Developing a Simulink Library to Verify Signal Temporal Logic Specifications

YU XIA

June 23, 2022

Abstract

1 Introduction

Cyber-Physical systems (abbreviated as CPS) have been massively employed in control systems in domains such as aerospace, automotive, bio-medicine. In such systems, software components sense and act on physical components. Therefore, CPS combines software and physical dynamics. Physical dynamics are typically modeled through formalism that captures the continuous evolution of the environment over time (typically in differential equations); the corresponding behaviors are typically represented as continuous signals. Software dynamics are typically modeled with discrete event systems (e.g., finite state machines, lookup tables), eg see [8].

Model-based development of control systems is today an established industrial practice. The use of models allows a precise and formal definition of the behavior with respect to time and also allows to raise the level of abstraction of the controller logic, thus allowing various forms of verification, such as model checking, monitoring, etc. Along modeling frameworks, Simulink, by Mathworks, is among the most popular modeling environments. The Simulink models used for the representation of cyber-physical systems are based on a synchronous reactive semantics. The model of the controlled physical system is defined by a system of differential equations, integrated in continuous time, while the model of the controller is typically discrete-time. In this context, verification is mainly performed by simulation [3], defining input signals or command and observe the simulated output.

Signal Temporal Logic (STL) provides formal specifications of timed behaviors of systems, and strengthens the expressive of timed properties (e.g. Finally/Eventually, Always/Globally, Until), therefore gaining its popularity among both researchers and end-users to prove a greater set of both safety and liveness properties.

The purpose of this work is to provide a new integration of STL within Simulink in order to support verification activities based on hybrid Simulink model and formal specification of requirements.

Concerns. We refer the reader to [D3] for a more global description on the history of STL, its various versions or similar temporal logics, as well as associated verification methods.

Let us recall two key concerns of ours:

Discrete-time, continuous-time, hybrid systems. While it is possible, and often convenient, to design models in Simulink that combine continuous-time components with discrete-time components, and rely on Simulink simulation engine to properly integrate the continuous part when simulating the full system, the integration with existing tools is not always satisfying.

One of the challenges when simulating such hybrid systems is to properly detect the combination of discrete events with integration steps. A variable-step integration algorithm observes zero-crossings of signals to detect when it should interrupt the current integration computation and record an intermediate point of the state space. The used of discrete components amounts to encode the discrete time events as a clock with zero-crossings occurring at the appropriate rate.

The may lead to bizarre results. For example, when one combines in the same MATLAB system two separate sub-systems, one being discrete-time and the other being continuous time, and without any relationship between the two systems, then the computed values of the continuous part are impacted by the choice the time steps of the discrete part. This is not the current focus of this work but it shows that if one wants to evaluate predicates with respect to a continuous-time system, it is important to model them in the system to allow the simulation engine to find proper integration steps when running the simulations.

As an example, methods such as the ones presented in [5, 7, 9, 6] requires an external tool to produce (bounded) signals, that are then analyzed with respect to a specification.

An interesting related work is [1] which also provide a Simulink library. We have similar constraints on the restriction of the language, at least when considering continuous time systems. We will further develop the comparison in Section 5.1. A key difference is the capability to perform both online and offline analyses.

Online and offline reasoning. Most STL based framework in Simulink intend to perform model testing, providing means to evaluate non trivial STL specifications. The approach proposed in [2] relies on a robust semantics for STL – where the semantics of a formula is expressed through a real-valued evaluation of the formula; the value being strictly positive being equivalent to the validity of the formula. It still focuses on monitoring traces but, thanks to the online evaluation, is able to return the validity of the formula before evaluating the full trajectory. It is presented as more efficient than analyzing it completely.

We share the same objective but for a different purpose. Since our goal is to provide a versatile STL library in Simulink both for monitoring and for verification purposes, it is mandatory to have a Simulink model that produces a result at each time step, and not only at the final time. Methods such as the one in [1] soundly represent STL monitors but cannot return the validity of the formula while evaluating it.

Contributions. We propose here an STL library in Simulink that addresses these concerns:

- STL components are encoded as Simulink subsystems;
- we rely on three-valued logic to model the undefined semantics of a temporal operator which value is not yet determined; therefore it is compatible with both online and offline monitoring while enabling model-checking or static analysis ;
- the approach is, in a large part, independent on the discrete- or continuous- nature of the time model;
- it is implemented in CoCoSim compatible components, providing means to generate code or models from our specification blocks.

2 Signal Temporal Logic

Let us recall the syntax and semantics of STL formulas.

Let us keep the definition of time generic for the moment. We are given with a totally ordered set of time $\mathbb{T}$.

Let $\mathcal{X}$ be a finite sets of signals. Without loss of generality, we can assume that all signals are defined as functions in $\mathbb{T} \to \mathbb{R}$ from time to real values. We will later specialize this definition for discrete-time signals and continuous-time signals.

Let μ be a predicate whose value is determined by the sign of a function of an underlying signal $x \in \mathcal{X}$, i.e., $\mu(t) \equiv \mu(x(t)) > 0$. Let φ, ψ be STL formula, .

Then STL formula φ is defined inductively as:

$$\varphi ::= \mu \mid \neg\varphi \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid \Box_{[a,b]}\psi \mid \Diamond_{[a,b]}\psi \mid \varphi \,\mathcal{U}_{[a,b]}\, \psi$$

Let us now define the semantics of an STL formula. The semantics of a formula φ is defined at a time $t \in \mathbb{T}$ and for a set of signals $\mathcal{X}$ as

$$(\mathcal{X}, t) \vDash \varphi$$

- μ: a predicate is evaluated locally, at time t over the current values of the signals

$$(\mathcal{X}, t) \vDash \mu \Leftrightarrow \mu(t) \tag{1}$$

- $\neg\varphi$ (Negation): the logical negation of φ.

$$(\mathcal{X}, t) \vDash \neg\varphi \Leftrightarrow \neg((\mathcal{X}, t) \vDash \varphi) \tag{2}$$

- $\varphi_1 \wedge \varphi_2$ (And): the logical "and" between φ_1 and φ_2.

$$(\mathcal{X}, t) \vDash \varphi_1 \wedge \varphi_2 \Leftrightarrow (\mathcal{X}, t) \vDash \varphi_1 \wedge (\mathcal{X}, t) \vDash \varphi_2 \tag{3}$$

We recall that, in STL, all temporal operators have to be associated to a bounded, non-singleton time interval $[a, b] \in \mathbb{T} \times \mathbb{T}$.

- $\varphi_1 \, \mathcal{U}_{[a,b]} \, \varphi_2$ (Until): a temporal operator that is satisfied if φ_1 holds until φ_2 becomes True within the time horizon $[a, b]$.

$$(\mathcal{X}, t) \vDash \varphi_1 \, \mathcal{U}_{[a,b]} \, \varphi_2 \Leftrightarrow \exists t' \in t + [a, b] : (\mathcal{X}, t') \vDash \varphi_2 \wedge \forall t'' \in [t, t'] : (\mathcal{X}, t'') \vDash \varphi_1 \tag{4}$$

- $\Diamond_{[a,b]}\varphi$ (Eventually): the condition is verified at least once within the time horizon $[a, b]$.

$$\Diamond_{[a,b]}\varphi \Leftrightarrow \exists t' \in t + [a, b] : (\mathcal{X}, t') \vDash \varphi \tag{5}$$

- $\Box_{[a,b]}\varphi$ (Always or Globally): the condition is always verified within the time horizon $[a, b]$.

$$\Box_{[a,b]}\varphi \Leftrightarrow \forall t' \in t + [a, b] : (\mathcal{X}, t') \vDash \varphi \tag{6}$$

Note that the usual definitions of $\Diamond_{[a,b]}$ and $\Box_{[a,b]}$ based on $\mathcal{U}_{[a,b]}$ still apply:

$$\Diamond_{[a,b]}\varphi = True \, \mathcal{U}_{[a,b]} \, \varphi, \text{ and} \tag{7}$$

$$\Box_{[a,b]}\varphi = \neg(\Diamond_{[a,b]}\neg\varphi). \tag{8}$$

One shall also make a remark on the use of time in STL. While the evaluation of predicates is performed at time t in $(\mathcal{X}, t) \vDash p \Leftrightarrow \mu(t)$, all occurrences of time intervals $[a, b]$ in the definitions of $\mathcal{U}_{[a,b]}$, $\Box_{[a,b]}$ or $\Diamond_{[a,b]}$ are used to delay the current time t: $t + [a, b] = [t + a, t + b]$. These time a and b are then relative times while t acts more as an absolute time.

3 STL online semantics

We now revisit the definitions of STL semantics in order to enable an online use of these. First, we characterize the positive, negative and non-yet-determined values of the various STL connectors. Then we propose Kleene's strong 3-valued logic as the mathematical framework used to reason about STL online semantics. In a third part, we propose a first order logic definition of the three-valued logic using positive and negative logics.

3.1 Formalizing STL indeterminacy

When performing monitoring of STL predicates, for a given value of simulation data, we have multiple characteristics. First the trajectory is typically finite. It is produced by a simulation engine and stored in a data file. It is then loaded by the monitoring tool and analyzed with respect to the STL specification. In this offline setting, the final outcome is whether the input signal satisfies or not the specification. It is a boolean output.

Since STL semantics forces all temporal connectors to be associated with bounded intervals, any STL predicate has a bounded horizon limit, after which it should be guaranteed to be able to determine the validity of a formula.

Let us consider a property $\square_{[0,10]}P$, a set of signals $\mathcal{X}$ and an initial time t_0, e.g., $t_0 = 0$. We are interested in checking $(\mathcal{X}, t_0) \vDash \square_{[0,10]}P$. Let us assume that we are given with a trace for $\mathcal{X}$ of length $l < 10$, e.g., $l = 8$, where the predicate P is valid along the whole trace. What is the validity of such a predicate? On the one hand, it is always valid, but on the other hand, it has not real definition within the time $[l, 10]$.

When performing the evaluation of the STL property in a online fashion, there is knowledge or guarantee that the trajectory will continue. But, in some case, the validity of the predicate can already be determined. Existing works regarding online semantics for STL, such as [2], do not intend to define undetermined status of a formula. But they rather try to optimize the runtime evaluation of the predicate monitoring, detecting when one can conclude, positively or negatively.

3.1.1 Kleene's strong 3-valued logic

In his book [4, §64], Kleene proposed a strong logic of indeterminacy. Each predicate semantics can be valued as *true* (**T**), *false* (**F**) or *unknown* (**U**). Other alternative three-valued logics exists, for example the one of Łukasiewicz. However, in Kleene's proposal, $\mathbf{U} \implies \mathbf{U}$ is not necessary true, while $\mathbf{U} \implies \mathbf{T}$ is valid.

Table 1 presents the truth tables showing the logical operations AND and OR, as well as the negation.

A and B		B		
		F	**U**	**T**
A	**F**	**F**	**F**	**F**
	U	**F**	**U**	**U**
	T	**F**	**U**	**T**

(a) AND Operation

A or B		B		
		F	**U**	**T**
A	**F**	**F**	**U**	**T**
	U	**U**	**U**	**T**
	T	**T**	**T**	**T**

(b) OR Operation

A	$\neg A$
F	**T**
U	**U**
T	**F**

(c) Negation Operation

Table 1: Truth Tables showing Kleene's 3-valued logic operations

This logic is appropriate to represent the unknown status of an STL formula. Simple ones, such as basic predicate will return **T** or **F** values, while more complex ones, involving temporal methods may return an **U** value until a condition has been met or a time window closed.

3.1.2 3-valued logic semantics for STL

The following definition are meant to model the online evaluation of the formulae while translating STL formulae into first order logic. As above, φ and ψ are STL formulae; $[a, b]$ is the time bound, and by assumption, $0 \leq a < b$. $\mathcal{X} = (x_1, x_2, ..., x_n)$ is the inputs starting from time instant t Therefore $(\mathcal{X}, t) \vDash \varphi$ is evaluated at time τ.

$$[\![(\mathcal{X}, t) \vDash \varphi]\!](\tau)$$

Let us now characterize for each construct, the sufficient and necessary conditions to determine a positive (P or T), a negative (N or F) or an indeterminate value (U for *unknown*).

Classical constructs (non temporal)

- Basic predicate μ validity can always be determined. $(\mathcal{X}, t) \vDash p \Leftrightarrow \mu(t)$

  T $\mu(t)$
  F $\neg\mu(t)$
  U $\perp$

- Combinatorial logical operators ($\wedge, \vee, \implies, \neg\varphi, \ldots$) are defined using Table 1 definitions.

Eventually $\Diamond_{[a,b]}\varphi$ Let us recall its definition:

$$\exists t' \in t + [a,b] : (\mathcal{X}, t') \vDash \varphi$$

T We can conclude positively only after the time $t+a$ characterizing the start of the time interval, if a valid condition has been observed.

$$\tau \geq t + a \wedge \exists t' \in [t + a, min(\tau, t + b)] : (\mathcal{X}, t') \vDash \varphi \tag{9}$$

F Invalidity requires to wait until the end of the time interval. Otherwise one cannot conclude.

$$\tau \geq t + b \wedge \forall t' \in [t + a, t + b], (\mathcal{X}, t') \vDash \neg\varphi \tag{10}$$

U The validity is unknown if we have not yet reached the end of the time interval but have not yet observed a suitable time.

$$\tau < t + b \wedge \forall t' \in [t + a, \tau], (\mathcal{X}, t') \vDash \neg\varphi \tag{11}$$

Always $\Box_{[a,b]}\varphi$ Let us recall its definition:

$$\forall t' \in t + [a,b] : (\mathcal{X}, t') \vDash \varphi$$

T Similarly to the negative case of the eventually operator, one needs to wait until the end of the interval to claim validity.

$$\tau \geq t + b \wedge \forall t' \in [t + a, t + b], (\mathcal{X}, t') \vDash \varphi \tag{12}$$

F Invalidity is detected as soon as an invalid time, within the proper time interval, is observed.

$$\tau \geq t + a \wedge \exists t' \in [t + a, min(\tau, t + b)] : (\mathcal{X}, t') \vDash \neg\varphi \tag{13}$$

U Unknown cases are either before the time interval or within it, if the property φ is valid, up to now.

$$\tau < t + b \wedge \forall t' \in [t + a, \tau], (\mathcal{X}, t') \vDash \varphi \tag{14}$$

Until $(\mathcal{X}, t) \vDash \varphi_1 \, \mathcal{U}_{[a,b]} \, \varphi_2$

The until operator is the most complex. As for other constructs, let us first recall its classical definition, before characterizing the true, false and unknown definitions.

$$\exists t' \in t + [a,b] : (\mathcal{X}, t') \vDash \varphi_2 \wedge \forall t'' \in [t, t'] : (\mathcal{X}, t'') \vDash \varphi_1$$

T We conclude positively when a event φ_2 occurred within the proper time interval, while satisfying the other conditions.

$$\exists t' \in [t + a, min(\tau, t + b)], \big((\mathcal{X}, t) \vDash \Box_{[0,t'-t]}\varphi_1\big) \wedge ((\mathcal{X}, t') \vDash \varphi_2) \tag{15}$$

F There are multiple conditions for the formula to be false. First whenever $(\mathcal{X}, \tau) \vDash \neg\varphi_1$ and $\tau < t + a$. Or, when within the time interval $[t + a, min(\tau, t + b)]$ there was a violation of the property, a first time t' where $((X), t') \vDash \varphi_1 \wedge \neg\varphi_2$ while φ_1 hold before, ie. $(\mathcal{X}, t) \vDash \Box_{[a,t'-t]}\varphi_1$.

$$\bigvee \begin{array}{l} (\exists t' \in [t, min(\tau, t + a)], (\mathcal{X}, t') \vDash \neg\varphi_1) \\ \big(\tau \geq t + b \wedge (\mathcal{X}, t) \vDash \Box_{[a,b]}(\varphi_1 \wedge \neg\varphi_2)\big) \\ (t + a \leq \tau \leq t + b \wedge \exists t' \in [t + a, \min(\tau, t + b)] : \big((\mathcal{X}, t') \vDash \neg\varphi_1 \wedge (\mathcal{X}, t) \vDash \Box_{[a,t'-t]}\neg\varphi_2\big)\big) \end{array} \tag{16}$$

U We cannot yet conclude on the validity of the formula, if, for the moment the formula is neither validated nor violated. A first condition is that φ_1 hold since time t until now. A second is that, at the current time (*currenttime*), we have $\neg\varphi_2$. Third, these condition only apply before reaching the end of the time interval, that is time $t + b$, where we are able to conclude positively or negatively.

$$\tau < t + b \wedge (\mathcal{X}, t) \vDash \Box_{[0,\tau-t]}\varphi_1 \wedge (\mathcal{X}, t) \vDash \Box_{[a,\tau-t]}\neg\varphi_2 \tag{17}$$

3.2 Positive and Negative logics

Designing and implementing these components with three-valued logics can be eased by considering two different logics: $\mathcal{L}_T$ denotes a *positive logic* that can model the *true* value on one hand, and *false/unknown* on the other hand. In that logic false and unknown cannot be distinguished. Similarly $\mathcal{L}_F$, the *negative logic*, can represent *false* and *true/unknown*, without being able to distinguish the two values in the latter.

3.2.1 Negative logic $\mathcal{L}_F$

Let $\mathcal{L}_F$ be the logics focusing on the characterization of False values. There are two truth values: U_T and F. U_T indicates True or Unknown and is associated to the boolean value *true*; while F indicates False and is associated with the boolean value *false*. The definitions with respect to operators (Always, Eventually, Until) are as followed.

1. **Eventually**, $(\mathcal{X},t) \vDash_{\mathcal{L}_F} \Diamond_{[a,b]}\varphi$

   U_T

   $$\tau < t+b \vee \exists t' \in [t+a, \min(\tau, t+b)] : (\mathcal{X},t') \vDash_{\mathcal{L}_F} \varphi \tag{18}$$

   F

   $$\tau \geq t+b \wedge \forall t' \in [t+a, \min(\tau, t+b)] : (\mathcal{X},t') \vDash_{\mathcal{L}_F} \neg\varphi \tag{19}$$

2. **Always**, $(\mathcal{X},t) \vDash_{\mathcal{L}_F} \Box_{[a,b]}\varphi$

   U_T

   $$\forall t' \in [t+a, \min(\tau, t+b)] : (\mathcal{X},t') \vDash_{\mathcal{L}_F} \varphi \tag{20}$$

   F

   $$\exists t' \in [t+a, \min(\tau, t+b)] : (\mathcal{X},t') \vDash_{\mathcal{L}_F} \neg\varphi \tag{21}$$

3. **Until** $(\mathcal{X},t) \vDash_{\mathcal{L}_F} \varphi_1 \mathcal{U}_{[a,b]} \varphi_2$:

   U_T

   $$\begin{aligned} & \bigvee \begin{cases} \left(\tau < t+b \wedge (\mathcal{X},t) \vDash_{\mathcal{L}_F} \Box_{[0,\tau-t]}\varphi_1 \wedge (\mathcal{X},t) \vDash_{\mathcal{L}_F} \Box_{[a,\tau-t]}\neg\varphi_2\right) \\ (\exists t' \in [t+a, min(\tau, t+b)], \left((\mathcal{X},t) \vDash_{\mathcal{L}_F} \Box_{[0,t'-t]}\varphi_1\right) \wedge (\mathcal{X},t') \vDash_{\mathcal{L}_F} \varphi_2) \end{cases} \\ \equiv \quad & \bigvee \begin{cases} (\tau < t+b \wedge \forall t' \in [t,\tau] : (\mathcal{X},t') \vDash_{\mathcal{L}_F} \varphi_1 \wedge \forall t' \in [t+a,\tau] : (\mathcal{X},t') \vDash_{\mathcal{L}_F} \neg\varphi_2) \\ (\exists t' \in [t+a, min(\tau, t+b)], (\forall t'' \in [t,t'] : (\mathcal{X},t'') \vDash_{\mathcal{L}_F} \varphi_1) \wedge (\mathcal{X},t') \vDash_{\mathcal{L}_F} \varphi_2) \end{cases} \end{aligned} \tag{22}$$

   F (we use the negation of the definition of U_T)

   $$\begin{aligned} & \neg \bigvee \begin{cases} (\tau < t+b \wedge \forall t' \in [t,\tau] : (\mathcal{X},t') \vDash_{\mathcal{L}_F} \varphi_1 \wedge \forall t' \in [t+a,\tau] : (\mathcal{X},t') \vDash_{\mathcal{L}_F} \neg\varphi_2) \\ (\exists t' \in [t+a, min(\tau, t+b)], (\forall t'' \in [t,t'] : (\mathcal{X},t'') \vDash_{\mathcal{L}_F} \varphi_1) \wedge (\mathcal{X},t') \vDash_{\mathcal{L}_F} \varphi_2) \end{cases} \\ \equiv \quad & \bigwedge \begin{cases} (\tau \geq t+b \vee \exists t' \in [t,\tau] : (\mathcal{X},t') \vDash_{\mathcal{L}_F} \neg\varphi_1 \vee \exists t' \in [t+a,\tau] : (\mathcal{X},t') \vDash_{\mathcal{L}_F} \varphi_2) \\ (\forall t' \in [t+a, min(\tau, t+b)], (\exists t'' \in [t+a,t'] : (\mathcal{X},t'') \vDash_{\mathcal{L}_F} \neg\varphi_1) \vee (\mathcal{X},t') \vDash_{\mathcal{L}_F} \neg\varphi_2) \end{cases} \end{aligned} \tag{23}$$

Proof. The U_T definition is directly built as a disjunction between the unknow and True cases. However the uses of the operator $\Box$ have been replaced by their definition in $\mathcal{L}_F$. Let us first check that these parts are sound. When $\tau \in t + [a,b[$ and the property $\varphi_1 \wedge \neg\varphi_2$ has been valid up to now, we are not able to conclude positively or negatively, hence the U_T value. Last, when the until property is valid, there exists a time $t' \in [t+a, \min(\tau, t+b)]$ such that φ_1 is valid from $t+a$ to t' while $(\mathcal{X},t') \vDash_{\mathcal{L}_F} \varphi_2$.

The definition of F is more difficult. Similarly, one can enumerate all cases and identify the matching parts in the definition. Let us give an example. One of the cases is when $(\mathcal{X},t+a) \vDash_{\mathcal{L}_F} \neg\varphi_1$. In that case, for any time $currenttime \geq a$, one can select the following atoms: (1) $\exists t' \in [t,\tau] : (\mathcal{X},t') \vDash_{\mathcal{L}_F} (\neg\varphi_1 \vee \varphi_2)$ with $t' = t+a$, hence $(\mathcal{X},t+a) \vDash_{\mathcal{L}_F} \neg\varphi_1$; (2) $\forall t' \in [a, min(t,b)], (\exists t'' \in [a,t'] : (\mathcal{X},t'') \vDash_{\mathcal{L}_F} \neg\varphi_1)$ with choosing the value $t'' = a$ regardless of the value of t, hence $(\mathcal{X},a) \vDash_{\mathcal{L}_F} \neg\varphi_1$.

□

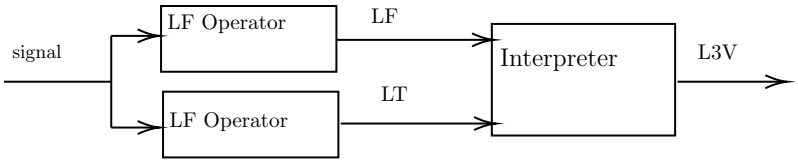


Figure 1: Interpreter from positive and negative logics into the 3-valued logic.

OPERATORS \ $[\mathcal{L}_F, \mathcal{L}_T]$	$[false, false]$	$[false, true]$	$[true, false]$	$[true, true]$
ALWAYS	False	impossible	Unknown	True
EVENTUALLY	False	impossible	Unknown	True
UNTIL	False	impossible	Unknown	True

Table 2: Conversion Table of Interpreter

3.2.2 Positive logic $\mathcal{L}_T$

Let $\mathcal{L}_T$ be the logics focusing on the characterization of True values. There are two truth values: T and U_F. T indicates True, corresponding to boolean value *true*; while U_F indicates False or Unknown, corresponding to boolean value *false*. The definitions with respect to operators (Always, Eventually, Until) are as followed:

1. **Eventually**, $(\mathcal{X}, t) \vDash_{\mathcal{L}_T} \Diamond_{[a,b]} \varphi$

   T

   $$\exists t' \in [t+a, \min(\tau, t+b)] : (\mathcal{X}, t') \vDash_{\mathcal{L}_T} \varphi \tag{24}$$

   U_F

   $$\forall t' \in [t+a, \min(\tau, t+b)] : (\mathcal{X}, t') \vDash_{\mathcal{L}_T} \neg\varphi \tag{25}$$

2. **Always**, $(\mathcal{X}, t) \vDash_{\mathcal{L}_T} \Box_{[a,b]} \varphi$

   T

   $$\tau \geq t+b \wedge \forall t' \in [t+a, \min(\tau, t+b)] : (\mathcal{X}, t') \vDash_{\mathcal{L}_T} \varphi \tag{26}$$

   U_F

   $$\tau < t+b \vee \exists t' \in [t+a, \min(\tau, t+b)] : (\mathcal{X}, t') \vDash_{\mathcal{L}_T} \neg\varphi \tag{27}$$

3. **Until**, $(\mathcal{X}, t) \vDash_{\mathcal{L}_T} \varphi_1 \mathcal{U}_{[a,b]} \varphi_2$

   T

   $$\begin{aligned} & \exists t' \in [t+a, min(\tau, t+b)], \big((\mathcal{X}, t) \vDash_{\mathcal{L}_T} \Box_{[0,t'-t]} \varphi_1\big) \wedge (\mathcal{X}, t') \vDash_{\mathcal{L}_T} \varphi_2 \\ \equiv \quad & \exists t' \in [t+a, min(\tau, t+b)], \forall t'' \in [t, t'] (\mathcal{X}, t'') \vDash_{\mathcal{L}_T} \varphi_1 \wedge (\mathcal{X}, t') \vDash_{\mathcal{L}_T} \varphi_2 \end{aligned} \tag{28}$$

   U_F

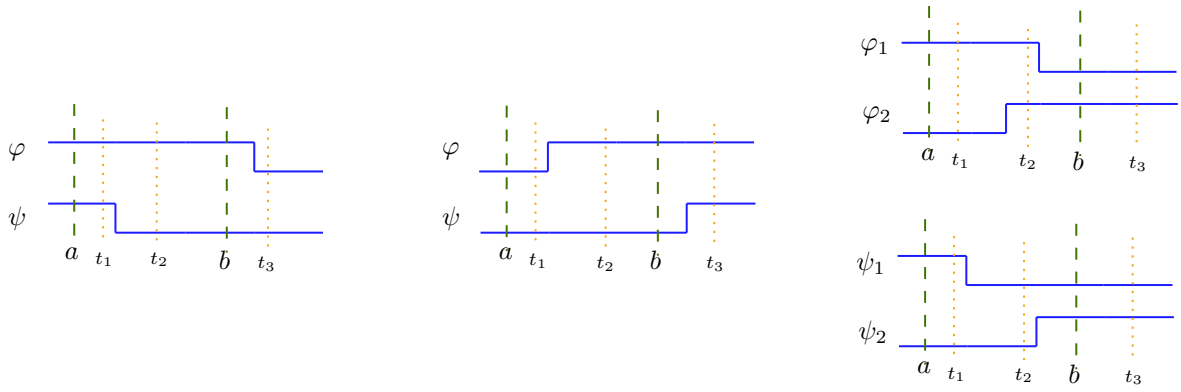
   $$\forall t' \in [t+a, min(\tau, t+b)], \exists t'' \in [t, t'] (\mathcal{X}, t'') \vDash \neg\varphi_1 \vee (\mathcal{X}, t') \vDash \neg\varphi_2 \tag{29}$$

3.2.3 From pairs of 2-valued logic values into 3-valued logic

We can now combine our two underlying logics to compute our three-valued logic. The interpretation process is shown in Figure 1: the signal under concern is evaluated with operators of both logics simultaneously, and the outputs going through the truth table presented in Table 2. Note that *impossible* is not a truth value. It indicates that such a combination will never appear.

For example, let us consider the **Always** operator, when $[\mathcal{L}_F, \mathcal{L}_T] = [false, false]$. Using the definitions above, we have

$$\bigwedge \begin{cases} \exists t' \in [t+a, \min(\tau, t+b)] : (\mathcal{X}, t') \vDash_{\mathcal{L}_F} \neg\varphi \\ \tau < t+b \vee \exists t' \in [t+a, \min(\tau, t+b)] : (\mathcal{X}, t') \vDash_{\mathcal{L}_T} \neg\varphi \end{cases} \tag{30}$$



Subfig(1): *Always* Subfig(2): *Eventually* Subfig(3): *Until*

Figure 2: Illustrations

Predicates		t_1	t_2	t_3
$\mathcal{L}_F$	$\Box_{[a,b]}\varphi$	U_T	U_T	U_T
	$\Box_{[a,b]}\psi$	U_T	F	F
$\mathcal{L}_T$	$\Box_{[a,b]}\varphi$	U_F	U_F	T
	$\Box_{[a,b]}\psi$	U_F	U_F	U_F
$\mathcal{L}_{3V}$	$\Box_{[a,b]}\varphi$	U	U	T
	$\Box_{[a,b]}\psi$	U	F	F

(a) Always

Predicates		t_1	t_2	t_3
$\mathcal{L}_F$	$\Diamond_{[a,b]}\varphi$	U_T	U_T	U_T
	$\Diamond_{[a,b]}\psi$	U_T	U_T	F
$\mathcal{L}_T$	$\Diamond_{[a,b]}\varphi$	U_F	T	T
	$\Diamond_{[a,b]}\psi$	U_F	U_F	U_F
$\mathcal{L}_{3V}$	$\Diamond_{[a,b]}\varphi$	U	T	T
	$\Diamond_{[a,b]}\psi$	U	U	F

(b) Eventually

Predicates		t_1	t_2	t_3
$\mathcal{L}_F$	$\varphi_1 U_{[a,b]}\varphi_2$	U_T	U_T	U_T
	$\psi_1 U_{[a,b]}\psi_2$	U_T	F	F
$\mathcal{L}_T$	$\varphi_1 U_{[a,b]}\varphi_2$	U_F	U_F	T
	$\psi_1 U_{[a,b]}\psi_2$	U_F	U_F	U_F
$\mathcal{L}_{3V}$	$\varphi_1 U_{[a,b]}\varphi_2$	U	T	T
	$\psi_1 U_{[a,b]}\psi_2$	U	F	F

(c) Until

Table 3: Truth Tables showing the Simulation Results.

which can be reduce to

$$\exists t' \in [t+a, \min(\tau, t+b)] : (\mathcal{X}, t') \vDash_{\mathcal{L}_F} \neg\varphi$$

Therefore there is undoubtedly a detected violation of the STL predicate, and the outcome truth value ought to be False. The other cases follow the same rules.

3.3 Examples

To help better understand these three logics and their differences, Figure 2 presents a test case for each temporal operator and Table 3 presents the obtained truth values observed at time t_1, t_2 and t_3, on the diagrams.

$$\neg(P_{LT}) \equiv (\neg P)_{LF}$$

$$\neg(\Diamond_{LT} P) \equiv \Box_{LF}(\neg P)$$

3.4 Summary of logical modeling

Based on [1]

Let us formulate some remarks on the proposed logical framework.

First, as said above, it is worth noting that the truth value "Unknown" is uniquely meaningful in an online fashion, in each type of logic. Because as stated in the definitions in 2, the trajectory of $\tau > b$ does not affect the result, in other words, the result is decided and observable as either True of False. This can

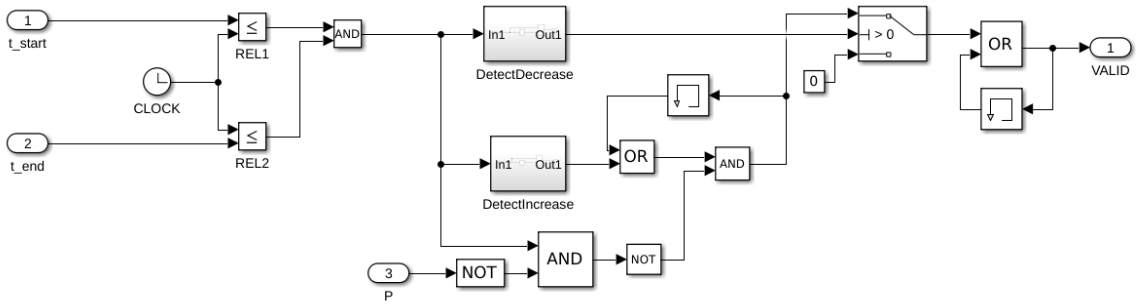


Figure 3: $\mathcal{L}_T$ **Always** component, implementing Equations.(26) – (27)

be made more complex when using nested operators. But in all cases, there is a finite bound in the required time to be able to produce a *true* vs *false* result.

The logic definition have been designed with genericity in mind. As an example, we tried to avoid as much as possible strict inequalities over time. In order to have a logical framework that is compatible with both discrete-time and continuous-time systems, it is preferable to avoid strict inequalities over time. When we will choose to encode a component in Simulink, we will try, as much as possible, to select the formula relying on loose inequalities. As an example in the definition of *Eventually* for the logic $\mathcal{L}_F$, the choice of Equation 19 is more appropriate than Equation 18. One can then define the component negatively and negate the output.

Note also that universal and existential quantifiers when computed on empty intervals, return, respectively, *true* and *false* values.

The proposed logics can also support other uses of the specification. As an example. $\mathcal{L}_F$ can be used together with the Stop Simulation block of Simulink to stop the simulation when the input is nonzero. The simulation completes the current time step before terminating. If the block input is a vector, any nonzero vector element causes the simulation to stop [1]. $\mathcal{L}_L$ is useful to build detectors - undesirable behavior turn to $0 - >$ Simulink Stop.

Last but not least, these logics can be applied to any fashion, online/offline; any system, continuous/discrete; any means of implementation, Simulink blocks / LustreC.

4 Simulink modeling of STL online logic(s)

For each of the construct presented above, we associate a Simulink block. Our encoding is similar with the one presented in [1], but, thanks to our two logics, we are able to produce *unknown* values before the validity of the STL predicate can be determined.

Figures 3, 4 and 5 are respectively associated to the $\mathcal{L}_T$ logic equations (26) – (27), (24) – (25) and (28) – (29), for the Always, Eventually and Until operators and describe the Simulink monitors encoding these constructs.

Similarly, for logic $\mathcal{L}_F$, Figures 6, 7 and 8 are respectively associated to the $\mathcal{L}_F$ logic equations (20) – (21), (18) – (19) and (22) – (23), for the Always, Eventually and Until operators and describe the Simulink monitors encoding these constructs.

Definition of the operator for the 3-valued logic are build using individual component and the 3-valued logic interpreter. This is depicted in Figures 9, 10 and 11.

[1] https://www.mathworks.com/help/simulink/slref/stopsimulation.html

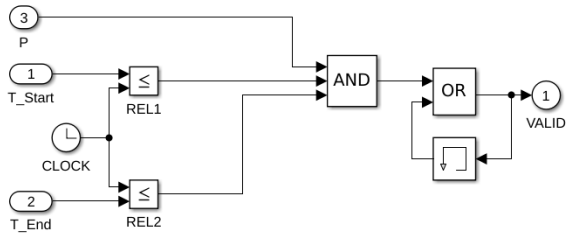


Figure 4: $\mathcal{L}_T$ **Eventually** component, implementing Equations.(24) – (25)

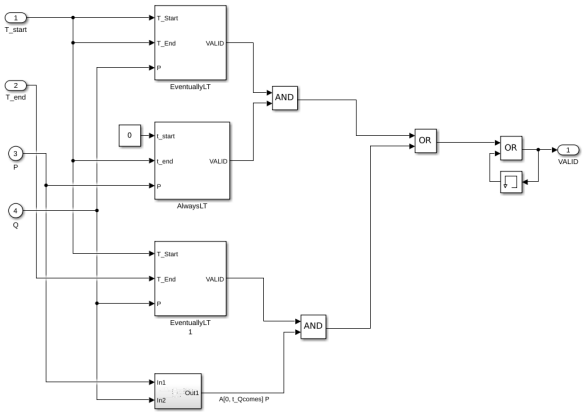


Figure 5: $\mathcal{L}_T$ **Until** component, implementing Equations.(28) – (29)

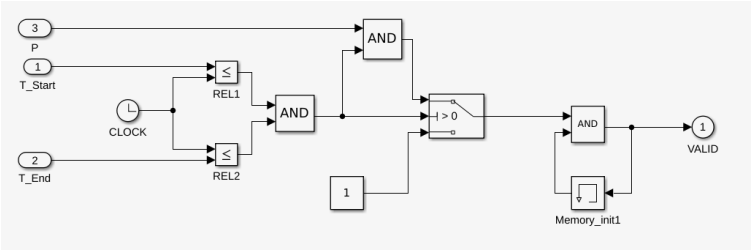


Figure 6: $\mathcal{L}_F$ **Always** component, implementing Equations.(20) – (21)
P - atomic predicates($\mathcal{X}$)
T_Start - a
T_End - b Both Construct and Equations are equivalent to:

```
if(clk < t_start)
    out = true;
elseif (clk >= t_start && clk <= t_end && pre_out)
    out = true;
else
    out = false;
end
elseif (clk >= t_start && clk <= t_end && not(pre_out))
    out = false;
else
    out = pre_out;
end
```

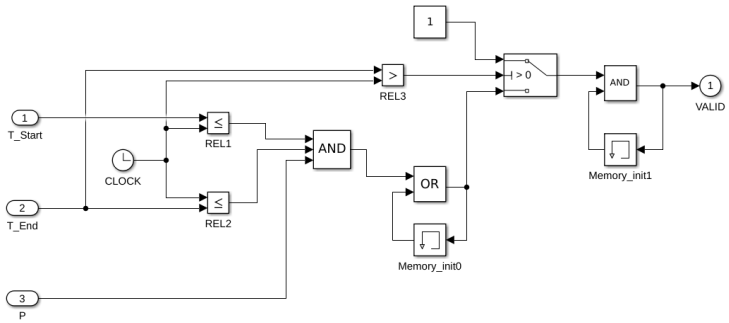


Figure 7: $\mathcal{L}_F$ **Eventually** component, implementing Equations.(19) – (18)

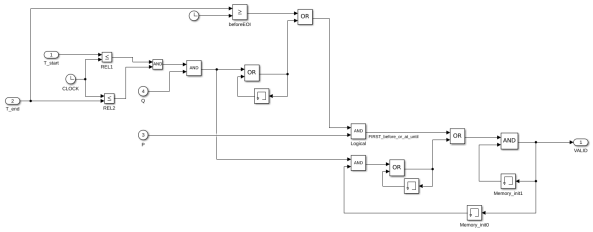


Figure 8: $\mathcal{L}_F$ **Until** component, implementing Equations.(23) – (22)

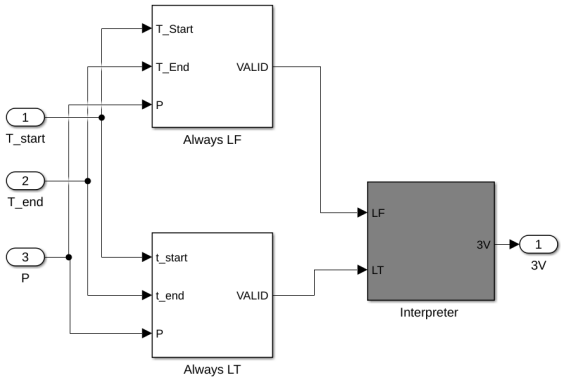


Figure 9: 3V Always

5 Implementation as a Simulink CocoSim library

5.1 Remarks on language restrictions.

In [1], the authors consider the following restrictions of their input language:

> In order to generate online monitors, we introduce the following restrictions to the STL language.
>
> - The maximum level of nesting for temporal operators is two.
> - If there is a nested temporal operator, the condition on which the outer operator is evaluated must be a conjunction and at least one of the terms of the conjunction must be a proposition (not a temporal operator).
> - If Tb is the maximum value for all the endpoints of the intervals defined in the inner (nested)

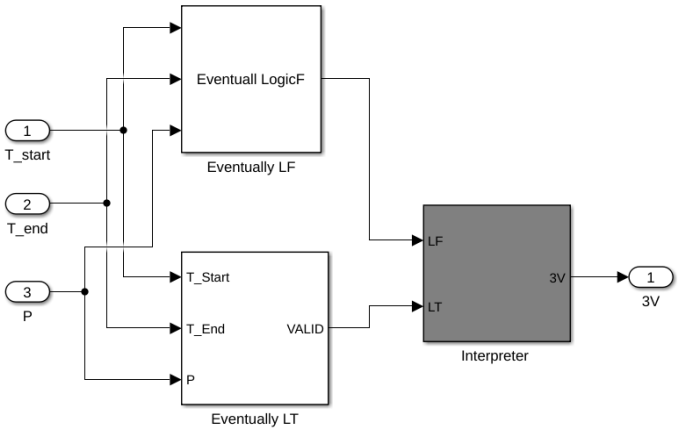


Figure 10: 3V eventually

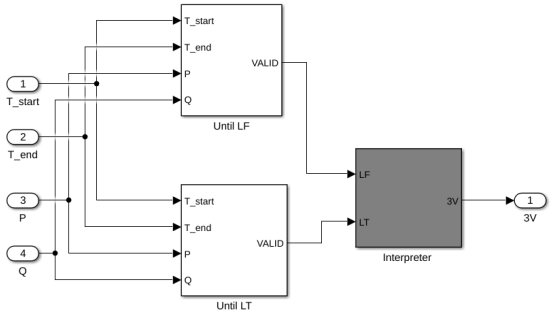


Figure 11: 3V until

temporal operators, then the terms of the conjunction that are not temporal operators can only be true at time instants that are separated by a time interval always greater than Tb.

5.1.1 First setting: no nested temporal operators

For the moment our proposal follows the same principle. In the definitions of $\mathcal{L}_T$ and $\mathcal{L}_F$, we made no assumptions on the sub-terms appearing in temporal operators. However, the previous figures consider regular boolean inputs rather than three valued logics.

A first use is then rely directly on these blocks, that provide online/offline capabilities while being identical for discrete-time and continuous-time systems. The clocks and the memories used in these will be appropriately converted to the proper format by the Simulink simulation engine, or through CoCoSim.

5.1.2 Second setting: no language restriction for discrete-time systems

The main issue with nested temporal operators is the computation of universal quantification. When one needs to check a property at each time step, if the property is a simple signal, one can accumulate it in a memory as proposed above.

However if this underlying formula relies also on one or more temporal operators, then one need to replicate the subsystems evaluate the sub-term for each occurrence of the time window. Thanks to the grammar-imposed bounded time interval, one could duplicate the observer of the underlying formula, a given number of time, depending on the time frame.

For example, φ is a discrete-timed Boolean signal, such as:

$$t = 01234567\ldots$$

$$\varphi = 00011000\ldots$$

Evaluating the STL formula can be seen as Starting from t=a, . This can be realized using our STL operators, as illustrated in Figure 12.

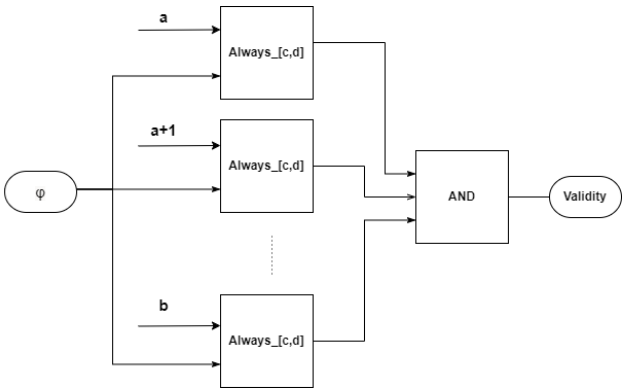


Figure 12: Illusion of nested STL operators

Note that this proposed method is not yet implemented. One can use the existing component and duplicate them, but the unrolling is not yet provided by our STL library.

List the commonly-used cases and provide solution. The sample time $Ts \in \mathbb{R}$,

1. Eventually-Eventually

$$\Diamond_{[a,b]}(\Diamond_{[c,d]}\varphi) \Leftrightarrow \Diamond_{[a+c,b+d]}\varphi \tag{31}$$

given that $Ts < d - c$, in all the logics

Proof. to be done □

OR

$$\Diamond_{[a,b]}(\Diamond_{[c,d]}\varphi) \Leftrightarrow \Diamond_{[a+c,a+d]}\varphi \vee \Diamond_{[a+c+Ts,a+d+Ts]}\varphi \ldots \vee \Diamond_{[b+c,b+d]}\varphi \tag{32}$$

2. Always-Always

$$\Box_{[a,b]}(\Box_{[c,d]}\varphi) \Leftrightarrow \Box_{[a+c,b+d]}\varphi \tag{33}$$

given that $Ts < d - c$

OR

$$\Box_{[a,b]}(\Box_{[c,d]}\varphi) \Leftrightarrow \Box_{[a+c,a+d]}\varphi \wedge \Box_{[a+c+Ts,a+d+Ts]}\varphi \ldots \wedge \Box_{[b+c,b+d]}\varphi \tag{34}$$

3. Eventually-Always

$$\Diamond_{[a,b]}(\Box_{[c,d]}\varphi) \Leftrightarrow \Box_{[a+c,a+d]}\varphi \vee \Box_{[a+c+Ts,a+d+Ts]}\varphi \ldots \vee \Box_{[b+c,b+d]}\varphi \tag{35}$$

4. Always-Eventually

$$\Box_{[a,b]}(\Diamond_{[c,d]}\varphi) \Leftrightarrow \Diamond_{[a+c,a+d]}\varphi \wedge \Diamond_{[a+c+Ts,a+d+Ts]}\varphi \ldots \wedge \Diamond_{[b+c,b+d]}\varphi \tag{36}$$

5. Until

6. specific cases used in Detectors

$$\Diamond_{[0,T]}(\varphi \wedge \Box/\Diamond_{[a,b]}\psi) \tag{37}$$

7. how

$$\Box_{[0,T]}(\varphi \wedge \Box/\Diamond_{[a,b]}\psi) \tag{38}$$

5.2 A library of components

A Simulink library implementing presented blocks is available [2].

5.2.1 Encoding of Three-valued logics

Encoding of Three-valued logics may designed to users' needs. One possibility to encode these values as integers, as shown in Figure 13:

$$True \coloneqq 1$$
$$False \coloneqq 0$$
$$Unknown \coloneqq -1$$

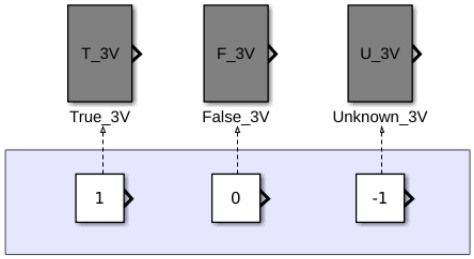


Figure 13: Three Values Encoding

In order to save memory, values can also be encoded in a vector of two boolean digits. For example, as shown in Figure 14:

$$True \coloneqq [1,1]$$
$$False \coloneqq [0,0]$$
$$Unknown \coloneqq [1,0]$$

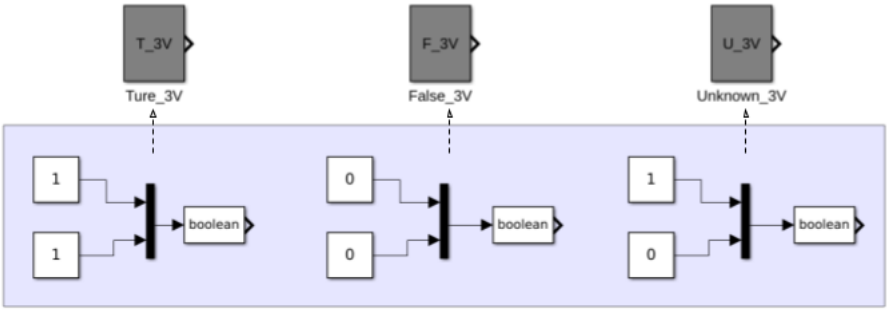


Figure 14: Three Values Encoding as vector

Whatever the encoding is, the logic operations involving this three-valued logics remain the same, because the digital values are wrapped in the three values T_3V, F_3V, U_3V. The data type of the outputs are different though: if encoded with "1, 0, -1", the outputs are in numeric data type such as DOUBLE; if encoded with "[1, 1], [0, 0], [1, 0]", the outputs are in data type ARRAY or VECTOR.

An construct of interpreter as explained in Figure 1 and in Table 2 is shown in Figure 15.

[2] https://github.com/xiayu3333/CoCoSTLib

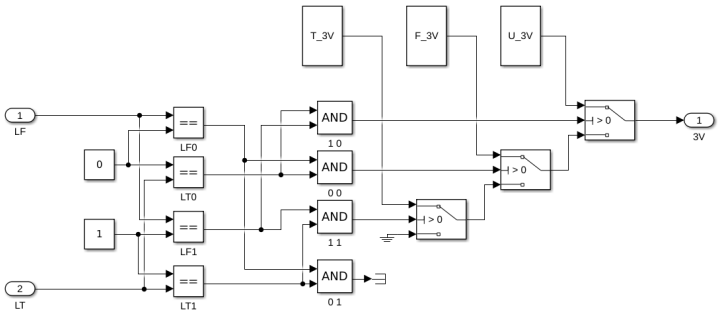


Figure 15: $\mathcal{L}_F, \mathcal{L}_T$ to $\mathcal{L}_{3V}$ Interpreter

Q	Pre Q	Result
0	0	0
0	1	0
1	0	0
1	1	1

(a) Truth Table of Latch $\mathcal{L}_F$

Q	Pre Q	Result
0	0	0
0	1	1
1	0	1
1	1	1

(b) Truth Table of Latch $\mathcal{L}_T$

Table 4: Truth Tables of Latches

5.2.2 Others

As stateful , Latches $\mathcal{L}_F$ Latch, $\mathcal{L}_T$ Latch, $\mathcal{L}_{3V}$ Latch is guaranteed by the former latches.

Based on the truth tables,

6 Use Cases

In this section, we apply CoCoSTLib in two use cases: classification of signal

6.1 Detecting

propose Signal Template Library (ST-Lib), a uniform modeling language to encapsulate a number of useful signal patterns in a formal requirement language with the goal of facilitating requirement formulation for automotive control applications. ST-Lib consists of basic modules known as signal templates. Informally, these specify a characteristic signal shape and provide numerical parameters to tune the shape. We propose two use-cases for ST-Lib: (1) allowing designers to classify design behaviors based on user-defined numerical parameters for signal templates, and (2) automatic identification of worst-case values for the signal template parameters for a given model. [9]

We demonstrate how CoCoSTLib can be used to express the specifications of ST-Lib. STL formula 39 is a template to define unacceptable spike behaviors

$$\Diamond_{[0,T]}(s' > m \wedge \Diamond_{[0,\omega]} s' < -m) \tag{39}$$

s' is denoted the discrete-time derivative or one continuous-time integration step difference of s. m, ω are positive real numbers, with ω greater than the minimum of step size. The parameter ω is related to the spike width, and $m \cdot \omega$ is proportional to the spike amplitude.

This specification captures the occurrence of spike events, True Logic.

Ref.ID	Description	Formula
ReqIP01	Settling Time - Pendulum angle θ (from the vertical, same applied below) becomes less than 0.02 rad in 3 seconds	$\Diamond_{[0.05,3]}\lvert\theta\rvert \leq 0.02$
ReqIP02	Steady State - Since t = 3s, θ is always smaller than 0.02 rad.	$\Box_{[3,10]}\lvert\theta\rvert \leq 0.02$
ReqIP03	Safety - θ is never more than 90 deg (1.571 rad)	$\Box_{[0,10]}\lvert\theta\rvert \leq 1.571$

Table 5: Requirements of Inverted Pendulum system.

6.2 Monitoring work with CoCoSim

In the second use case, we will apply CoCoSim and CoCoSTLib to test two Inverted Pendulum system models. The illustration of balancing an Inverted Pendulum on a cart is shown in Figure 8: Inverted Pendulum on a Cart. A pendulum is attached to a stick on a cart. The controller moves the cart to balance this pendulum.

The requirements are given in 5. The variable used for testing is the angle between the stick that pendulum attached to and the line of gravity ("θ" in figure and "Theta" in models). In ReqIP01, time bound starts at t = 0.5 because the initial value of θ could be close to θ, so setting the time bound t = 0.5 can avoid this interference. Meanwhile, in ReqIP02 and ReqIP03, time bounds end at t = 10 because the simulations stop at t = 10.

We question about two Inverted Pendulum models, one using PI-controller[3] (hereinafter referred to as "IP-PI") and one using neural network controller[4] (as "IP-NN"). IP-PI is given in form of M-Files, so we run the model and use the produced trajectory to perform offline testing; IP-NN is Simulink models, therefore we perform online testing by associating the model directly with contract.

Firstly, build the CoCoSim Contract to implement the test cases using CoCoSTLib with respect to the requirements. See Figure 16, we have three Guarantees, and each refers to one requirement. In this case, we use False Logic operators (Eventually_LF and Always_LV) in CoCoSim Contract, because at this moment, CoCoSim does not support Three-valued Logic, and False Logic is more suitable when verifying systems against requirements.

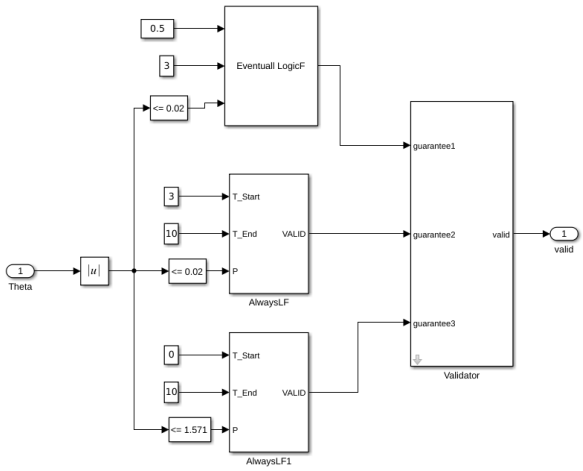


Figure 16: Under the mask of CoCoSim Contract used to verify Inverted Pendulum systems

[3]Source - https://www.mathworks.com/matlabcentral/fileexchange/33942-pi-controller-for-inverted-pendulum

[4]Source - Yin H, Seiler P, Arcak M. Stability analysis using quadratic constraints for systems with neural network controllers[J]. IEEE Transactions on Automatic Control, 2021.

Secondly, associate the contract with models. For IP-PI, we run the model scripts, obtain the simulation data, and then import the data in a Simulink model to perform tests; for IP-NN, we associate the model with the contract (Figure 17).

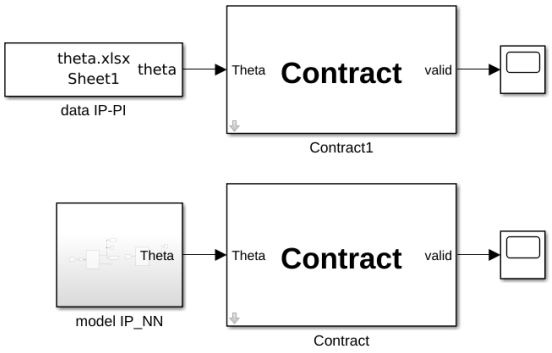


Figure 17: Contract associated with model and trajectory.

At last, run the testing models, and find results. IP-PI passed the test, meaning that it met all three requirements, (see Figure 14: Test result of IP-PI, the result is 1 or True). While IP-NN didn't, (see Figure 15: Test result of IP-NN, the result became 0 or False at 0.05 seconds). We invested in each Guarantee and saw the results in Figure 16: Decomposition of Test result of IP-NN. It shows that each Guarantee gives a False, meaning that this model does not meet any requirement.

6.3 Three-valued logic use case

7 Conclusion

In future work: 1. enrich the logics and library to include nested operations 2. enrich the library from boolean satisfaction to robust satisfaction 3. automatic generation 4. integration to tools, i.e., FRET, CoCoSim.

References

[1] A. Balsini et al. "Generation of simulink monitors for control applications from formal requirements". In: *2017 12th IEEE International Symposium on Industrial Embedded Systems (SIES)*. 2017, pp. 1–9. DOI: 10.1109/SIES.2017.7993389.

[2] Jyotirmoy V. Deshmukh et al. *Robust Online Monitoring of Signal Temporal Logic*. 2015. arXiv: 1506.08234 [cs.SY].

[3] James Kapinski et al. "Simulation-based approaches for verification of embedded control systems: An overview of traditional and advanced modeling, testing, and verification techniques". In: *IEEE Control Systems Magazine* 36.6 (2016), pp. 45–64.

[4] Stephen Cole Kleene. *Introduction to Metamathematics*. Amsterdam: North-Holland, 1952.

[5] Oded Maler and Dejan Nickovic. "Monitoring Temporal Properties of Continuous Signals". In: *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*. Ed. by Yassine Lakhnech and Sergio Yovine. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 152–166. ISBN: 978-3-540-30206-3.

[6] Shiva Nejati et al. "Evaluating Model Testing and Model Checking for Finding Requirements Violations in Simulink Models". In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2019. Tallinn, Estonia: Association for Computing Machinery, 2019, pp. 1015–1025. ISBN: 9781450355728. DOI: 10.1145/3338906.3340444. URL: https://doi.org/10.1145/3338906.3340444.

[7] Dejan Nickovic and Tomoya Yamaguchi. *RTAMT: Online Robustness Monitors from STL*. 2020. arXiv: 2005.11827 [cs.LO].

[8] André Platzer. *Logical foundations of cyber-physical systems*. Vol. 662. Springer, 2018.

[9] Tomoya Yamaguchi et al. "ST-Lib: A Library for Specifying and Classifying Model Behaviors". In: *SAE 2016 World Congress and Exhibition*. SAE International, Apr. 2016. DOI: https://doi.org/10.4271/2016-01-0621.