represented by equations in TIC (as shown in Sect. 4.2). Each equation involves two timed traces which denote connected block ports. When reasoning about TIC models of Simulink diagrams, we often need to substitute one timed trace for another provided they indicate a connection. In other words, values of these two timed traces are equal in any interval. However, this kind of substitution is tedious in PVS since we have to completely expand the detailed encoding of the TIC semantics to the low level of time points to enable the PVS prover to automatically discharge proof goals. To simplify the substitution process, we define a set of rewriting rules to replace one timed trace by another at the interval level under various circumstances. For example, the following rule `BB_eq_sub` passes a constant value `v` between two equivalent timed traces `tr1` and `tr2`, namely, from `tr1` to `tr2` and vice versa. This rule saves five proof steps such as expanding the function `AllS` which encodes the interval brackets ⟦ ⟧ in PVS.

```
BB_eq_sub: LEMMA FORALL (tr1, tr2: Trace, v: real):
     fullset = AllS(LIFT(tr1) = LIFT(tr2))
  => AllS(LIFT(tr1) = LIFT(v)) = AllS(LIFT(tr2) = LIFT(v));
```

Discrete systems with periodic execution are a common application domain of Simulink. The time domain of these discrete systems in Simulink is decomposed into a sequence of sample time intervals as defined in Sect. 3.1, and systems are updated at sample time hits. Based on these features, we define several domain-specific rules to ease the analysis of this domain. For instance, to verify a safety requirement `pl` of a discrete system, we can apply the following rule `CO_to_All` if `pl` is independent of interval operators (namely, α, ω, and δ). Note that a safety requirement must be valid in every non-empty interval. The rule simplifies the checking of `pl` by considering only all sample time intervals, which are expressed in PVS at lines 2 and 3, rather than all non-empty intervals. In addition, the function `No_Term?` at line 1 indicates that `pl` is not affected by any interval operator.

```
1  CO_to_All: LEMMA st > 0 AND No_Term?(pl) =>
2       subset?(COS(exNat( lambda(k: nat): LIFT(ALPHA) = LIFT(k) * LIFT(st) AND
3                                  LIFT(OMEGA) = (LIFT(k) + LIFT(1)) * LIFT(st))),
4               COS(pl))
5    => fullset = AllS(pl);
```

We have constructed 25 supplementary rules which have been validated in PVS. These rules facilitate the validation of Simulink diagrams by elevating the automation grade in PVS. We will show the capability and advantages of our formal framework by our experimental studies in the following section.

6. Experimental studies

To assist the usability of our approach, we apply the Java technology to implement the framework. The work flow is shown in Fig. 6. Two translators automatically transform system designs. Specifically, *Sim2TIC* transforms Simulink diagrams into TIC models, and *TIC2PVS* translates the TIC models of requirements and Simulink library blocks and diagrams to PVS specifications. Moreover, supplementary rules are imported to simplify proving processes in PVS.

Using this framework, we can rigorously validate various systems modeled in Simulink, such as multi-rate discrete or hybrid systems. Open systems are also supported, as we can formally specify constraints of environment variables in TIC based on the TIC models transformed from Simulink diagrams. With the powerful

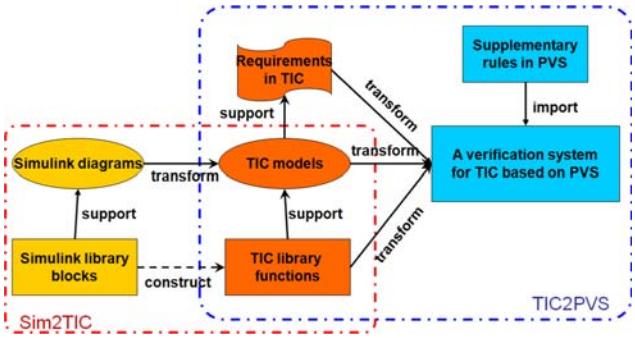


Fig. 6. Work flow of the formal framework on modeling and validating Simulink diagrams

verification capability, the framework can handle the analysis of continuous dynamics and validate important timing requirements such as bounded liveness requirements.

We here apply our framework to an adapted tank control system from [KSF+99] for illustrating the benefits of the framework. The system under consideration simplifies the mathematical functions of physical variables for the sake of easier explanation. Nevertheless it possesses continuous and discrete behavior as well as discrete logics. We firstly describe the system description, specifications and requirements. Next, we sketch the validation of two requirements. One requirement is concerned with bounded liveness, and the other is a safety requirement.

6.1. Specifications of system design and requirements

The Simulink diagram of the tank control system is shown in Fig. 7, and mainly consists of two subsystems. Subsystem *plant* describes the continuous behavior of water volume in a tank, and subsystem *controller* depicts the control logic of a tank valve.

- In the subsystem *plant*: Input port *volumeIn* and output port *volumeOut* indicate the water volume. Input port *valve* denotes commands from the subsystem *controller*, and its value is either 0 meaning that the valve must be open, or 1 meaning that the valve must be closed. Enabled subsystems *close* and *open* model water flow rate as their outputs *flow* for different commands. Elementary block *inverse* outputs the value 1 if its input value is 0, and outputs the value 0 otherwise.
  - The subsystem *close* is enabled when the valve is closed, and $flow = (volumeIn - 6) * -0.1$.
  - The subsystem *open* is enabled when the valve is open, and $flow = volumeIn * -0.1$.

  Elementary block *switch* outputs an appropriate flow rate according to *valve*. If the valve is open, *switch* outputs its third input which is connected by *open*; otherwise, it outputs its first input which is connected by *close*. Elementary block *accumulate* is an instance of the *Integrator* library block. Namely, it continuously outputs the water volume by integrating the flow rate. Initially the water volume equals to the value 2, which is specified via the *InitialCondition* block parameter of *accumulate*.
- In the subsystem *controller*: Input port *volume* indicates the water volume from the subsystem *plant*. Output port *valve* denotes commands for the valve, which depend on *volume*. Specifically, when the *volume* value is not smaller than a maximum value 3, *valve* outputs the value 0 to open the valve; and when the *volume* value is not larger than a minimum value 1, *valve* outputs the value 1 to close the valve. Elementary block *initial* outputs the value 1 at the time point 0, which indicates that the valve is initially closed. After that time point, *initial* outputs values from *switch*. Elementary block *max* compares its input with the constant 3; its output value is 1 when its input value is less than 3, and is 0 otherwise.

In practice, periods of communications are not negligible. Elementary block *delay* which is an instance of the *Unit Delay* library block (defined in Sect. 3.2) models a delay from *controller* to *plant*; specifically, a command is postponed for 1 time unit.

We apply the strategy presented in Sect. 4 to automatically transform the Simulink diagram to TIC schemas based on the TIC library defined in Sect. 3. The transformation captures the timing aspect, namely, the sample times of the diagram. In the diagram, only the sample time of *delay* is specified with the value 1, and the sample

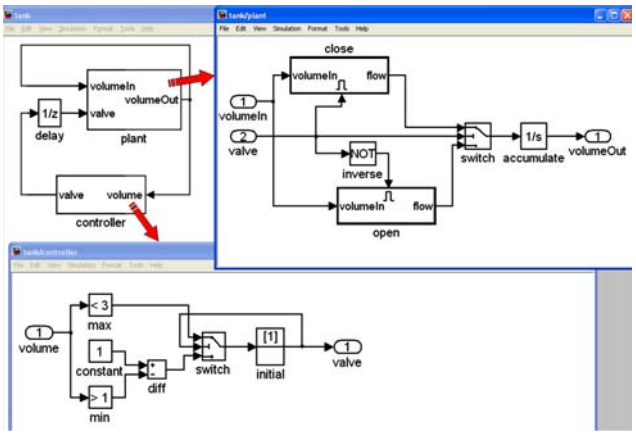


Fig. 7. A Simulink diagram models the water tank control system

times of other blocks are unspecified. Based on the method formalized in Sect. 4.1, the sample time of *accumulate* is 0 due to its library block type. Using the algorithm presented in Sect. 4.4, we can derive all unspecified sample times. For example, the elementary block *switch* in *plant* is connected by *valve* in *plant* and the enabled subsystems *open* and *close*, and its sample time is 0. This is because the outputs of *open* and *close* are continuous. Note that the sample time of *valve* in *plant* is 1 as *valve* inherits the sample time of *delay*. We present some transformed TIC schemas below, which are used in the requirement specifications and proofs. The complete TIC schemas of the Simulink diagram and the TIC library functions used are available in Appendix B.

- The following TIC schemas model wires in the subsystem *plant* and some of its components. For example, schema *tank_plant_inverse* captures the sample time of *inverse* which equals 1 time unit as inherited from *delay*, and schema *tank_plant_accumulate* stores the initial water volume as indicated by the value 2. We ignore schemas *tank_plant_close* and *tank_plant_open* which respectively denote the enabled subsystems *close* and *open*. Note that *open* was analyzed in Sect. 4.5.2. We also omit TIC predicates which specify the conditional behavior of the enabled subsystems in schema *tank_plant* below.

$$tank\_plant\_inverse \mathrel{\widehat{=}} Logic\_NOT(1) \qquad tank\_plant\_accumulate \mathrel{\widehat{=}} Integrator(2)$$
$$tank\_plant\_switch \mathrel{\widehat{=}} Switch\_G(0, 0)$$

tank_plant

$volumeIn, volumeOut : \mathbb{T} \nrightarrow \mathbb{R};\ valve : \mathbb{T} \to \mathbb{R};\ inverse : tank\_plant\_inverse$
$switch : tank\_plant\_switch;\ accumulate : tank\_plant\_accumulate$
$close : tank\_plant\_close;\ open : tank\_plant\_open$

$\ldots$
$\mathbb{I} = [\![volumeIn = close.volumeIn]\!] \wedge \mathbb{I} = [\![volumeIn = open.volumeIn]\!] \wedge$
$\mathbb{I} = [\![valve = close.Enable]\!] \wedge \mathbb{I} = [\![valve = inverse.In_1]\!] \wedge \mathbb{I} = [\![close.flow = switch.In_1]\!] \wedge$
$\mathbb{I} = [\![valve = switch.In_2]\!] \wedge \mathbb{I} = [\![inverse.Out = open.Enable]\!] \wedge \mathbb{I} = [\![open.flow = switch.In_3]\!] \wedge$
$\mathbb{I} = [\![switch.Out = accumulate.In_1]\!] \wedge \mathbb{I} = [\![accumulate.Out = volumeOut]\!]$

- The elementary block *delay* is modeled by the following TIC schema *tank_delay*, where the application $UnitDelay(1, 1)$ captures its sample time value and its initial output value. The connections among *plant*, *controller* and *delay* are specified in the schema *tank* below.

$$tank\_delay \mathrel{\widehat{=}} UnitDelay(1, 1)$$

tank

$delay : tank\_delay;\ plant : tank\_plant;\ controller : tank\_controller$

$\mathbb{I} = [\![delay.Out = plant.valve]\!] \wedge \mathbb{I} = [\![plant.volumeOut = plant.volumeIn]\!] \wedge$
$\mathbb{I} = [\![plant.volumeOut = controller.volume]\!] \wedge \mathbb{I} = [\![controller.valve = delay.In_1]\!]$

The tank control system is designed to satisfy several important requirements. For instance, the tank should never be overflowed or empty. Moreover, real-time constraints are imposed on the control system such as a response period of the valve. These requirements can be easily and precisely represented as TIC predicates based on the transformed TIC schemas over the whole system or some components. We here introduce two requirements which are used later to show how to formally conduct validation.

Requirement *Response* concerns the reaction time of the valve when the water volume is too high. Specifically, when a *left and right-closed* interval (denoted by $[\ ]$) during which the water volume is not lower than the value 3 (by $sys.plant.volumeOut \geq 3$) lasts more than 2 time units (by $\delta > 2$), *plant* should receive a command to open the valve (by $sys.plant.valve = 0$) within 2 time units in the interval. The receiving of an *open* command in time is captured by a *left-closed*, *right-open* interval, namely, $[\delta < 2)$, with the concatenation operator $\frown$ (introduced in Sect. 2.2) which connects two sets of intervals end-to-end.

$$Response == \forall sys : tank \bullet [sys.plant.volumeOut \geq 3 \wedge \delta > 2] \subseteq [\delta < 2) \frown [sys.plant.valve = 0]$$

Requirement *Safety* is about the safety of the tank system. It states that the water volume shall be always less than a maximal volume 4 in any non-empty interval.

$$Safety == \forall sys : tank \bullet \mathbb{I} = [\![sys.plant.volumeOut < 4]\!]$$

6.2. Validating system design against requirements

Validating systems modeled in Simulink against requirements is non-trivial, because these systems usually contain continuous dynamics and requirements often investigate behavior over arbitrary (infinite) intervals. After transforming Simulink diagrams to TIC schemas, we can apply well-defined TIC reasoning rules and mathematical laws (of arithmetic and calculus) to rigorously prove whether the TIC schemas logically imply TIC predicates which represent requirements.

To reduce the complexity of manual proofs in TIC, we exploit our existing verification system [CDS08] to support machine-assisted proofs. We have extended the verification system in Sect. 5.1 to support the automatic translation from axiomatic definitions to PVS functions, and to import the supplementary rules defined in Sect. 5.2 to the PVS prover. Using our framework, the main objective of a proof is usually to assign proper values to the quantified variables which represent intervals and time points where an assignment is often automatic, as resulting (propositional) predicates usually can be automatically discharged.

We sketch below the validation of the tank control system against two requirements described early, in order to demonstrate the capability of our framework on effectively conducting a formal analysis. Proof details are available online [CDS07a].

6.2.1. *Checking the requirement* Response

This requirement concerns the timing relation between the water volume sent from *plant* and commands received by *plant*. In other words, it involves only two components, namely, *delay* and *controller*. On one hand, *delay* updates its output at sample time hits, namely, every 1 time unit. On the other hand, *controller* continuously outputs a command to *delay*, and *Response* considers arbitrary *left and right-closed* intervals whose endpoints may not be sample time hits. We adopt the *proof by exhaustion* method to divide all non-empty intervals into finite cases to simplify the proof complexity.

We develop a theorem named *Endpoints_general_form* to classify any non-empty interval to one of four groups in terms of their endpoints; whether they are sample time hits or not. This theorem is shown below where ST is a sample time, and it has been validated in PVS.

$$Endpoints\_general\_form == \forall i : \mathbb{I} \bullet \exists m, p : \mathbb{N};\ n, q : \{0\} \cup \mathbb{R}^+ \mid n < ST \wedge q < ST \bullet \alpha(i) = m * ST + n \wedge \omega(i) = p * ST + q$$

In addition, for the sake of simplicity, we directly apply a pre-proved property of *controller*: *controller* outputs the value 0 when the input water volume is not less than the value 3. Hence, based on the wires specified in the schema *tank* and the substitution rule *BB_eq_sub* in Sect. 5.2, we can further narrow the checking of Simulink components down to one, the elementary block *delay* only.

$$Response == \forall sys : tank \bullet [sys.delay.In_1 = 0 \land \delta > 2] \subseteq [\delta < 2) \frown [sys.delay.Out = 0]$$

Next, we divide the checking of *Response* to four sub-proofs, and each sub-proof is concern with one group of intervals. We outline here the sub-proof for the group where each interval starts with not a sample time hit, namely, $1 > n > 0$ since $ST = 1$, and ends with a sample time hit, namely, $q = 0$.

1. The *skolemization* technique in PVS replaces universal quantifiers in consequent formulas by arbitrary (Skolem) constants. This technique is applied to *Response* to result in the following proof obligation, where the identifiers with a suffix ! are *Skolem* constants which are automatically generated by the PVS prover. The obligation is to construct two connected intervals $i2$ and $i3$, so $i2$ is *left-closed*, *right-open* and lasts less than 2 time units, while $i3$ is *left and right-closed* and *delay* outputs the value 0 in $i3$.

$$\begin{aligned} & i1! \in [sys!.delay.In_1 = 0 \land \delta > 2] \land \alpha(i1!) = m! + n! \land \omega(i1!) = p! \\ \Rightarrow\ & \exists i2, i3 : \mathbb{I} \bullet i2 \in [\delta < 2) \land i3 \in [sys!.delay.Out = 0] \land i1! = i2 \cup i3 \end{aligned}$$

2. As $i1!$ lasts longer than 2 time units, we instantiate $i2$ and $i3$ in the following way: $i2 = [m! + n! \ldots m! + 2)$ and $i3 = [m! + 2 \ldots p!]$. It is easy to see that $i2 \in [\delta < 2)$ and $i1! = i2 \cup i3$.
3. To prove $i3 \in [sys!.delay.Out = 0]$, we need to show $\forall t : [m! + 2 \ldots p!] \bullet sys!.delay.In_1(\lfloor \frac{t}{1} \rfloor - 1) = 0$ where $\lfloor\ \rfloor$ represents a mathematical *floor* function, because of the discrete behavior of *delay*. Note that the functionality of *delay* is also described in Sect. 3.2.
4. Using the mathematical laws of floor functions and real numbers, we can deduce the following relationship between an arbitrary time point t and the endpoints of $i1!$, specifically, $m! + n!$ and $p!$.

   - $\lfloor \frac{t}{1} - 1 \rfloor \geq \lfloor \frac{m!+2}{1} \rfloor - 1 = m! + 2 - 1 = m! + 1 > m! + n!$
   - $\lfloor \frac{t}{1} - 1 \rfloor \leq \lfloor \frac{p!}{1} \rfloor - 1 = p! - 1 \leq p!$

   From the hypothesis of $i1!$ at step 1, we can prove the proof goal at step 3 and hence complete the proof.

In the above procedure, it is difficult to automatically find out appropriate values for intervals $i2$ and $i3$ at step 2, because the time domain here is continuous. Human heuristics is helpful at this step to guide the PVS prover. On the other hand, the PVS prover facilitates the procedure by its support of the skolemization technique (at step 1) and the mathematical laws (at step 4); in addition, the reasoning of $i2$ at step 2 and the proof at step 4 are automatic.

Note that the above validation involves only the subsystem *controller* and the elementary block *delay*, which make up an open system excluding the subsystem *plant* that models the continuous behavior of the water volume. Supporting the analysis of open systems is useful in practice, because in Simulink (1) continuous behavior is hard to be depicted precisely and (2) open systems cannot be analyzed by means of simulations. We have investigated in [CDS07b] the applicability of our approach on handling open systems by specifying environment constraints in TIC.

6.2.2. *Checking the requirement* Safety

To simplify the checking process, we apply a refined TIC reasoning rule from our previous work [CDS08] to check *Safety* for only *left and right-closed* intervals rather than all non-empty intervals. The reasoning rule $[\neg P] = \varnothing \Rightarrow \mathbb{I} = [\![P]\!]$ states that a predicate P holds in all non-empty intervals provided there exists no *left and right-closed* interval where the negation of P (namely, $\neg P$) holds and P is irrelevant to interval operators. Thus, we can have $Safety == \forall sys : tank \bullet [sys.plant.volumeOut \geq 4] = \varnothing$.

Instead of checking the invalidity of *Safety* for all *left and right-closed* intervals, we exploit the *proof by contradiction* method to reduce the number of intervals needed to be examined to one. We check if there is a contradiction when we presume that the water volume is not less than the value 4 during a *left and right-closed* interval i, namely, $i \in [sys.plant.volumeOut \geq 4]$.

As *volumeOut* is continuous due to the continuous block *accumulate*, we can apply a theorem developed by us. The theorem models a property of continuous functions at the interval level: for a continuous timed trace

tr and a threshold TH, an interval j, where $tr(\alpha(j)) < TH$ and $tr(\omega(j)) > TH$, can be decomposed into two connected intervals $j1$ and $j2$, where $j = j1 \cup j2 \wedge tr(\alpha(j2)) = TH \wedge \forall t : j2 \bullet tr(t) \geq TH$. This theorem has also been validated in PVS.

We assign *sys.plant.volumeOut* to tr and the value 3 to TH, and then infer that the interval $[0 \ldots \alpha(i)]$ can be decomposed to two connected intervals, where the latter interval $i1$ is *left and right-closed* and the water volume equals 3 at its starting point and is not less than 3 elsewhere. Note that the initial water volume is 2. Namely, $i1 \in [sys.plant.volumeOut(\alpha) = 3 \wedge sys.plant.volumeOut \geq 3] \wedge \omega(i1) = \alpha(i)$

We hence analyze the water volume *volumeOut* at the ending point of the interval $i1$ as $\omega(i1) = \alpha(i)$. The analysis is divided to two cases according to the length of $i1$.

- *When* $\delta(i1) \leq 2$*:* Based on the connections in *plant*, particularly blocks *switch* and *accumulate* and port *volumeOut*, we can obtain the following integral equation for computing the water volume.

$$\begin{aligned} sys.plant.volumeOut(\omega(i1)) &= sys.plant.volumeOut(\alpha(i1)) + \int_{\alpha(i1)}^{\omega(i1)} sys.plant.switch.Out \\ &= 3 + \int_{\alpha(i1)}^{\omega(i1)} sys.plant.switch.Out \end{aligned}$$

  The output of *switch* depends on its second input which denotes the valve status as shown below.

  – When the valve is opened at a time point t, *switch* outputs its third input connected by the subsystem *open*. We can have $sys.plant.switch.Out(t) = -0.1 * sys.plant.volumeOut(t) \leq 0$, as the water volume is never negative.
  – When the valve is closed at a time point t, *switch* outputs its first input connected by the subsystem *close*. We can have $sys.plant.switch.Out(t) = -0.1 * (sys.plant.volumeOut(t) - 6) \leq 0.3$, because $i1 \in [sys.plant.volumeOut \geq 3]$.

  By a mathematical law (from the NASA PVS library) which relates *integration bounds* of an integrated function and *bounds* of the integrated function, we can compute the upper limit of the water volume at the ending point of $i1$. Note that $\delta(i1) = \omega(i1) - \alpha(i1)$.

$$sys.plant.volumeOut(\omega(i1)) = 3 + \int_{\alpha(i1)}^{\omega(i1)} sys.plant.switch.Out \leq 3 + 2 * 0.3 < 4$$

  Therefore, we find out a contradiction that the values of the water volume at the same time point, $\omega(i1)$ and $\alpha(i)$, are inconsistent, as $sys.plan.volumeOut(\alpha(i)) \geq 4$.

- *When* $\delta(i1) > 2$*:* From the requirement *Response*, we can derive that there exist two intervals $i2$ and $i3$ which compose $i1$ and satisfy the following properties.

$$\begin{aligned} &i2 \in [sys.plant.volumeOut(\alpha) = 3 \wedge sys.plant.volumeOut \geq 3 \wedge \delta < 2) \\ &i3 \in [sys.plant.volumeOut \geq 3 \wedge sys.plant.valve = 0] \wedge \omega(i3) = \omega(i1) = \alpha(i) \end{aligned}$$

  Firstly, it is easy to prove that $i2 \in [sys.plant.volumeOut < 4)$ from the proof of the first case where $\delta(i1) \leq 2$. To be specific, we can replace the ending point $\omega(i1)$ in the previous proof by an arbitrary time point of $i2$ and follow similarly the reasoning process.
  Next, to analyze the *volumeOut* value at the end of $i3$, we use a proved property of the subsystem *plant*, which indicates that the water volume decreases in an interval in which the valve is open. Hence, we can deduce that $sys.plant.volumeOut(\omega(i3)) \leq sys.plant.volumeOut(\alpha(i3))$.
  Since the water volume is less than the value 4 at the end of $i2$ from the first case, we can imply that $sys.plant.volumeOut(\omega(i3)) < 4$. That is to say, we find out another contradiction.

We therefore show that a contradiction exists in both cases. Namely, there is no *left and right-closed* interval in which the water volume can be larger than or equals to 4. We hence complete the proof. We remark that the analysis of continuous behavior (for instance, the water volume) is supported in our framework.

In this section, we illustrated an application of our framework to the tank control system whose behavior includes continuous-time, discrete-time, and discrete logics. The framework can formally model various behaviors of the control system, and assist rigorous validation of a bounded liveness requirement and a safe requirement. The validation is beyond Simulink and is systematically conducted with a high degree of automation and the support for analyzing continuous dynamics.

7. Conclusion

In this article, we developed a framework to formally model and rigorously validate Simulink diagrams. This framework captures the functional and timing aspects of Simulink diagrams, and supports the validation of the systems denoted in Simulink against important requirements with a high grade of automation.

We elaborately constructed a set of TIC library functions to represent frequently used Simulink library blocks. These library functions model the time-dependent mathematical relationships between block inputs and outputs, and they were checked to conform to the behavior of those library blocks by thorough simulation. From the rigorous procedure of constructing and validating these library functions, we discovered incomplete semantics and a bug of those library blocks in the original Simulink documentation.

Based on the TIC library functions, we implemented a translator in Java to automatically transform Simulink diagrams to TIC schemas in the bottom-up order. The transformation can calculate all derivable sample times which are the timing information of elementary blocks in Simulink diagrams. *Enabled* subsystems and *triggered* subsystems, are also supported. We presented our solution to systematically handle these conditionally executed subsystems according to their control inputs and system inputs. After the transformation, we can precisely and concisely specify in TIC various (timing) requirements of a system or some components, and specify environment properties for open systems as well.

We enhanced our existing verification system used in the framework to automatically translate axiomatic definitions to PVS functions, and to include the supplementary rules dedicated to Simulink modeling features. Using this framework, we can hence rigorously carry out the validation of complex systems which may possess continuous and discrete behavior, with a high level of automation (for instance, arithmetic reasoning is automatic) and the support of analyzing continuous dynamics (for example, the mathematical laws of integration) and common proof methods (such as proof by contradiction and proof by induction).

There are several directions on extending our framework. One is to support more Simulink library blocks. Currently we support 51 library blocks from 10 categories including *continuous*, *discrete*, *signal routing*, etc, which cover all library blocks of the *Commonly Used* category in Simulink [Mat08a]. Compared to other existing works, our approach handles more library blocks and some of them are only supported by ours, in particular, library blocks of the *Continuous* category. We are inspired by the TIC expressiveness and aim to model all mathematical relationships denoted by library blocks. Another direction is to improve the automation of the validation of Simulink diagrams. Though verification of complex Simulink diagrams is challenging, we are developing more supplementary rules dedicated to specific domains (such as hybrid control systems, the primary domain of Simulink modeling) to simplify reasoning processes. We also plan to expand the framework to support formal analysis of Stateflow diagrams by using PAT [LSD08, SLDW08], a toolkit supporting an expressive modeling language and state-of-art model checking techniques. Applying our framework to industrial-scale case studies is also one of our goals.

Acknowledgment

The preliminary work was presented in the Doctoral Symposium of Formal Methods 2006, and we are grateful for the valuable comments from the Examination Panel of the Doctoral Symposium. We thank Anders P. Ravn, Chaochen Zhou, Mark Adams and Jeremy Dawson for their insightful discussion on related work. We also appreciate Chang Duan and Wolff Greg from the MathWorks for their assistance in using Simulink. This work is supported by the project "Formal Design Techniques for Reactive Embedded Systems" from Singapore A*Star Research Grants.

Appendix A: Supported Simulink library blocks

A.1. Simulink library blocks modeled in TIC library functions

- Continuous Library: Integrator, Derivative.
- Discrete Library: Memory, *Discrete-Time Integrator*, Unit Delay, Zero-order Delay.
- Logic and Bits Operations Library: *Combinational Logic*, *Comparator to Constant*, *Compare to Zero*, Interval Test, Logical Operator, Relational Operator.

- Math Operations Library: Abs, Add, *Bias*, Divide, Dot Product, *Fcn*, Gain, *Math Function*, *MinMax*, Product, *Sign*, Subtract, Sum, Unary Minus.
- Discontinuous Library: Dead Zone, *Hit Crossing*, *Relay*, Saturation.
- Signal Routing Library: *Bus Creator*, *Bus Selector*, *Demux*, Mux, Switch.
- Source Library: Constant, Clock, Digital Clock, *Ground*.
- Signal Attributes Library: *Data Type Conversion*, IC.
- Sinks: *Display*, *Scope*, *Terminator*.

A.2. Simulink library blocks handled in the transformation

- Ports and Subsystems: Enable, Enabled Subsystem, Inport, Outport, Subsystem, Trigger, Triggered Subsystem.

Appendix B: TIC models of the tank control system

B.1. TIC library functions of the Simulink library blocks in the tank control system

Here we provide the TIC library functions which model the library blocks used in Fig. 7. Note that the TIC library functions *Integrator*, *UnitDelay* and *Sum_PM* for blocks *accumulate*, *delay* and *diff* have been specified in Sect. 3.2.

$$
\begin{array}{|l}
Constant : \mathbb{R} \to \mathbb{P}[Out : \mathbb{T} \to \mathbb{R};\ IniVal : \mathbb{R}] \\
\hline
\forall cv : \mathbb{R} \bullet Constant(cv) = [Out : \mathbb{T} \to \mathbb{R};\ IniVal : \mathbb{R} \mid cv = IniVal \wedge \mathbb{I} = [\![Out = IniVal]\!]\,]
\end{array}
$$

$$
\begin{array}{|l}
Comparator\_l : \mathbb{T} \times \mathbb{R} \to \mathbb{P}[In_1 : \mathbb{T} \to \mathbb{R};\ Out : \mathbb{T} \to \{0,1\};\ st : \mathbb{T};\ TH : \mathbb{R}] \\
\hline
\forall t : \mathbb{T};\ p : \mathbb{R} \bullet (t = 0 \Rightarrow Comparator\_l(t,p) = [In_1 : \mathbb{T} \to \mathbb{R};\ Out : \mathbb{T} \to \{0,1\};\ st : \mathbb{T};\ TH : \mathbb{R} \mid \\
\qquad st = 0 \wedge p = TH \wedge [\![In_1 < TH]\!] = [\![Out = 1]\!] \wedge [\![In_1 \geq TH]\!] = [\![Out = 0]\!]\,]) \\
\wedge \quad (t > 0 \Rightarrow Comparator\_l(t,p) = [In_1 : \mathbb{T} \to \mathbb{R};\ Out : \mathbb{T} \to \{0,1\};\ st : \mathbb{T};\ TH : \mathbb{R} \mid \\
\qquad t = st \wedge st > 0 \wedge p = TH \wedge [\!(\exists k : \mathbb{N} \bullet \alpha = k * st \wedge \omega = (k+1) * st)\!) \subseteq \\
\qquad\qquad [\!(In_1(\alpha) < TH \Rightarrow Out = 1) \wedge (In_1(\alpha) \geq TH \Rightarrow Out = 0))\!])
\end{array}
$$

$$
\begin{array}{|l}
Comparator\_g : \mathbb{T} \times \mathbb{R} \to \mathbb{P}[In_1 : \mathbb{T} \to \mathbb{R};\ Out : \mathbb{T} \to \{0,1\};\ st : \mathbb{T};\ TH : \mathbb{R}] \\
\hline
\forall t : \mathbb{T};\ p : \mathbb{R} \bullet (t = 0 \Rightarrow Comparator\_g(t,p) = [In_1 : \mathbb{T} \to \mathbb{R};\ Out : \mathbb{T} \to \{0,1\};\ st : \mathbb{T};\ TH : \mathbb{R} \mid \\
\qquad st = 0 \wedge p = TH \wedge [\![In_1 > TH]\!] = [\![Out = 1]\!] \wedge [\![In_1 \leq TH]\!] = [\![Out = 0]\!]\,]) \\
\wedge \quad (t > 0 \Rightarrow Comparator\_g(t,p) = [In_1 : \mathbb{T} \to \mathbb{R};\ Out : \mathbb{T} \to \{0,1\};\ st : \mathbb{T};\ TH : \mathbb{R} \mid \\
\qquad t = st \wedge st > 0 \wedge p = TH \wedge [\!(\exists k : \mathbb{N} \bullet \alpha = k * st \wedge \omega = (k+1) * st)\!) \subseteq \\
\qquad\qquad [\!(In_1(\alpha) > TH \Rightarrow Out = 1) \wedge (In_1(\alpha) \leq TH \Rightarrow Out = 0))\!])
\end{array}
$$

$$
\begin{array}{|l}
Switch\_G : \mathbb{T} \times \mathbb{R} \to \mathbb{P}[In_1, In_2, In_3, Out : \mathbb{T} \to \mathbb{R};\ TH : \mathbb{R};\ st : \mathbb{T}] \\
\hline
\forall t : \mathbb{T};\ th : \mathbb{R} \bullet (t = 0 \Rightarrow Switch\_G(t,th) = [In_1, In_2, In_3, Out : \mathbb{T} \to \mathbb{R};\ TH : \mathbb{R};\ st : \mathbb{T} \mid \\
\qquad st = 0 \wedge th = TH \wedge [\![In_2 > TH]\!] = [\![Out = In_1]\!] \wedge [\![In_2 \leq TH]\!] = [\![Out = In_3]\!]\,]) \\
\wedge \quad (t > 0 \Rightarrow Switch\_G(t,th) = [In_1, In_2, In_3, Out : \mathbb{T} \to \mathbb{R};\ TH : \mathbb{R};\ st : \mathbb{T} \mid \\
\qquad t = st \wedge st > 0 \wedge th = TH \wedge [\!(\exists k : \mathbb{N} \bullet \alpha = k * st \wedge \omega = (k+1) * st)\!) \subseteq \\
\qquad\qquad [\!(In_2(\alpha) > TH \Rightarrow Out = In_1(\alpha)) \wedge (In_2(\alpha) \leq TH \Rightarrow Out = In_3(\alpha)))\!]\,])
\end{array}
$$

$$
\begin{array}{l}
InitCond : \mathbb{T} \times \mathbb{R} \to \mathbb{P}[In_1, Out : \mathbb{T} \to \mathbb{R};\ IniVal : \mathbb{R};\ st : \mathbb{T}] \\
\hline
\forall t : \mathbb{T};\ init : \mathbb{R} \bullet (t = 0 \Rightarrow InitCond(t, init) = [In_1, Out : \mathbb{T} \to \mathbb{R};\ IniVal : \mathbb{R};\ st : \mathbb{T} \mid \\
\qquad st = 0 \land init = IniVal \land [\![0 = \alpha]\!] \subseteq [\![Out(\alpha) = IniVal]\!] \land [\![0 < \alpha]\!] \subseteq [\![Out = In_1]\!]\,]) \\
\quad \land \quad (t > 0 \Rightarrow InitCond(t, init) = [In_1, Out : \mathbb{T} \to \mathbb{R};\ IniVal : \mathbb{R};\ st : \mathbb{T} \mid \\
\qquad t = st \land st > 0 \land init = IniVal \land (\!|\alpha = 0 \land \omega = st|\!) = (\!|Out = IniVal|\!) \land \\
\qquad (\!|\exists k : \mathbb{N}_1 \bullet \alpha = k * st \land \omega = (k + 1) * st|\!) \subseteq (\!|Out = In_1(\alpha)|\!)\,])
\end{array}
$$

$$
\begin{array}{l}
Logic\_NOT : \mathbb{T} \to \mathbb{P}[In_1 : \mathbb{T} \to \mathbb{R};\ Out : \mathbb{T} \to \{0, 1\};\ st : \mathbb{T}] \\
\hline
\forall t : \mathbb{T} \bullet (t = 0 \Rightarrow Logic\_NOT(t) = [In_1 : \mathbb{T} \to \mathbb{R};\ Out : \mathbb{T} \to \{0, 1\};\ st : \mathbb{T} \mid \\
\qquad st = 0 \land [\![In_1 = 0]\!] = [\![Out = 1]\!] \land [\![In_1 \neq 0]\!] = [\![Out = 0]\!]\,]) \\
\quad \land \quad (t > 0 \Rightarrow Logic\_NOT(t) = [In_1 : \mathbb{T} \to \mathbb{R};\ Out : \mathbb{T} \to \{0, 1\};\ st : \mathbb{T} \mid \\
\qquad t = st \land st > 0 \land (\!|\exists k : \mathbb{N} \bullet \alpha = k * st \land \omega = (k + 1) * st|\!) \subseteq \\
\qquad\qquad (\!|(In_1(\alpha) = 0 \Rightarrow Out = 1) \land (In_1(\alpha) \neq 0 \Rightarrow Out = 0)|\!)\,])
\end{array}
$$

$$
\begin{array}{l}
Gain : \mathbb{T} \times \mathbb{R} \to \mathbb{P}[In_1, Out : \mathbb{T} \to \mathbb{R};\ GValue : \mathbb{R};\ st : \mathbb{T}] \\
\hline
\forall t : \mathbb{T};\ gv : \mathbb{R} \bullet (t = 0 \Rightarrow Gain(t, gv) = [In_1, Out : \mathbb{T} \to \mathbb{R};\ GValue : \mathbb{R};\ st : \mathbb{T} \mid \\
\qquad st = 0 \land gv = GValue \land \mathbb{I} = [\![Out = In_1 * GValue]\!]\,]) \\
\quad \land \quad (t > 0 \Rightarrow Gain(t, gv) = [In_1, Out : \mathbb{T} \to \mathbb{R};\ GValue : \mathbb{R};\ st : \mathbb{T} \mid \\
\qquad t = st \land st > 0 \land gv = GValue \land \\
\qquad (\!|\exists k : \mathbb{N} \bullet \alpha = k * st \land \omega = (k + 1) * st|\!) \subseteq (\!|Out = In_1(\alpha) * GValue|\!)\,])
\end{array}
$$

B.2. Transformed TIC schemas of the Simulink diagrams for the tank control system

The following TIC schemas model all subsystems of the tank control system. The schemas for the whole system have been represented in Sect. 6.1.

B.2.1. Subsystem controller

$$
\begin{array}{ll}
tank\_controller\_max \mathrel{\widehat{=}} Comparator\_l(0, 3) & tank\_controller\_diff \mathrel{\widehat{=}} Sum\_PM(0) \\
tank\_controller\_min \mathrel{\widehat{=}} Comparator\_g(0, 1) & tank\_controller\_switch \mathrel{\widehat{=}} Switch\_G(0, 0) \\
tank\_controller\_constant \mathrel{\widehat{=}} Constant(1) & tank\_controller\_initial \mathrel{\widehat{=}} InitCond(0, 1)
\end{array}
$$

tank_controller

$$
\begin{array}{l}
volume, valve : \mathbb{T} \nrightarrow \mathbb{R};\ max : tank\_controller\_max;\ min : tank\_controller\_min \\
constant : tank\_controller\_constant;\ diff : tank\_controller\_diff \\
switch : tank\_controller\_switch;\ initial : tank\_controller\_initial \\
\hline
\mathbb{I} = [\![volume = max.In_1]\!] \land \mathbb{I} = [\![volume = min.In_1]\!] \land \mathbb{I} = [\![initial.Out = valve]\!] \land \\
\mathbb{I} = [\![max.Out = switch.In_1]\!] \land \mathbb{I} = [\![min.Out = diff.In_2]\!] \land \\
\mathbb{I} = [\![constant.Out = diff.In_1]\!] \land \mathbb{I} = [\![diff.Out = switch.In_3]\!] \land \\
\mathbb{I} = [\![initial.Out = switch.In_2]\!] \land \mathbb{I} = [\![switch.Out = initial.In_1]\!]
\end{array}
$$

B.2.2. Subsystem open

$$
tank\_plant\_open\_K \mathrel{\widehat{=}} Gain(0, -0.1)
$$

tank_plant_open

$$
\begin{array}{l}
Enable : \mathbb{T} \to \mathbb{R};\ volumeIn, flow : \mathbb{T} \nrightarrow \mathbb{R};\ K : tank\_plant\_open\_K \\
\hline
\mathbb{I} = [\![volumeIn = K.In_1]\!] \land \mathbb{I} = [\![K.Out = flow]\!]
\end{array}
$$

B.2.3. Subsystem close

$tank\_plant\_close\_constant \mathrel{\widehat=} Constant(6)$ $tank\_plant\_close\_K \mathrel{\widehat=} Gain(0, -0.1)$
$tank\_plant\_close\_sum \mathrel{\widehat=} Sum\_PM(0)$

tank_plant_close

$Enable : \mathbb{T} \to \mathbb{R};\ volumeIn, flow : \mathbb{T} \pfun \mathbb{R};\ K : tank\_plant\_close\_K$
$constant : tank\_plant\_close\_constant;\ sum : tank\_plant\_close\_sum$

$\mathbb{I} = [\![volumeIn = sum.In_1]\!] \land \mathbb{I} = [\![constant.Out = sum.In_2]\!]$
$\mathbb{I} = [\![sum.Out = K.In_1]\!] \land \mathbb{I} = [\![K.Out = flow]\!]$

B.2.4. Subsystem plant

$tank\_plant\_inverse \mathrel{\widehat=} Logic\_NOT(1)$ $tank\_plant\_accumulate \mathrel{\widehat=} Integrator(2)$
$tank\_plant\_switch \mathrel{\widehat=} Switch\_G(0, 0)$

tank_plant

$volumeIn, volumeOut : \mathbb{T} \pfun \mathbb{R};\ valve : \mathbb{T} \to \mathbb{R};\ inverse : tank\_plant\_inverse$
$switch : tank\_plant\_switch;\ accumulate : tank\_plant\_accumulate$
$close : tank\_plant\_close;\ open : tank\_plant\_open$

$\{open.Enable \le 0 \land open.Enable(\omega) > 0 \land \alpha = 0 \land \omega = 1\} \subseteq \{open.volumeIn = 0\}$
$\{open.Enable \le 0 \land open.Enable(\omega) \le 0 \land \alpha = 0 \land \omega = 1\}$
$\quad \subseteq \{open.volumeIn = 0 \land open.volumeIn(\omega) = 0\}$
$\{open.Enable \le 0 \land open.Enable(\omega) > 0 \land \exists k : \mathbb{N}_1 \bullet \alpha = k \land \omega = k + 1\}$
$\quad \subseteq \{open.volumeIn(\alpha) = open.volumeIn\}$
$\{open.Enable \le 0 \land open.Enable(\omega) \le 0 \land \exists k : \mathbb{N}_1 \bullet \alpha = k \land \omega = k + 1\}$
$\quad \subseteq \{open.volumeIn(\alpha) = open.volumeIn \land open.volumeIn(\alpha) = open.volumeIn(\omega)\}$
$\{open.Enable > 0 \land open.Enable(\omega) > 0 \land \exists k : \mathbb{N} \bullet \alpha = k \land \omega = k + 1\}$
$\quad \subseteq \{volumeIn = open.volumeIn\}$
$\{open.Enable > 0 \land open.Enable(\omega) \le 0 \land \exists k : \mathbb{N} \bullet \alpha = k \land \omega = k + 1\}$
$\quad \subseteq \{volumeIn = open.volumeIn \land volumeIn(\omega) = open.volumeIn(\omega)\}$
$\{close.Enable \le 0 \land close.Enable(\omega) > 0 \land \alpha = 0 \land \omega = 1\} \subseteq \{close.volumeIn = 0\}$
$\{close.Enable \le 0 \land close.Enable(\omega) \le 0 \land \alpha = 0 \land \omega = 1\}$
$\quad \subseteq \{close.volumeIn = 0 \land close.volumeIn(\omega) = 0\}$
$\{close.Enable \le 0 \land close.Enable(\omega) > 0 \land \exists k : \mathbb{N}_1 \bullet \alpha = k \land \omega = k + 1\}$
$\quad \subseteq \{close.volumeIn(\alpha) = close.volumeIn\}$
$\{close.Enable \le 0 \land close.Enable(\omega) \le 0 \land \exists k : \mathbb{N}_1 \bullet \alpha = k \land \omega = k + 1\}$
$\quad \subseteq \{close.volumeIn(\alpha) = close.volumeIn \land close.volumeIn(\alpha) = close.volumeIn(\omega)\}$
$\{close.Enable > 0 \land close.Enable(\omega) > 0 \land \exists k : \mathbb{N} \bullet \alpha = k \land \omega = k + 1\}$
$\quad \subseteq \{volumeIn = close.volumeIn\}$
$\{close.Enable > 0 \land close.Enable(\omega) \le 0 \land \exists k : \mathbb{N} \bullet \alpha = k \land \omega = k + 1\}$
$\quad \subseteq \{volumeIn = close.volumeIn \land volumeIn(\omega) = close.volumeIn(\omega)\}$
$\mathbb{I} = [\![volumeIn = close.volumeIn]\!] \land \mathbb{I} = [\![volumeIn = open.volumeIn]\!]$
$\mathbb{I} = [\![valve = close.Enable]\!] \land \mathbb{I} = [\![valve = inverse.In_1]\!]$
$\mathbb{I} = [\![valve = switch.In_2]\!] \land \mathbb{I} = [\![close.flow = switch.In_1]\!]$
$\mathbb{I} = [\![inverse.Out = open.Enable]\!] \land \mathbb{I} = [\![open.flow = switch.In_3]\!]$
$\mathbb{I} = [\![switch.Out = accumulate.In_1]\!] \land \mathbb{I} = [\![accumulate.Out = volumeOut]\!]$

Appendix C: Handling conditionally executed subsystems

This appendix is complementary to Sects. 4.5.1 and 4.5.2 to deal with two additional types of *enabled* subsystems and *triggered* subsystems. One is for triggered subsystems with discrete control inputs, and the other is for enabled subsystems with continuous control inputs.

C.1. Triggered subsystems of discrete control inputs

When the control input of a triggered subsystem is discrete, trigger events occurs only at sample time hits. In addition, there is no trigger event at time point 0 as the input is constant in the initial sample time interval. Note that discrete behavior in Simulink is piecewise-constantly continuous. We specify the behavior by constraining the values of subsystem inputs in terms of sample time intervals, which are *left-closed, right-open* and formed by a pair of consecutive sample time hits. We remark that triggered subsystems output the last value between any two events. As subsystem outputs are determined by subsystem inputs, it is thus necessary and important to mode the way of assigning subsystem inputs. Specifically the value of a subsystem input can come from the block which is outside the subsystem and connects to the input or be the last value which is obtained at the time point when the last event happens. Moreover, the type of trigger events varies the kinds of situations to be handled. Particularly, according to the occurrences of trigger events at both endpoints of any sample time interval, if the type is *either*, there are six kinds of situations relevant to the assignment of input values; else the type is either *rising* or *falling*, and there are five kinds of situations since it is impossible that two events occur at a pair of sample time hits.

We take a simple system shown in Fig. 8 as an example. The control input of the triggered subsystem *trigsys* is connected by a source which outputs discretely, every 1 time unit. The type of trigger events is *either*. Namely, a trigger event occurs when the control input rises from a negative or zero value to a positive value or the control input falls from a positive or a zero value to a negative value.

The schema *sys_trigsys* shown below denotes the subsystem *trigsys*: the first predicate constrains that there is no trigger event at the time point 0; the second predicate captures that the time points where trigger events can occur are multiples of the sample time which is 1 in this example.

sys_trigsys

$$Trigger : \mathbb{T} \to \{0, 1\};\ In1, Out1 : \mathbb{T} \to \mathbb{R}$$

$$[Trigger = 0] \subseteq [\alpha = 0 \land \omega = 0] \land [Trigger = 1] \subseteq [\exists k : \mathbb{N}_1 \bullet \alpha = k \land \alpha = \omega] \land \mathbb{I} = [\![In1 = Out1]\!]$$

sys

$$In1 : \mathbb{T} \to \mathbb{R};\ trigsys : sys\_trigsys;\ \ldots$$

$$\ldots$$

$$[trigsys.Trigger(\omega) = 0 \land \alpha = 0 \land \omega = 1) \subseteq [trigsys.In1 = 0 \land trigsys.In1(\omega) = 0) \quad \text{[Predicate1]}$$

$$[trigsys.Trigger(\omega) = 1 \land \alpha = 0 \land \omega = 1) \subseteq [trigsys.In1 = 0) \quad \text{[Predicate2]}$$

$$[trigsys.Trigger(\alpha) = 1 \land trigsys.Trigger(\omega) = 0 \land \exists k : \mathbb{N}_1 \bullet \alpha = k \land \omega = k + 1) \subseteq [In1(\alpha) = trigsys.In1 \land In1(\alpha) = trigsys.In1(\omega)) \quad \text{[Predicate3]}$$

$$[trigsys.Trigger(\alpha) = 1 \land trigsys.Trigger(\omega) = 1 \land \exists k : \mathbb{N}_1 \bullet \alpha = k \land \omega = k + 1) \subseteq [In1(\alpha) = trigsys.In1) \quad \text{[Predicate4]}$$

$$[trigsys.Trigger(\alpha) = 0 \land trigsys.Trigger(\omega) = 0 \land \exists k : \mathbb{N}_1 \bullet \alpha = k \land \omega = k + 1) \subseteq [trigsys.In1(\alpha) = trigsys.In1 \land trigsys.In1(\alpha) = trigsys.In1(\omega)) \quad \text{[Predicate5]}$$

$$[trigsys.Trigger(\alpha) = 0 \land trigsys.Trigger(\omega) = 1 \land \exists k : \mathbb{N}_1 \bullet \alpha = k \land \omega = k + 1) \subseteq [trigsys.In1(\alpha) = trigsys.In1) \quad \text{[Predicate6]}$$

The above part of the schema *sys* represents the conditional execution of *trigsys* by six predicates. These predicates model the way to assign the subsystem input *trigsys.In*1 based on whether an event occurs at any ending points of every sample time interval, namely, checking $trigsys.Trigger(\alpha)$ and $trigsys.Trigger(\omega)$. Predicate1 and Predicate2 are concerned with the initial sample time interval: the default value of *trigsys.In*1 is 0 during the interval; and if no event happens at the ending point, the value 0 is assigned to *trigsys.In*1 at the ending point (expressed by Predicate1). The last four predicates deal with non-initial sample time intervals (by $[\exists k : \mathbb{N}_1 \bullet \alpha = k \land \omega = k + 1)$). Predicate4 states that when events occur at both ending points, the value of the

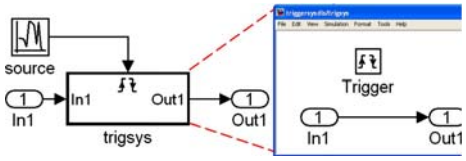


Fig. 8. A triggered subsystem *trigsys* controlled by a discrete input

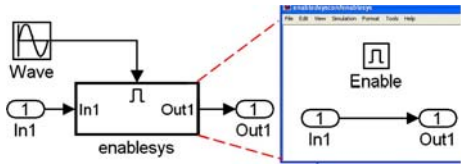


Fig. 9. An enabled subsystem *enablesys* which is controlled by a continuous input

input port at the starting point ($In1(\alpha)$) is the last value to $trigsys.In1$ in the interval; moreover, if no event occurs at the ending point, one more constraint is added to assign the last value to $trigsys.In1$ at the ending point (by Predicate3). When no event occurs at the starting point but one event at the ending point, the last value during the interval is the input value at the starting point (by Predicate6); furthermore, if no event occurs at the ending point, we need to also assign the last value to $trigsys.In1$ at the ending point (by Predicate5).

C.2. Enabled subsystems of continuous control inputs

When the control input of an enabled subsystem is continuous, the subsystem executes whenever the value of the input is positive. Here we handle the case that enabled subsystems outputs the most recent values when it is disabled. To model the conditional execution, we use a similar method which has been applied for triggered subsystems to specify how to assign subsystem inputs appropriate values in two circumstances, namely, enabled and disabled. We further restrict that the intervals during which the control input values are positive are left and right-closed.

For example, Fig. 9 shows an enabled subsystem *enablesys* which is controlled by a continuous wave. This continuity feature is captured in the schema *sys_enablesys* which denotes the subsystem *enablesys*, specifically, by the symbol $\twoheadrightarrow$ in the declaration of the control input *Enable*.

$$sys\_enablesys \mathrel{\widehat=} [Enable : \mathbb{T} \twoheadrightarrow \mathbb{R};\ In1, Out1 : \mathbb{T} \to \mathbb{R} \mid \mathbb{I} = [\![In1 = Out1]\!]]$$

Part of the following schema *sys* specifies the conditional behavior in three TIC predicates. Predicate1 indicates that whenever the subsystem is enabled, the value of the subsystem input $enablesys.In1$ is assigned by the input port $In1$ which connects the subsystem input. Predicate2 and Predicate3 are concerned with the situation where the subsystem is enabled ($enablesys.Enable > 0$). Specifically, if the interval during which the subsystem is disabled starts with time point 0, the value of $enablesys.In1$ is 0 by default (expressed by Predicate2); else the interval starts with positive time point, and we assign the value of $enablesys.In1$ at the starting point as the last value within the interval (by Predicate3). Note that the reason for choosing the last value is similar to the one for handling triggered subsystems (as discussed in Sect. 4.5.1).

sys

$In1 : \mathbb{T} \twoheadrightarrow \mathbb{R};\ enablesys : sys\_enablesys;\ \ldots$

$\ldots$

$[\![enablesys.Enable > 0]\!] \subseteq [\![In1 = enablesys.In1]\!]$ [Predicate1]

$[\![enablesys.Enable \leq 0 \land \alpha = 0]\!] \subseteq [\![enablesys.In1 = 0]\!]$ [Predicate2]

$[\![enablesys.Enable \leq 0 \land \alpha > 0]\!] \subseteq [\![enablesys.In1(\alpha) = enablesys.In1]\!]$ [Predicate3]

References

[AC05] Adams MM, Clayton PB (2005) ClawZ: cost-effective formal verification for control systems. In: Proceedings of the 7th international conference on formal engineering methods. Springer, Heidelberg, pp 465–479

[ACOS00] Arthan R, Caseley P, O'Halloran C, Smith A (2000) Clawz: control laws in Z. In: Proceedings of the 3rd international conference on formal engineering methods. IEEE Computer Society, Washington, pp 169–176

[But04] Butler RW (2004) Formalization of the integral calculus in the PVS theorem prover. Technical report, NASA Langley Research Center, Hampton, Virginia

[CCO05] Cavalcanti A, Clayton P, O'Halloran C (2005) Control law diagrams in Circus. In: Proceedings of the 13th international symposium of formal methods europe. Springer, Heidelberg, pp 253–268

[CD06] Chen C, Dong JS (2006) Applying timed interval calculus to simulink diagrams. In: Proceedings of the 8th international conference on formal engineering methods. Springer, Heidelberg, pp 74–93

[CDS07a] Chen C, Dong JS, Sun J (2007) A formal framework for modeling and verifying simulink diagrams. http://www.comp.nus.edu.sg/~chenchun/SimInTIC

[CDS07b] Chen C, Dong JS, Sun J (2007) Machine-assisted proof support for validation beyond Simulink. In: Proceedings of the 9th international conference on formal engineering methods. Springer, Heidelberg, pp 96–115

[CDS08] Chen C, Dong JS, Sun J (2008) A verification system for timed interval calculus. In: Proceedings of the 30th international conference on software engineering. ACM, New York, pp 271–280

[CSW03] Cavalcanti A, Sampaio A, Woodcock J (2003) A refinement strategy for Circus. Formal Asp Comput 15(2–3):146–181

[FHM98] Fidge CJ, Hayes IJ, Mahony BP (1998) Defining differentiation and integration in Z. In: Proceedings of the 2nd international conference on formal engineering methods. IEEE Computer Society, Washington, pp 64–73

[FHMW98] Fidge CJ, Hayes IJ, Martin AP, Wabenhorst A (1998) A set-theoretic model for real-time specification and reasoning. In: Proceedings of the mathematics of program construction. Springer, Heidelberg, pp 188–206

[GKR04] Gupta S, Krogh BH, Rutenbar RA (2004) Towards formal verification of analog designs. In: proceedings of the international conference on computer-aided design. IEEE Computer Science, Washington, pp 210–217

[HS06] Henzinger TA, Sifakis J (2006) The embedded systems design challenge. In Proceedings of the 14th international symposium on formal methods. Springer, Heidelberg, pp 1–15

[JCZE00] Jersak M, Cai Y, Ziegenbein D, Ernst R (2000) A transformational approach to constraint relaxation of a time-driven simulation model. In: Proceedings of the 13th international symposium on System synthesis. IEEE Computer Society, Washington, pp 137–142

[JS05] Jantsch A, Sander I (2005) Models of computation and languages for embedded system design. IEE Proc Comput Digit Tech 152(2):114–129

[KSF+99] Kowalewski S, Stursberg O, Fritz M, Graf H, Hoffmann I, Preußig J, Remelhe M, Simon S, Treseler H (1999) A case study in tool-aided analysis of discretely controlled continuous systems: The two tanks problem. In: Hybrid systems V. Springer, Heidelberg, pp 163–185

[LSD08] Liu Y, Sun J, Dong JS (2008) An analyzer for extended compositional process algebras. In: Companion of the 30th international conference on software engineering. ACM, New York, pp 919–920

[MBR06] Meenakshi B, Bhatnagar A, Roy S (2006) Tool for translating simulink models into input language of a model checker. In: Proceedings of the 8th international conference on formal engineering methods. Springer, Heidelberg, pp 606–620

[MCDB03] Muñoz C, Carreño V, Dowek G, Butler RW (2003) Formal verification of conflict detection algorithms. Int J Softw Tools Technol Transf 4(3):371–380

[MD98] Mahony BP, Dong JS (1998) Blending object-Z and timed CSP: an introduction to TCOZ. In: Proceedings of the 20th international conference on software engineering. IEEE Computer Society, Washington, pp 95–104

[MD00] Mahony BP, Dong JS (2000) Timed communicating object Z. IEEE Trans Softw Eng 26(2):150–177

[MH92] Mahony BP, Hayes IJ (1992) A case-study in timed refinement: a mine pump. IEEE Trans Softw Eng 18(9):817–826

[ORS92] Owre S, Rushby JM, Shankar N (1992) PVS: a prototype verification system. In: Proceedings of the 11th international conference on automated deduction. Springer, Heidelberg, pp 748–752

[Pnu02] Pnueli A (2002) Embedded systems: challenges in specification and verification. In: Proceedings of the 2nd international conference on embedded software. Springer, Heidelberg, pp 1–14

[SCBR01] Sims S, Cleaveland R, Butts K, Ranville S (2001) Automated validation of software models. In: Proceedings of the 16th international conference on automated software engineering. IEEE Computer Society, Washington, pp 91–96

[SLDW08] Sun J, Liu Y, Dong JS, Wang HH (2008) Specifying and verifying event-based fairness enhanced systems. In: Proceedings of the 10th international conference on formal engineering methods. Springer, Heidelberg, pp 5–24

[TSCC05] Tripakis S, Sofronis C, Caspi P, Curic A (2005) Translating discrete-time Simulink to Lustre. Trans Embed Comput Syst 4(4):779–818

[Mat08a] The MathWorks. Simulink® 7—reference, March 2008

[Mat08b] The MathWorks. Simulink® 7—using Simulink, March 2008

[TSR03] Tiwari A, Shankar N, Rushby JM (2003) Invisible formal methods for embedded control systems. Proc IEEE 91(1):29–39

[Wan04] Wang F (2004) Formal verification of timed systems: a survey and perspective. Proc IEEE 92(8):1283–1305

[WD96] Woodcock J, Davies J (1996) Using Z: specification, refinement and proof. Prentice-Hall, Englewood Cliffs

[ZHR91] Zhou C, Hoare CAR, Ravn AP (1991) A calculus of durations. Inf Proc Lett 40(5):269–276

[ZL94] Zhou C, Li X (1994) A mean value calculus of durations. In: A classical mind: essays in honour of C. A. R. Hoare. Prentice-Hall International, Englewood Cliffs, pp 431–451

[ZRH93] Zhou C, Ravn AP, Hansen MR (1993) An extended duration calculus for hybrid real-time systems. In: Hybrid systems. Springer, Heidelberg, pp 36–59

Received 9 January 2008
Accepted in revised form 31 December 2008 by U.H.M. Martin and J.C.P. Woodcock
Published online 18 March 2009