SAPIENZA
UNIVERSITÀ DI ROMA

Generazione automatica di monitor in MATLAB Simulink per formule espresse in Temporal Logic

Facoltà di Ingegneria dell'informazione, Informatica e Statistica
Corso di laurea in Informatica

Candidato
Giorgio Mariani
n° matricola 1618563

Responsabile
Prof. Federico Mari

A/A 2016/2017

Abstract

Model Checking is a system verification technique, which consists in the process of showing that a system specification (model) is compliant with some logic property. This can be obtained through a systematic exploration of the set of reachable states, verifying that the desired property holds for each of these.

Different logics have been designed and adapted for Model Checking, starting from the classical *Linear Temporal Logic* and *Computation tree logic*. These, however, are limited to discrete-time models. For this reason, different real-time temporal logics (logics designed for a continuous time environment) have been proposed. In this document we will focus on the *Signal Temporal Logic* (*STL*), a logic derived from the *Metric Interval Temporal Logic*, both real-time temporal logics.

We present `OnMonGen` (*Online Monitor Generator*), a tool which allows the verification of *STL* properties on a *Simulink* model. It does so by creating a library of *Simulink* blocks, starting from properties specifications. Such blocks can be added to *Simulink* models and used, during simulation, to monitor if these properties hold.

Contents

1 Introduction

Model Checking is a formal method which can be used in the verification of a system. The usual process behind Model checking is the following: a model specifying the system's behaviour is built, and on it a certain logic property is tested. Typically a model is said to satisfy a certain property if, for all its possible behaviour, such property holds. Common logics used in Model Checking for properties specification, are:

***Linear Temporal Logic* (*LTL*):** A temporal logic which extends propositional or predicate logic, describing how the system should behave during its execution. The use of *Linear* in the name is used to delineate that this logic is path-based, and so time is viewed in a linear manner.

***Computation tree logic* (*CTL*):** This logic is based on a *branching* notion of time, it allows, differently form *LTL*, properties do not need to be satisfied for all possible computation, but for some desired subset, described by the logic's operators.

Note:
Model Checking can only test if the system model is correct. If such model accurately represent all the system behaviour is another matter altogether.

Sometimes, for reason of size or unbounded variables, building these models is not a plausible option [3]. In those scenarios, Run-Time Verification techniques can be used: instead of checking all possible system's behaviours, a way of determining if a particular system behaviour is compliant with the interested property is adopted, using monitor (tester, observer) for such property. Those are typically some kind of software module. The task of exhaustively tests each behaviour of the system can then be delegated to a test generation procedure. This verification method can be used together with a simulation software like *Simulink*, which is able to simulate such system's behaviour (obviously this means that it must be possible to represent such system through a *Simulink* model).

We will present `OnMonGen` (*Online Monitor Generator*[1]), a tool which can be used with such an approach: if executed, with a set of temporal logic properties as input, it will be able to produce a collection of *Simulink* blocks (grouped inside a *Simulink* library), that, when added to a *Simulink* Model, will be able to test if those properties hold during the model's simulation.

[1] *Online* in this context means that the monitors are an internal component of the tested model.

2 Background

2.1 Simulink

Note:

This section is simply a synthesis of part 3 (*How Simulink Works*) of the *Simulink® User's Guide* [1]. The reader should refer to it for more in detail explanations.

The *Simulink* software is used for simulating and analyzing dynamic systems, which are those systems whose output changes and depends over time. Such systems can be use to model a wide variety of real world process, like: electrical circuits, braking systems, shock absorbers and many others. The use of *Simulink* can be logically divided in two steps:

Model Design This consists in the realization of a mathematical model, represented through a *Block Diagram*. Such a diagram is able to illustrate time-dependent relations between the system's inputs, outputs and states (Example in Figure 1).

Model Simulation This consists in the simulation of the system for a certain time interval. The simulation requires a first and a last instant.

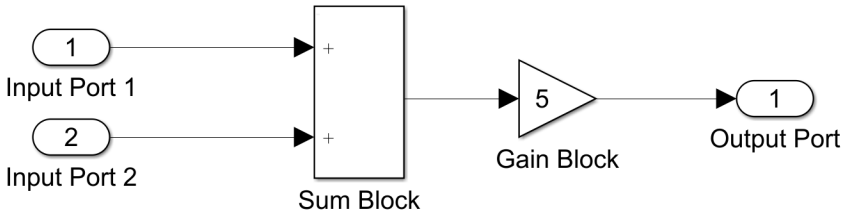
2.1.1 Block Diagram Semantics

A *Block Diagram* is mainly composed by two types of objects:

- Blocks
- Lines (Signals)

The set of all the blocks and lines in a diagram represents a dynamic system. A block itself represents a dynamic sub-system. Indeed we have that a block can either inductively contain another block diagram or be a *built-in*[2] block. This kind of behaviour allows for more readable, and sometimes more expressive, diagrams. On the other hand, signals in a diagram are used to describe relationships between its dynamic sub-systems (blocks). Usually diagrams and blocks are provided with *Input* and *Output* ports, which represent the inputs and outputs of the underlying dynamic system. If we attach a signal from an output port of a block b_1 to an input port of (possibly) another block b_2 than the dynamic system represented by b_2 has as input the output of the dynamic system of b_1.

Figure 1: Simple *Simulink* model



Example Let us consider the diagram in Figure 1. It is mainly composed by two blocks: *Sum Block* and *Gain Block* (the other blocks are simply place holders used to indicate input and output ports). Those two blocks intuitively represent the sum between two input signals and the multiplication with a constant respectively. Both are built-in blocks and represent the following dynamic systems:

Sum Block

$$y(t) = u_1(t) + u_2(t) \tag{2.1}$$

Gain Block

$$y(t) = u(t) \cdot 5 \tag{2.2}$$

With y and u indicating the output and input respectively. Note, since this is a dynamic system, that they are both time-dependent values (t represents time). The signal between *Sum Block* and *Gain Block* imposes that the output of *Sum Block* is the input of *Gain Block*. The complete dynamic system of the model is thus the following:

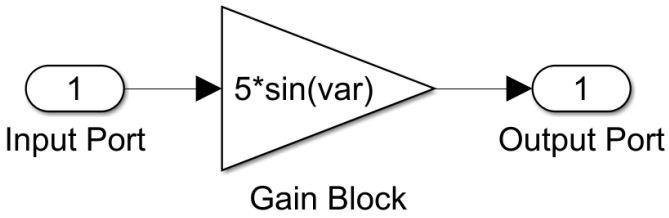
$$y(t) = (u_1(t) + u_2(t)) \cdot 5$$

2.1.2 Time

A *SimulinkBlock Diagram* describes time-based relationships between blocks and signals. By evaluating all these relationships over a given time interval it is possible to obtain the solution of the block-diagram. Specifically a block diagram represents the

[2]A block whose behaviour is hard-coded directly in *Simulink*, an example could be a *Sum* block.

Figure 2: A gain block with as parameter the expression: $5 * sin(var)$



behaviour of the system for a certain simulation instant. Thus in order to determine the system behaviour over time we need to solve the model repeatedly at intervals (called time steps), from the start time to the end time. This process, (i.e. solving the problem at successive intervals), is called the system's simulation.

2.1.3 Parameters

Each block in *Simulink* can contain *parameters*: a parameter of a block is a parameter of the underlying dynamic system, and is used to compute the output of the block. Parameters are usually *MATLAB* expressions, which, right before the simulation start, are evaluated and passed to their respective blocks.

Example A simple example could be the block *Gain*, which multiplies a signal by some constant value. Such value is a parameter of the block (Figure 2).

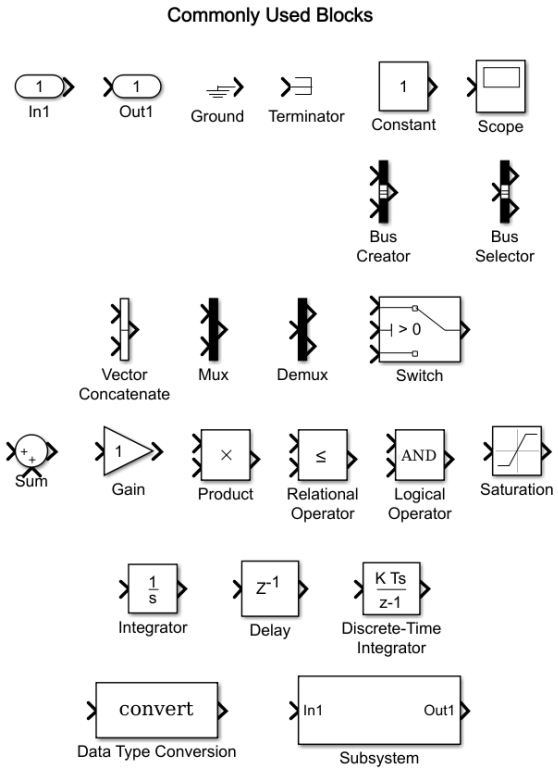
2.1.4 Signals

A *Simulink signal* can be seen as a time-varying value (i.e. signals are functions of time). In *Simulink*, signals have a range of attributes: *name*, *data type*, *numeric type* (real or complex) and *dimensionality*. Many blocks impose constraints on the type of signals that they can receive as input. Usually a block which has only output signals is called a *source*, while a block with only input signal can be called a *sink*.

2.1.5 S-Functions

An *S-Function Block* (which stands for *System Function Block*) is a particular type of block which allows great customization. Specifically, an S-Function Block is a wrap-

Figure 3: *Simulink* Library containing the most used blocks



per to a certain *MATLAB* or *MEX*[3] file containing the actual implementation of the dynamic system function (this is the reasoning behind the name *System Function*). Such files are called S-functions.

2.1.6 Block Libraries

A *Simulink Block Library* is a collection of blocks that can be added to a *Simulink* model (Example in Figure 3). More specifically, it is possible to search within libraries using the *Simulink Library Browser* (which is a pane that allows the user to search trough the libraries of blocks currently installed) and copy a block from the library into the user's model. *Simulink* grants access to a number of standard libraries, containing the basic and most common blocks, however it possible for users to create and modify custom libraries.

[3] *MEX* stands for *MATLAB Executable* and it's a file that can be executed through the *MATLAB* software. The source code of such programs are typically written in *C*, *C++* or *Fortran*.

2.2 Temporal Logics

> "In mathematics, logic is static. It deals with connections among entities that exist in the same time frame. When one designs a dynamic computer system that has to react to ever changing conditions, ... one cannot design the system based on a static view. It is necessary to characterize and describe dynamic behaviors that connect entities, events, and reactions at different time points. Temporal Logic deals therefore with a dynamic view of the world that evolves over time." Amir Pnueli (Translated from the original Hebrew)

This is a quote from Amir Pnueli after receiving the *Israel Prize* in the field of Computer Science. Pnueli was one of the most important researchers behind temporal logic applied to Model Checking. This quote is important because it shows the need to specify requirements (regarding verification of dynamic systems, like a concurrent program or a cyber-physical system) in a new way, different from the typical logic used in mathematics. Indeed any kind of temporal logic aims to describe the system's behaviour, which is in stark contrast with some other formal methods used in the verification of systems. An example could be the *Hoare Logic*, which is used to verify static properties reasoning with *pre* & *post* conditions of systems. *Hoare Logic* could be a very useful tool for verification of a certain class of procedures, it is however unable to express properties on programs like a server, which (theoretically) is never going to stop execution, but still must be compliant with certain properties (e.g. it must answer to a client in less than a given threshold). Overall we could say that temporal logics try to formalize those kind of statements, where the satisfaction of a property depends on the execution of a system, not its output. One of the classical temporal Logic used in Model Checking, and system verification in general, is the *Linear Temporal Logic*. It is a kind of logic focused on describing properties which must be valid for all the possible computation path that a system can produce.

2.2.1 *Linear Temporal Logic*

As already stated, the *Linear Temporal Logic* (*LTL*) is a logic used to specify temporal properties whose satisfaction is defined over a system's sequence of states. These sequences are often referred to as *computation paths* or *traces*. An *LTL* property describes then relationship between states that can occur during a system execution. For example, we are able to express in this logic statements like:

- In the future it must be true that the system reaches a state where properties P_1 and P_2 are valid.

- The system stays in the state where P_1 yields, until it reaches a state such that P_2 is true.
- Currently in the system properties $P_1 \dots P_n$ hold.

The main *LTL*'s operators ar the following:

$$\begin{aligned} \Diamond\, \phi \quad & \text{(Future operator)} \\ \Box\, \phi \quad & \text{(Globally operator)} \\ \phi_1\, \mathcal{U}\, \phi_2 \quad & \text{(Until operator)} \end{aligned}$$

Their meaning can be described as:

$\Diamond\, \phi$: Property ϕ must be true in a future state.

$\Box\, \phi$: Property ϕ must always hold for each future state.

$\phi_1\, \mathcal{U}\, \phi_2$: Property ϕ_1 must yield for each future state, until a state such that ϕ_2 holds is crossed.

2.2.2 *Metric Interval Temporal Logic*

Unfortunately, *LTL* cannot be easily applied to dense-time systems (i.e. like a pendulum or a bouncing ball). This is caused by the fact that *LTL* is designed for discrete system, so in order to test *LTL* properties on dense-time systems, they must be explicitly discretized. obviously such thing is not optimal. One of the many answers to this problem is the *Metric Interval Temporal Logic* (*MITL*)[2]: *MITL* is an evolution of the *LTL*, proposed by R. Alur, T. Feder and T.A. Henzinger, which generalizes *LTL*'s semantics in a continuous-time environment. We will see in this document a fragment of this logic called $MITL_{[a,b]}$, which by applying some restrictions allows an useful property: having a sufficiently long trace prefix, it is possible to determine an $MITL_{[a,b]}$ formula satisfaction for all the traces with such prefix. This is not, generally speaking, valid for *LTL* and *MITL* formulae.

2.3 $MITL_{[a,b]}$

We describe in this section $MITL_{[a,b]}$[3], a fragment of *MITL*. Before we can formally define this logic, we need to give some simple definitions:

Definition 2.1. *A* Finite Length Signal *over a set $\mathcal{D}$ is a function $s : [a, b) \to \mathcal{D}$ such that $a \leq b$.*

We call an *n-dimension signal* an n-tuple: $(s_1, \ldots, s_n)$, such that $\{s_i\}_{i=1..n}$ a sequence of one dimensional signal. To get a one dimension signal from an n-dimension signal s we use the projection operator $\pi_k(s)$.

Definition 2.2. *Let $s = (s_1, \ldots s_n)$ be an n-dimensional signal, the projection of s over its i-th coordinate is:*

$$\pi_i(s) = s_i$$

Note:
If during this document we use the notation: $\pi_x(s)$, where $x \notin \mathbb{N}$, then we are assuming that exists an implicit encoding enc, mapping from object of the same type of x to a value in $\mathbb{N}$, such that $\pi_{\mathrm{enc}(x)}(s)$ is well defined.

In the rest of this document we will assume, when using boolean signal (i.e. signals where $\mathcal{D} = \{0, 1\} = \mathbb{B}$), to be using piecewise constant signals.

2.3.1 Syntax and Satisfaction

The *MITL*$_{[a,b]}$'s syntax is described by the following inductive definition:

$$\phi ::= \rho \mid \neg\phi_1 \mid \phi_1 \vee \phi_2 \mid \phi_1 \wedge \phi_2 \mid \phi_1 \,\mathcal{U}_{[\alpha,\beta]}\, \phi_2 \mid \Diamond_{[\alpha,\beta]}\, \phi_1 \mid \Box_{[\alpha,\beta]}\, \phi_1$$

(with ρ a proposition, $\alpha \in \mathbb{R}$, $\beta \in \mathbb{R}$ and such that $0 \leq \alpha < \beta$)

We describe now how an *MITL*$_{[a,b]}$ formula ϕ is interpreted over an n-dimensional signal s (we sometime refer to such signal as *signal trace*). We are assuming that ϕ contains at most n propositions ρ, each with index $\mathrm{idx}(\rho)$. The satisfaction relation ($(s,t) \models \phi$) tells then if ϕ is valid over s, starting from the instant t. Such relation is defined in the following inductive fashion:

$$\begin{aligned}
(s,t) \models \rho &\iff \big(\pi_{\mathrm{idx}(\rho)}(s)\big)(t) = 1 \\
(s,t) \models \neg\phi_1 &\iff (s,t) \not\models \phi_1 \\
(s,t) \models \phi_1 \vee \phi_2 &\iff (s,t) \models \phi_1 \text{ or } (s,t) \models \phi_2 \\
(s,t) \models \phi_1 \wedge \phi_2 &\iff (s,t) \models \phi_1 \text{ and } (s,t) \models \phi_2 \\
(s,t) \models \phi_1 \,\mathcal{U}_{[\alpha,\beta]}\, \phi_2 &\iff \exists t' \in [t+\alpha, t+\beta]\,(s,t') \models \phi_2 \text{ and } \forall t'' \in [t,t']\,(s,t'') \models \phi_1 \\
(s,t) \models \Diamond_{[\alpha,\beta]}\, \phi_1 &\iff \exists t' \in [t+\alpha, t+\beta]\,(s,t') \models \phi_1 \\
(s,t) \models \Box_{[\alpha,\beta]}\, \phi_1 &\iff \forall t' \in [t+\alpha, t+\beta]\,(s,t') \models \phi_1
\end{aligned}$$

It can be easily seen that the operators $\wedge$, $\Box_{[\alpha,\beta]}$ and $\Diamond_{[\alpha,\beta]}$ are not necessary in order to describe a particular property and do not enrich the logic semantics. Indeed we have

that the following holds:

$$(s,t) \models \Diamond_{[\alpha,\beta]}\, \phi \iff (s,t) \models \mathit{true}\, \mathcal{U}_{[\alpha,\beta]}\, \phi$$
$$(s,t) \models \Box_{[\alpha,\beta]}\, \phi \iff (s,t) \models \neg \Diamond_{[\alpha,\beta]}\, \neg\phi$$
$$(s,t) \models \phi_1 \wedge \phi_2 \iff (s,t) \models \neg(\neg\phi_1 \vee \neg\phi_2)$$

For this reason in the future we will restrict our operators only to the ones below:

$$\phi ::= \rho \mid \neg\phi_1 \mid \phi_1 \vee \phi_2 \mid \phi_1\, \mathcal{U}_{[\alpha,\phi]_2}$$

Minimum signal length Note that from how we defined the $MITL_{[a,b]}$ semantics, when determining whether $(s,t) \models \phi$ is satisfied, we could find ourself trying to determine s for an instant not inside its domain. To avoid this undefined behaviour the function $\#(\phi)$ is used (introduced in [3]). Such function is recursively defined as follows:

$$\#(\phi) = \begin{cases} 0 & \phi = \rho \\ \#(f_1) & \phi = \neg f_1 \\ \max(\#(f_1), \#(f_2)) & \phi = f_1 \vee f_2 \\ \beta + \max(\#(f_1), \#(f_2)) & \phi = f_1\, \mathcal{U}_{[\alpha,\beta]}\, f_2 \end{cases}$$

Note:
For each $t \in [a, b - \#(\phi))$ the relation $(s,t) \models \phi$ is well defined, while for other instants is not. This means that in order to correctly test a formula for a signal s, such signal length must be greater than $\#(\phi)$.

An interesting $MITL_{[a,b]}$ property is the following: given a formula ϕ and a signal $s : [a, b + \#(\phi))$, we have that for all the signal s' "containing"[4] s, ϕ's satisfaction is the same, at least for a certain interval (for all instants in $[a, b)$).

Claim 2.1. *Given a formula ϕ and a signal $s : [a, b + \#(\phi)) \to \mathbb{B}$, for all boolean signal s', such that:*

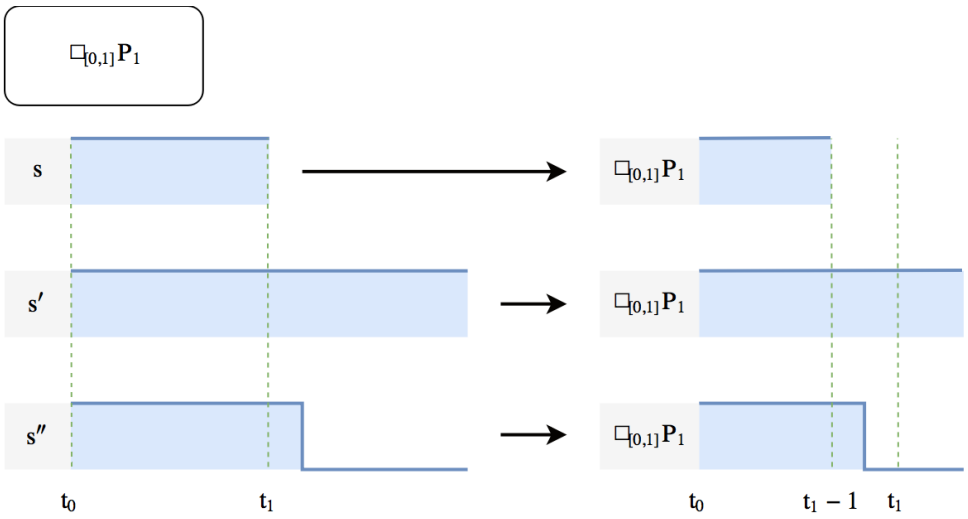
$$\mathrm{dom}(s) \subset \mathrm{dom}(s')$$
$$\forall t \in \mathrm{dom}(s) : s'(t) = s(t)$$

It holds, $\forall t \in [a, b)$, the following:

$$(s,t) \models \phi \iff (s',t) \models \phi$$

[4] i.e. such that $\mathrm{dom}(s) \subset \mathrm{dom}(s')$ and with $s'(t) = s(t)$ for each t in $\mathrm{dom}(s)$

Figure 4: Example of an instant inside a prefix signal s domain, such that the satisfaction of the extending signals s', s'' changes.



This property let us use prefix signals to assert satisfaction, for same instants, over all the extending signals. This can be used to divide the computation of a formula satisfaction over a sequence of smaller signals, instead of a single long one.

Observation:

For the instants $t \in [b, b + \#(\phi))$, the truth value of $(s', t) \models$ may depends over s' (visual example in Figure 4).

2.3.2 Monitoring Algorithm

Consider the problem to find all the instants where a certain $MITL_{[a,b]}$ formula ϕ, given an n-dimensional signal $s : [a, b) \to \{0, 1\}$, is valid.
Formally we ask all the instants $t \in [a, b)$ such that:

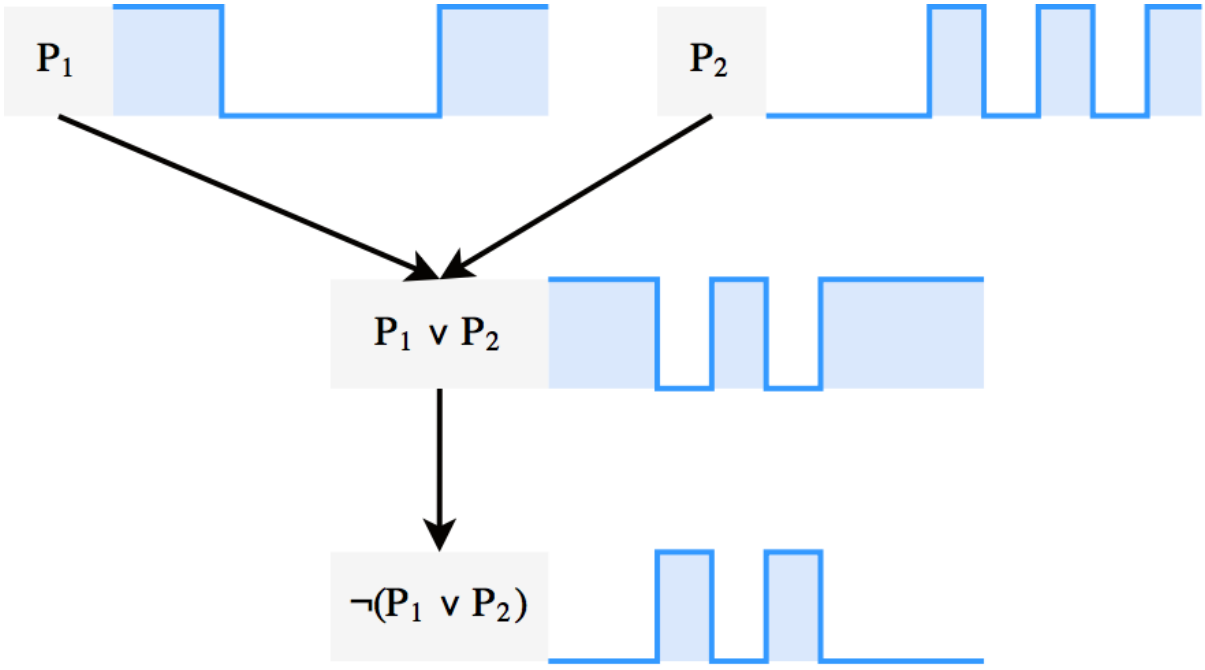
$$(s, t) \models \phi$$

An algorithm that solves this problem has been proposed by Maler and Nickovic [3]. We describe it here and refer to it with the name `monitor`. The algorithm inputs are:

- an n-dimensional boolean signal (which we will usually refer to as s);
- an $MITL_{[a,b]}$ formula (referred to as ϕ).

The algorithm's output is a boolean signal (indicated with o), such that: if the formula under verification holds ($(s, t) \models \phi$) for an instant t, then $o(t) = 1$. Otherwise

Figure 5: Algorithm Execution on formula: $\neg(P_1 \vee P_2)$



$o(t) = 0$. This algorithm works in an inductive fashion, constructing signals indicating when sub-formulae are true, and using such signals to compute the output (Example in Figure 5).

Consistent Interval Coverings Before continuing on the algorithm description we must first define how it is possible represent a boolean piecewise signal through a set of intervals $\mathcal{I}$.

Definition 2.3. *An* Interval Covering *over $I = [a, b)$ is a set of intervals $\mathcal{I} = \{I_1, I_2 \dots I_n\}$ such that:*

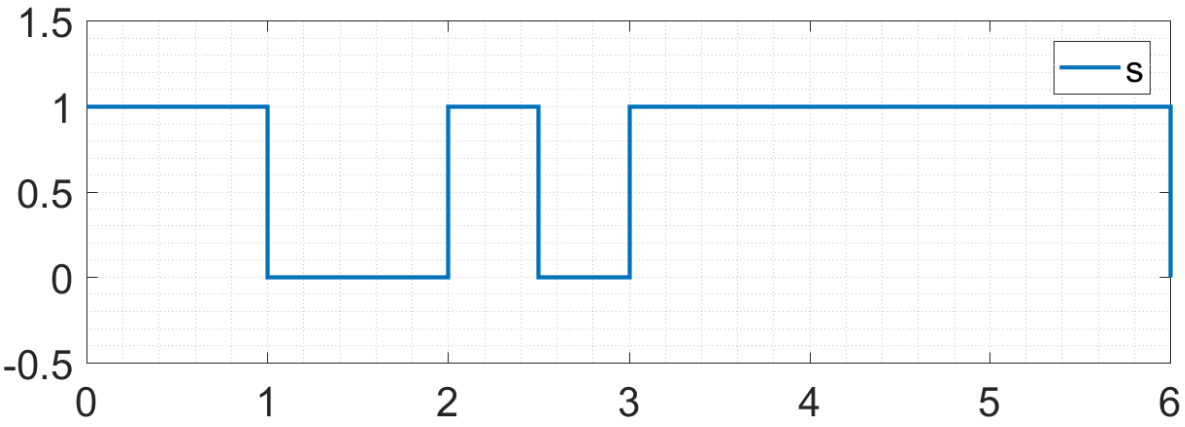
$$\bigcup_{i=1}^{n} I_i = [a, b)$$
$$\forall i, j : i \neq j \implies I_i \cap Ij = \emptyset,$$

An interval covering $\mathcal{I}$ over $[a, b)$ is *consistent* with a signal $s : [a, b) \to \mathbb{B}$ if and only if $s(t) = s(t')$, for each t,t' in the same interval I_i. For that reason we abuse notation and use $s(I_i)$.

Definition 2.4. *Given a signal $s : [a, b) \to \mathbb{B}$ we say that:*

1. *$\mathcal{I}_s$ it's the minimal interval covering consistent with s.*
2. *$\mathcal{I}_s^+$ it's the set of interval with image 1 (i.e. $\{I \in \mathcal{I}_s : s(I) = 1\}$).*

Figure 6: Interval covering example



$\mathcal{I}$ consistent with s	$\{[0,1), [1,1.35), [1.35,2), [2,2.5), [2.5,3), [3,3.2), [3.2,6)\}$
$\mathcal{I}_s$	$\{[0,1), [1,2), [2,2.5), [2.5,3), [3,6)\}$
$\mathcal{I}_s^+$	$\{[0,1), [2,2.5), [3,6)\}$
$\mathcal{I}_s^-$	$\{[1,2), [2.5,3)\}$

3. *$\mathcal{I}_s^-$ it's the set of interval with image* 0 *(i.e.* $\{I \in \mathcal{I}_s : s(I) = 0\}$*).*

Any boolean signal $s : [a,b) \to \mathbb{B}$ can be represented by the pair $([a,b), \mathcal{I}_s^+)$.

Definition 2.5. *A signal s is* unitary *if $\mathcal{I}_s^+$ is a singleton.*

Algorithm Explanation We describe here the behaviour of `monitor`. This algorithm works in an inductive manner over the input formula's structure. For this reason it is defined by cases (each for an $MITL_{[a,b]}$ operator[5]).
Regarding the output we have that, if the algorithm's inputs are an n-dimensional signal $s : [a,b) \to \mathbb{B}^n$ and a formula ϕ, then the output is a boolean signal o such that:

$$\forall t \in \mathrm{dom}(o) : o(t) = 1 \iff (s,t) \models \phi$$

Note:
We will usually refer to $s = (s_1 \dots s_n)$ and o through their pair representation: $([a,b), (\mathcal{I}_{s_1}^+, \dots \mathcal{I}_{s_n}^+))$ and $([a,b), \mathcal{I}_o^+)$ respectively.

[5]We use the restricted version of the logic, without the redundant operators.

Propositions Propositions are the algorithm's base case. Typically for a proposition ρ, computing the output signal o is an easy feat: indeed o is simply the projection of s over the coordinate at position $\mathrm{idx}(()\rho)$ (we are assuming that $PropositionIndex(\rho)$ indicates the ρ's index in the super-formula under test ϕ). Formally speaking, we say that:

$$\forall t \in \mathrm{dom}(s) : o(t) = \big(\pi_{\mathrm{idx}(\rho)}(s)\big)(t)$$

For example if we want to tests the formula $\phi = P_1 \vee P_2$ over the signal $s = (s_1, s_2)$, then when the algorithm recursively calls: $o = \texttt{monitor}(s, P_2)$, we will have that $o = s_2$.

Formula Negation Suppose that the following yields:

$$\phi = \neg\phi_1$$

Then ϕ is the negation of another formula ϕ_1. As we already stated this kind of formulae are true only if the sub-formula is false. Suppose that o_1 is the output function obtained by executing:

$$o_1 = (D, \mathcal{I}^+_{o_1}) = \texttt{monitor}(s, \phi_1)$$

Then the output signal o can be easily computed. That is because we have that $o(t) = 1 \iff o_1(t) = 0$ so if we express o through it's covering then we can simply indicate o as:

$$o = (D, \mathcal{I}^-_{o_1})$$

Formula Disjunction Suppose to be in the inductive case where:

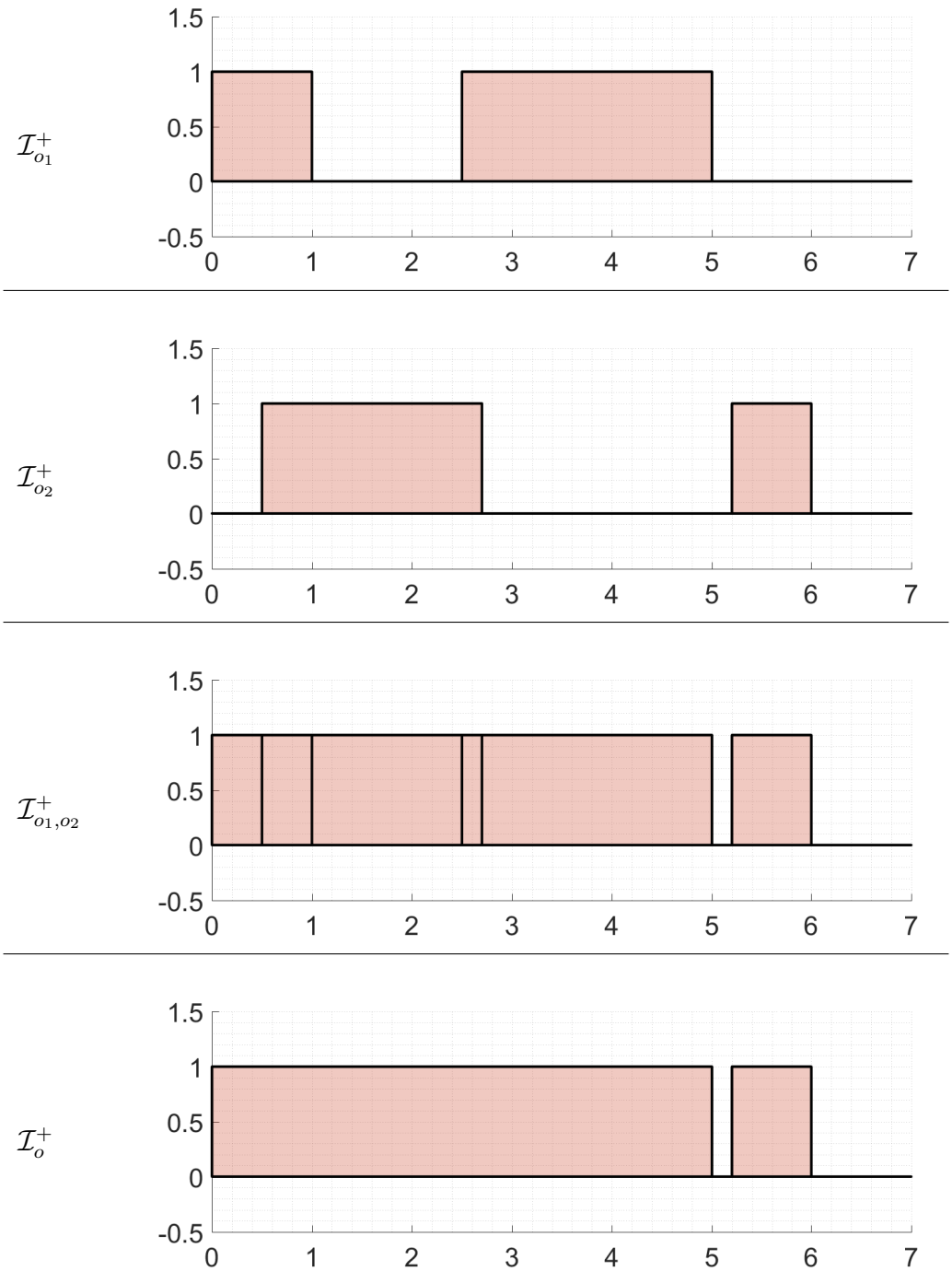
$$\phi = \phi_1 \vee \phi_2$$

In other words ϕ is the disjunction of two formulae. This means that ϕ is true for a certain n-dimensional signal s and instant t, if and only if at least one between ϕ_1 or ϕ_2 is valid. This means that the set of instants such that ϕ is satisfied can be simply seen as the union between the instants where ϕ_1 is valid and the instants where ϕ_2 is valid. Let:

$$o_1 = (D, \mathcal{I}^+_{o_1}) = \texttt{monitor}(s, \phi_1)$$
$$o_2 = (D, \mathcal{I}^+_{o_2}) = \texttt{monitor}(s, \phi_2)$$

We have that intuitively the output signal $o = (D, \mathcal{I}^+_o)$ can be obtained by creating a set containing all elements in $\mathcal{I}^+_{o_1} \cup \mathcal{I}^+_{o_2}$, and then merging all the adjacent and intersecting intervals (Example in Figure 7). We describe now the approach used in [3] to compute

Figure 7: Disjunction signal example



the output. Let $\mathcal{I}_{o_1,o_2}$ be the maximal interval covering consistent with both o_1 and o_2. To compute the output signal we need both intervals in $\mathcal{I}^+_{o_1}$ and $\mathcal{I}^+_{o_2}$. We formalize that notion using the following equation:

$$\mathcal{I}^+_{o_1,o_2} = \{I_i \in \mathcal{I}_{o_1,o_2} : I_i \subseteq \mathcal{I}^+_{o_1} \cup \mathcal{I}^+_{o_2}\}$$

We say that $\mathcal{I}^+_o$ is obtained by taking the elements in $\mathcal{I}^+_{o_1,o_2}$ and merging all adjacent intervals (Example in Figure 7).

Temporal Formula Suppose to be in the inductive case, where the following holds:

$$\phi = \phi_1 \, \mathcal{U}_{[\alpha,\beta]} \, \phi_2$$

Note:
To compute the satisfaction of $\phi_1 \, \mathcal{U}_{[\alpha,\beta]} \, \phi_2$ we need to define the function θ:

$$[m, n) \; \theta \; [a, b] = [m - b, n - a)$$

We assume also, to have already recursively invoked:

$$o_1 = \texttt{monitor}(s, \phi_1)$$
$$o_2 = \texttt{monitor}(s, \phi_2)$$

We (for now) restrict ourself to the case where o_1 and o_2 are unitary ($o_1 = (D, \{I_{o_1}\})$ and $o_2 = (D, \{I_{o_2}\})$). With this assumption we have then that o is unitary or empty, and defined as follows [3]:

$$\mathcal{I}^+_o = \{((I_{o_1} \cap I_{o_2}) \; \theta \; [\alpha, \beta]) \cap I_{o_1}\}$$

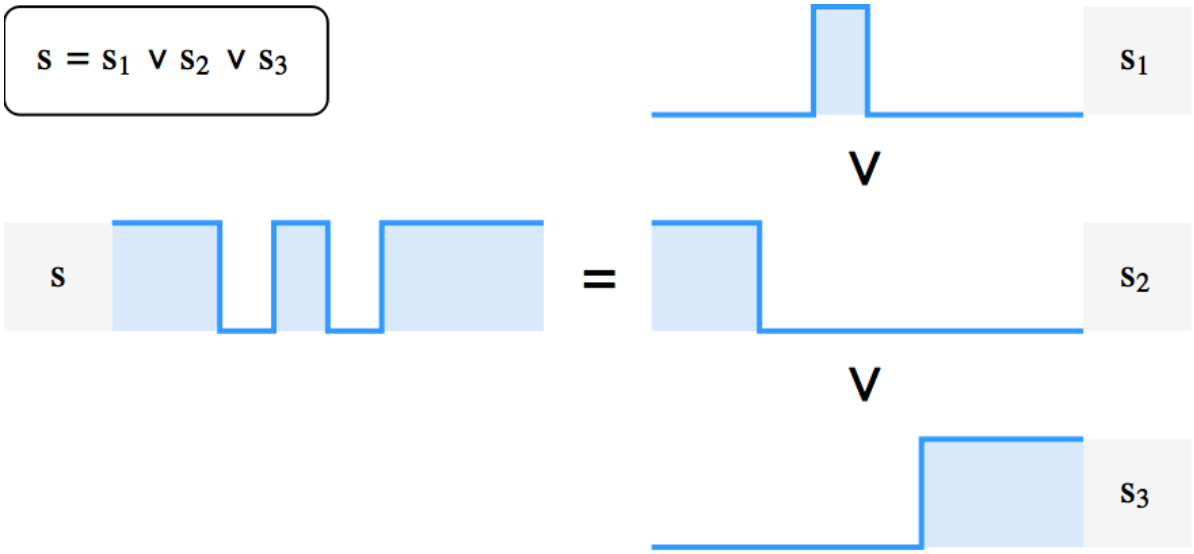
Observation:
We can represent every, not constantly equal to 0 and piecewise constant, boolean signal as disjunction of unitary signals (Example in Figure 8).

We now describe the general case, where $\mathcal{I}_{o_1}+$ and $\mathcal{I}_{o_2}+$ are not singleton. We first observe that if at least one of o_1 or o_2 is constantly equal to 0, then the output signal o must be equal to 0 for all t in its domain. This let us consider a case where both o_1 and o_2 are not always equal to 0. We describe then such signals as disjunction of unitary signals:

$$o_1 = \bigvee_{i=0}^{k_1} o_{1,i} \qquad\qquad o_2 = \bigvee_{j=0}^{k_2} o_{2,j}$$

The output signal is then computed using the following until operator's property:

Figure 8: Unitary signal union example



Claim 2.2. *[3] Let $p = p_1 \vee \cdots \vee p_m$ and $q = q_1 \vee \ldots q_n$ be two signals (each written as an union of unitary signals). Then it holds that:*

$$p \, \mathcal{U}_{[\alpha,\beta]} \, q = \bigvee_{i=1}^{m} \bigvee_{j=1}^{n} p_i \, \mathcal{U}_{[\alpha,\beta]} \, q_j$$

To obtain the output signal o we simply need to compute for each unitary signal $o_{1,i}$ in o_1 and $o_{2,j}$ in o_2 the unitary until $o_{1,i} \; \mathcal{U}_{[\alpha,\beta]} \; o_{2,j}$ and then compute the resulting intervals' disjunction.

2.3.3 *Signal Temporal Logic*

We now generalize the $MITL_{[a,b]}$ so that we are able to determine satisfaction over real-valued signals instead of only boolean ones. This will require a slightly change in the logic definition. We will call the resulting logic *Signal Temporal Logic* (*STL*) and we will see how to reduce the satisfaction of an *STL* formula to a $MITL_{[a,b]}$ one.

Syntax The *STL* syntax is very similar to what we already saw in the $MITL_{[a,b]}$. The only real difference consists in using predicates (defined over a set of variables) instead

of simple propositions. Such predicates must be defined in the following form[6]:

$$\rho ::= \sum_{i=0}^{n} \alpha_i \cdot x_i \; \mathcal{R} \; \beta \mid \textit{true} \mid \textit{false}$$
$$\mathcal{R} ::= \; <|\leq|>|\geq|=|\neq$$

We say that a predicate $\rho = \sum_{i=0}^{n} \alpha_i \cdot x_i \; \mathcal{R} \; \beta$ is valid regarding a certain k-tuple w if and only if:

$$\sum_{i=0}^{n} \alpha_i \cdot \pi_{x_i}(w) \; \mathcal{R} \; \beta$$

Satisfaction Before describing how to determine the satisfaction of an *STL* formula we must define how it is possible, starting from a real-valued signal and a predicate, to produce a boolean signal.

Definition 2.6. *Given an m-dimensional real-valued signal μ and a predicate ρ (defined over at least m variables) we will use the notation $\rho(\mu)$ to describe a boolean signal with domain equal to $\mathrm{dom}(\mu)$ and such that:*

$$\Big(\rho(\mu)\Big)(t) = 1 \iff \rho \textit{ is valid over } \mu(t)$$

Assume to have a certain *STL* formula ϕ, whose predicates are the sequence $\rho_1 \dots \rho_n$, and the equivalent $MITL_{[a,b]}$ formula ϕ' (having propositions instead of predicates). We define then the satisfaction of ϕ regarding a certain m-dimensional signal $\mu = (\mu_1, \dots \mu_m)$ starting from the instant t in the following way:

$$(\mu, t) \models \phi \iff (s, t) \models \phi'$$

with s defined as:

$$s = \big(\rho_1(\mu), \dots, \rho_n(\mu)\big)$$

Note:
We will sometimes refer to μ as the *simulation trace*. In our context μ will be always represented by the sequence of states reached by the *Simulink* model during execution.

[6] In the original *STL* definition [3] the predicates' structure is not defined. We choose a different approach and restrict our-self to predicates indicating constraints on linear combinations of variables. The reason behind this choice is that it allows for an easy *Simulink* implementation of such predicates.

Problem of representing real-valued signals Unfortunately representing a real-valued signal μ on a computer is a much more difficult task than representing a boolean one. Real-valued signals can be represented through a sequence of pairs[7] $(t_0, \mu(t_0)) \dots (t_k, \mu(t_k))$, this is at least the approach utilized by *Simulink*. For obvious reason this kind of approach is just an approximation of the actual function, and a crude one. Fortunately for our goals and with some reasonable assumption we can see that those representation do not impede determining the satisfaction of an *STL* formula. Indeed we can observe that the *STL* does not directly use the input signals' values. It first convert the signal to an n-dimensional boolean one, using the predicates ρ_k. We also can observe that as long as each of the predicates ρ_k's truth value remain constant over $\mu(t)$, we don't need the exact value of $\mu(t)$. Thus if we are sure to have a sampling such that each change in the predicates' truth values is "captured", then it is reasonable to think that it is possible to correctly determine the formula satisfaction. Formally: given a real-valued signal μ and a sampling T such that for each $t \in \mathrm{dom}(\mu)$ if

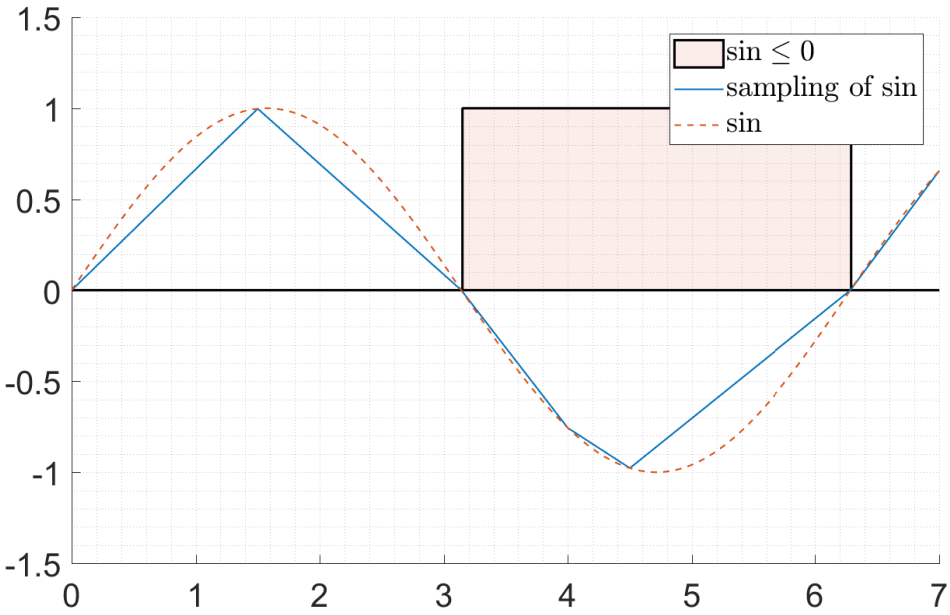
$$\lim_{t' \to t^-} \big(\rho_k(\mu)\big)(t') \neq \lim_{t' \to t^+} \big(\rho_k(\mu)\big)(t')$$

then t must be in T. Then even if we represent μ through T the loss of information won't change the formula satisfaction (Figure 9). *Simulink* automatically (at least for some blocks) chooses, during simulation, time-steps that allow it to pick the almost exact moment where a "*state event*"[8] occur. *Simulink* is able to grant such feature using the process called *Zero-Cross Detection*[1].

[7] We call the sequence $t_0 \dots t_k$ the sampling of μ.

[8] This is how an instant in the simulation interval where a discontinuity occur is usually called.

Figure 9: Example of a "good" sampling of the $\sin$ function.



3 The OnMonGen Tool

We describe here OnMonGen (*Online Monitor Generator*). OnMonGen is a tool that can be used for automatic system verification of *Simulink* models. If executed OnMonGen produces a library of property monitors (observer or tester), implemented through *Simulink* custom blocks, which can be used to check if a particular simulation trace (i.e. the states reached by a system during simulation), complies with a temporal specification, written in *STL*. These monitors can be copied on *Simulink* models and, after being correctly setup, be used to check validity of the desired properties.
To correctly execute, OnMonGen requires some informations, namely:

- File specifying what formulae the output monitors should test.
- Where to place the library file (file with extension *.slx*) containing these monitors.
- Name that such file will have.
- Name that the output library will have inside the *Library Browser*.

3.1 Script Options

To run OnMonGen a script, named "run" and whose interpreter is system dependant (*cmd* for Windows operating systems and *bash* for Linux and Mac OS), must be executed. Such script takes as input the following options:

-f This flag must be followed by the name (relative or absolute) of a file containing *STL* formulae. We will say more about this file's format in the next few pages. This option is mandatory.

-d This flag must be followed by the name (relative or absolute) of the directory which will contain the created library. Such directory must not already exist. If this option was not given then the system will automatically create a directory named **monitor_lib** and use it for storing the necessary files.

-n This flag must be followed by a string. Such string will be the name assigned to the library file (filename extension: *.slx*) containing our generated monitors. If this option was not passed to the system, then the library's name will automatically be equal to the directory's name.

-b This flag must be followed by the name that our aforementioned library will have inside the *Library Browser*. If this option was not given to OnMonGen then such name will be equal to the library file name.

Note:
If any of the above input strings has any white space character, then it should be surrounded by quotes (') or double quotes ("), when passed to the script.

3.2 Formulae File

We will describe here what format the formulae file, in input to `OnMonGen`, should comply.

Attention:
The formulae file *encoding* should be **ISO/IEC 8859-1** (sometimes called *latin1*). However if such file does not contain any special character and has a one-byte ASCII based encoding, then `OnMonGen` should be able to correctly parse it.

The file must contain a sequence of *STL* formulae, each divided from the next through character "`|`". Each formula can also be preceded by a name (which can contain white spaces), the latter separated by the former using "**:**".

3.2.1 Example:

`First Formula Name:`*formula* `|` *formula* `|` `Second Name:`*formula*

3.2.2 Formula Representation

The problem now is how to describe a certain *STL* formula through an unambiguous string? The answer is simple: each operator must be mapped to a specific keyword, and for each operator precedence rules must be set, in order to solve cases of ambiguity in such formula.

$$
\begin{aligned}
\neg\phi_1 &\rightarrow \texttt{NOT}\ \phi_1 \\
\phi_1 \vee \phi_2 &\rightarrow \phi_1\ \texttt{OR}\ \phi_2 \\
\phi_1 \wedge \phi_2 &\rightarrow \phi_1\ \texttt{AND}\ \phi_2 \\
\phi_1\, \mathcal{U}_{[\alpha,\beta]}\, \phi_2 &\rightarrow \phi_1\ \texttt{UNTIL[}\alpha\texttt{,}\beta\texttt{]}\ \phi_2 \\
\Diamond_{[\alpha,\beta]}\, \phi_1 &\rightarrow \texttt{FUTURE[}\alpha\texttt{,}\beta\texttt{]}\ \phi_1 \\
\Box_{[\alpha,\beta]}\, \phi_1 &\rightarrow \texttt{GLOBALLY[}\alpha\texttt{,}\beta\texttt{]}\ \phi_1
\end{aligned}
$$

Figure 10: Example of a formulae file content

```
Formula 1:
Globally[1,2](x<=0 And y>=0) Or y+2*x=1 Until[0,5] 5*z=50
|
FuTurE[5,10] position + velocity < 30
|
another formula: NOT v>=0.001 OR GLOBALLY[1e-2,0.1] p <= 2e5
```

Predicates We describe now how a predicates should be expressed. Let us consider a predicate ρ such as:

$$\rho = \sum_{i=0}^{m} \alpha_i x_i \; \mathcal{R} \; \beta$$

Then it must be represented as a string in the following manner:

$$\alpha_1 * x_1 + \ldots + \alpha_m * x_m \; \mathcal{R} \; \beta$$

With $\mathcal{R}$ equals to one of the following: =, $\sim$=, <=, >=, <, >. The constant values α_i can also be omitted (it is assumed a value of 1 in such case). If the predicate is a boolean constant than it can be simply expressed with keywords `TRUE` or `FALSE`.

Note:
Each of the keywords (`TRUE`, `FALSE`, `NOT`, `OR`, `AND`, `UNTIL`, `GLOBALLY` and `FUTURE`) is case insensitive. For example `UNTIL` is equivalent to `Until`, `UnTiL`, etc.

Operator Precedence and Associativity Operators precedence is described by Table 1: operators on the same row have same precedence, while lower rows have lower precedence than higher rows. Associativity of all binary operators is always to the left.

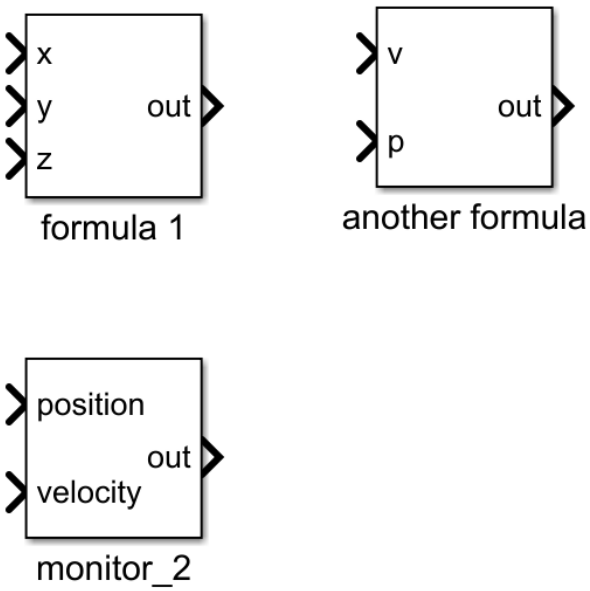
Table 1: Operator Precedence Table

1	`NOT` `GLOBALLY` `FUTURE`
2	`AND`
3	`OR`
4	`UNTIL`

For example, the formula (ϕ_1 `OR` ϕ_2 `OR` ϕ_3) is equivalent to the formula ((ϕ_1 `OR` ϕ_2) `OR` ϕ_3).

3.3 Output

Figure 11: Example of a generated library. This particular one was obtained by executing `OnMonGen` on the file showed in Figure 10.



We describe here how the result of a `OnMonGen` correct invocation should present itself to the user. We have that, during its execution, `OnMonGen` creates an output folder, with the name specified by the user (or the default one if no names was specified), containing a file *.slx*. This file is a *Simulink* block library. Its blocks represent property monitors for the properties specified in the formula file. More precisely, for each input file formula ϕ exists a monitor in the output library. Each of these monitors will have as name the one preceding ϕ inside the input file. If such name was not specified then a default one will be assigned to the monitor.

3.3.1 Using the output

We describe here how the output produced by `OnMonGen` can be used for automatic verification techniques. Suppose that we executed the system having:

monitor_lib_dir: as output directory;

monitor_lib: as library file;

Monitor Library Name: as the library's *Library Browser* name.

Note:
We will use the values t_{start} and t_{end} to indicate first and last simulation instants respectively.

The process for testing a *Simulink* model simulation is composed by four steps:

Add directory to search path Before anything else, we need to make sure that *Simulink* is able to access our library files and meta-files. This allows *Simulink*'s *Library Browser* to show our monitor library (with name "*Monitor Library Name*"). In order to do so we need to add the output directory to *MATLAB*'s *search path*. The *search path* is a list of directories which *MATLAB* visits to locate files. For example if a user wish to execute a *MATLAB* script then it (the script) must be placed inside the search path[9].

Copy monitors in model We now can open (in *Simulink*) the model under verification (MUV) and, browsing through the *Library Browser*, find our library. Then any monitor in *monitor_lib* can be copied inside the MUV simply through *Drag&Drop*.

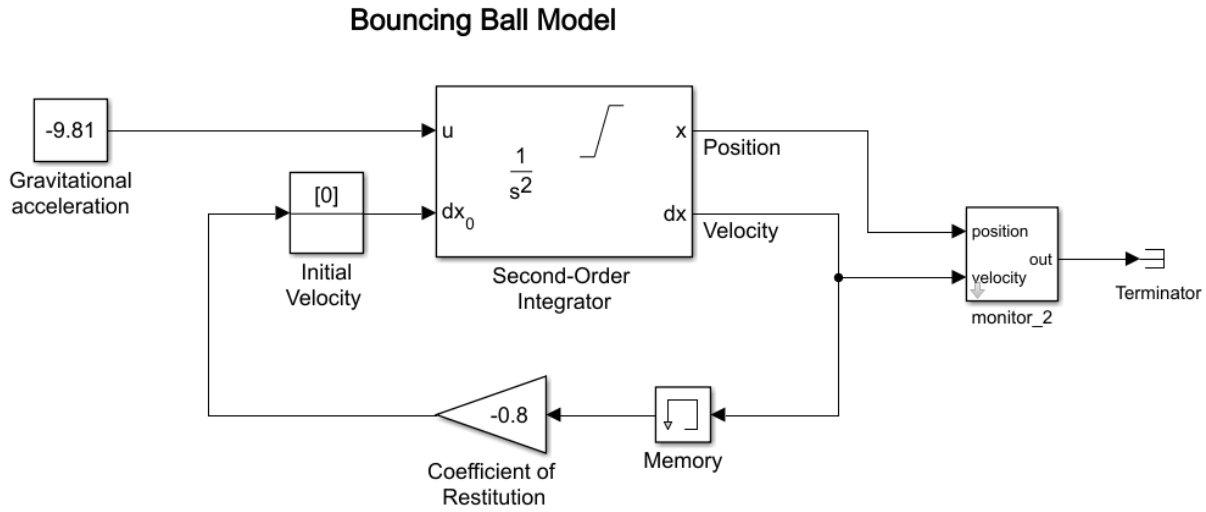
Monitors setup Once the required monitors have been copied in the MUV, we must setup their respective input and output signals. This is usually a fairly trivial and mostly consists in adding a line between the signal that we must verify to one of the monitor's input ports, each representing one of the monitored formula's variables. However sometimes a conversion must be effected, since the monitors' input ports only accept *scalar* signals with *data type double*.

Simulation If all the previous steps were successfully completed then it should be possible to check satisfaction of the interested *STL* properties using the output of their respective monitors. Indeed we have that if a formula is not satisfied during a simulation, then our monitor output signal will assumes value 1, otherwise 0 is constantly returned. Our monitors, however, suffer a limitation: suppose to have a monitor for the *STL* property ϕ, then, in order to have a correct result, the simulation duration ($t_{end} - t_{start}$) must be greater than $\#(\phi)$.

Attention:
In order to obtain a correct output signal, the monitor for a property ϕ must be simulated for a time greater than $\#(\phi)$!

[9] It can be also located in the *current directory*.

Figure 12: Example of monitor in a model



3.3.2 Interpreting the Monitors' Output

There is the problem of how to use these monitor in a correct way. To do so a better understanding of their outputs is required. We will now address this problem.

Note:

The output signal of every monitor is of *data type boolean (logical)*, consequently it can only assume value 0 or 1.

Suppose to have in your system a monitor for the property ϕ. We will say that, because of how the monitor is implemented, during the simulation instants between t_{start} and $t_{start} + \#(\phi)$ (included) its output signal will assume only value 0. The reason behind this behaviour is because in all those simulation instants the model did not produce a sufficiently long trace to let the monitor correctly determine ϕ's satisfaction for t_{start}. Just after the simulation instant $t_{start} + \#(\phi)$ (to be precise, after $t_{start} + \#(\phi) + \epsilon$, for an $\epsilon > 0$, and such that $t_{start} + \#(\phi) + \epsilon < t_{end}$) the monitor output signal will assume a value, which can be either 0 or 1, and constantly hold it until the end of simulation. Such value is used to indicate ϕ's satisfaction for the instant t_{start}. More precisely, if the output signal assumes:

- value 0 then ϕ is satisfied. More formally, if we define the simulation trace of our model as μ, the following must hold:

$$(\mu, t_{start}) \models \phi$$

- value 1 then ϕ is not satisfied. More formally, if we define the simulation trace of our model as μ, the following must hold:

$$(\mu, t_{start}) \not\models \phi$$

Observation:

The output signal may be not intuitive to some readers, since it assumes value 0, which is typically interpreted as false, to assert the property satisfaction, while it assumes value 1, which typically interprets true, to assert the formula unsatisfaction.

3.4 Limitations

`OnMonGen` suffer different limitations, we try now to describe a comprehensive list:

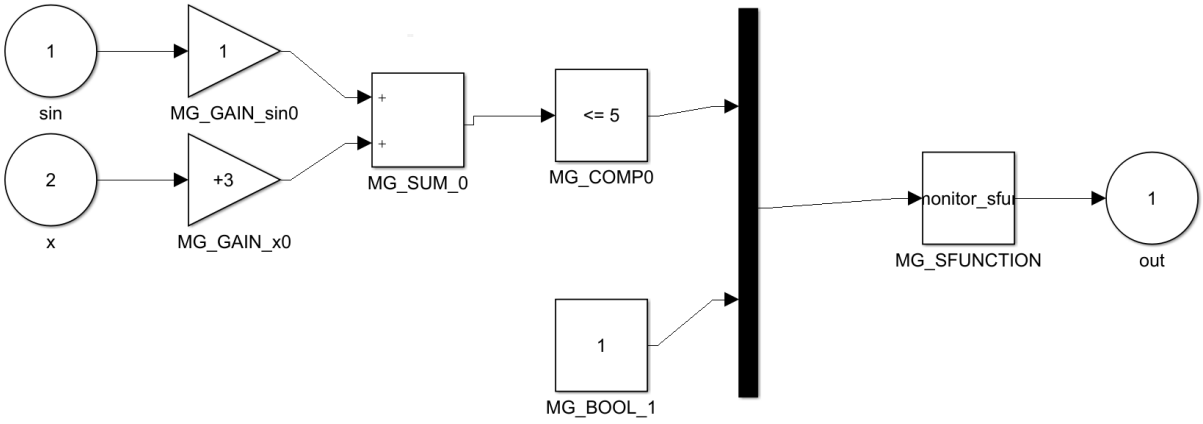
Required *MATLAB* The system is mainly composed by *MATLAB* scripts, which means that a *MATLAB* distribution must be installed in order to execute it. The release must be *R2017a* or older.

Required *C++ MEX* compatible compiler To work our system must compile some *C++* source files: this means that the *MATLAB MEX* command should be configured to work with a *C/C++* compiler.

Monitor placement These monitors cannot always be placed in sub-systems inside a block diagram: it is possible to place these monitors only inside *virtual or atomic* sub-system blocks [1].

No SimState It is impossible to save the model's state during a simulation, if it contains one of these property monitors. This behaviour is caused by an *S-function* block inside such monitors.

Figure 13: Monitor Internal Block Diagram. The formula tested by this monitor is: $\Box_{[0,5]}(sin + 3x \leq 5 \wedge true)$



4 Monitors Genereted by `OnMonGen`

In previous sections we described what are inputs and outputs of `OnMonGen` and the consequently generated monitors. Here we will focus on describing *how* these monitors produce the kind of output signals that they produce.

4.1 Monitor's Internal Organization

The property monitors generated by `OnMonGen` are themselves *Simulink* models[10]. An example of a monitor's block diagram can be seen in Figure 13. Generally speaking, the block types inside a monitor are the following:

Sum Block This block's output signal is the addition or subtraction of its input signals.

Gain Block This block's output signal is the multiplication of its input for a constant value. Such value is the block's parameter.

Relational Operator Block This block is provided with a parameter indicating a binary relation[11]. Such relation is then tested by the block on its input signals. If it holds then the outgoing signal assumes value 1, otherwise the assumed value is 0.

MUX Block This block takes as input different signals of the same *data* and *numeric* type. The output signal is the combination of all inputs into a single vector, with dimensionality equal to the sum of all input signals' dimensionalities.

[10] More precisely they are *Simulink Masked Sub-systems* Blocks.

[11] The available relations, during the writing of this document, are: $<, >, \leq, \geq, \neq, =$.

S-function Block This block's output is determined by the underlying binary file, which, given parameters and input signals, computes the block's output. In our monitors such file is a *MEX*-file, compiled by *C++* source code.

Suppose to have a monitor for the formula ϕ: we will show how to use the first four blocks (*Sum*, *Gain*, *Rel. Operator* and *MUX* blocks) to emulate *STL* predicates, thus converting real-valued signals into an n-dimensional boolean signal. Such signal will then be passed to an S-function block, implementing the algorithm described in Section 2.3.2, which will determine ϕ's satisfaction. We can consider the monitoring process as composed by two steps:

1. Real-valued signals to n-dimension boolean signal conversion.
2. Execution of the $MITL_{[a,b]}$ monitoring algorithm through an S-function block.

Those steps are executed for each *Simulink* major time-step. Indeed, At each time-step it is computed satisfaction for a portion of the interested simulation trace (the reason is simple *Simulink* may not have simulated the entire trace). This is possible due to what was said in claim 2.1. This kind of behaviour allows us to optimize memory usage (at the expense of time[12] however).

4.2 Real-valued Signals Conversion

In Section 2.3.3 we described how the satisfaction of an *STL* formula is reduced to the satisfaction of an $MITL_{[a,b]}$ one. Such feat required a "conversion" from an m-dimensional real-valued signal to an n-dimensional boolean one. We present here how to correctly implement this conversion , using *Simulink*.

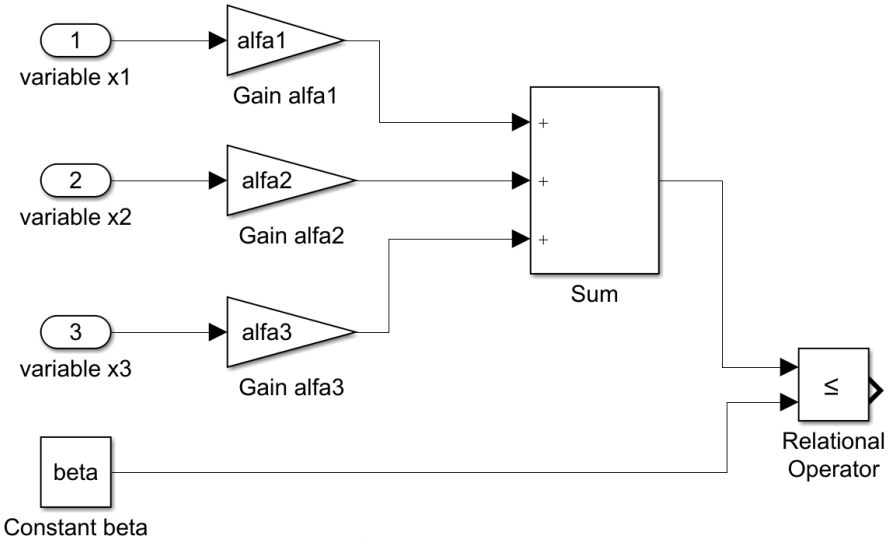
Observation:
Strictly speaking, we do not have an m-dimension real-valued signal, but a sequence of m real-valued one-dimensional ones (obtained from the monitors' input ports). This is not a problem, since we can simply consider this sequence to be an m-dimensional signal.

Let us consider a monitor which tests an *STL* formula ϕ, containing the predicates $\rho_1 \dots \rho_n$. We need to design a system able to produce, starting from a sequence of input signals $x_1 \dots x_m$, an n-dimensional boolean signal $s = (s_1 \dots s_n)$ such that:

$$s_i(t) = \text{truth value of } \rho_i \text{ at simulation time } t \tag{4.1}$$

[12] That is because we need to invoke the monitoring algorithm a lot of times with small inputs, instead of calling it one time with the complete trace

Figure 14: Block Diagram emulating predicate: $\sum_{i=0}^{3} \alpha_i x_i \leq \beta$.



We will show that such a system is not hard to build. We firstly describe how to design a block diagram emulating a predicate and then discuss how to generate the overall diagram.

Boolean Constants

Suppose to have: $\rho_i = \textit{true}$ or $\rho_i = \textit{false}$, we only need to use a constant block, with correct parameter (0 for *false*, 1 for *true*) and *data type* boolean.

Predicates

We already saw the underlying dynamic systems of *Sum* and *Gain* blocks (equations (2.1) and (2.2)). We express now the (intuitive) system function behind a *Relational Operator*[13]:

$$y(t) = \begin{cases} 0 & u_1(t) \ \mathcal{R} \ u_2(t) \\ 1 & \text{otherwise} \end{cases}$$

Let us consider the following:

$$\rho_i = \sum_{i=0}^{m} \alpha_i \cdot x_i \ \mathcal{R} \ \beta$$

We have than, because of how the blocks' dynamic system function are defined, that the output signal of a system like one in Figure 14 (in this particular case we have $n = 3$ and $\mathcal{R}$ equal to $\leq$), is equivalent to the signal described in (4.1).

[13]We are considering $\mathcal{R}$ as block parameter.

Making of the Complete Signal

Having described how to build signals able to emulate *STL* predicates we now can show how to build the complete "converted" signal (look at Figure 13 for an example). We need to:

- for each predicate (or boolean constant) build the respective subsystem;
- add a *MUX* block, with as input all the above predicates' output signals.

The *MUX* block's output signal is the desired n-dimensional boolean signal. As such our monitor will use it, as input trace, for determining satisfaction of ϕ (to be precise, for an $MITL_{[a,b]}$ formula equivalent to our *STL* ϕ).

4.3 Formula Satisfaction Computation

We previously described how to produce a multi-dimensional boolean signal, starting from real-valued signals and an *STL* formula. Such signal represents the truth values assumed by the formula's predicates during a simulation. We need now to describe how to use said signal to determine the formula satisfaction.

Note:
In this section we will assume to be using an $MITL_{[a,b]}$ formula, instead of an *STL* one. However we consider such formula to be obtained from the original *STL* one, substituting each predicate with a proposition. We will also assume that, at the index of such proposition, the input signal coordinate will be the original predicate's truth values.

The interested formula satisfaction is checked through an *S-function block*. Such block takes two inputs:

Boolean Signal The boolean signal obtained with the aforementioned process is passed, through an input port, to the S-function.

Tree of *MATLAB* structures This tree structure must represents the syntax tree of the formula under verification. Strictly speaking this is not an input, since this is passed as a parameter to the block. Nevertheless it is extremely crucial for the S-function execution, since it tells the formula's structure to the s-function.

Regarding the S-function output signal we have that:

1. It is of *data type* boolean;
2. It indicates our formula satisfaction (0 the formula is satisfied, 1 the formula is not satisfied).

4.3.1 Monitor Library

We stated that our monitors contain a *MEX S-function block*. This means that a *MEX binary file*, implementing a certain set of procedures, must be present inside the *MATLAB* path. Such file is compiled, together with a variety of custom made *C++* classes[14], during `OnMonGen` execution. This collection of *C++* files and headers is called *Monitor Library* (Figure 15).

Class Summary We give now a brief description of the main classes inside *Monitor Library*:

monitor_sfunction This is not properly a *C++* class, is instead a *C++* file, implementing the above cited procedures. It can be seen as the access point of the library to *Simulink*. It contains a series of callbacks invoked by *Simulink* during simulation phase.

Interval Class representing a right-open and left-closed interval (i.e. an interval like: $[a, b)$). This class represents only non-empty intervals.

Signal Class representing a boolean signal s. It does so by maintaining a list of (right-open, left-closed) intervals, indicating the intervals where such signal is equal to 1 ($\mathcal{I}_s^+$). The class maintains also two values, $first$ and $last$, used to delineate the signal's domain ($\mathrm{dom}(s) = [first, last)$).

ValidatorNode The goal of this interface is the determination of a formula's satisfaction for adjacent time intervals. This is achieved by organizing instances of this interface in a tree like fashion, mimicking the formula under verification syntax tree (Figure 16). A sequence of inductive (on the tree structure) methods are then called on the tree's root. Indeed, such methods are used to compute a signal representing the formula's satisfaction truth values with respect to time.

Monitor Class used to check satisfaction of a certain $MITL_{[a,b]}$ formula ϕ. It does so by creating a tree of *ValidatorNode*, reflecting ϕ's syntax tree. It then update such tree until it is able to determine the formula's satisfaction for the first simulation instant.

Interface *ValidatorNode* Description It is quite evident that the library's core element is the interface *ValidatorNode*. For this reason we want to give a more in detail

[14]Specifically designed for $MITL_{[a,b]}$ formulae's satisfaction checking, and used during the s-function execution.

Figure 15: Monitor Library Class Diagram

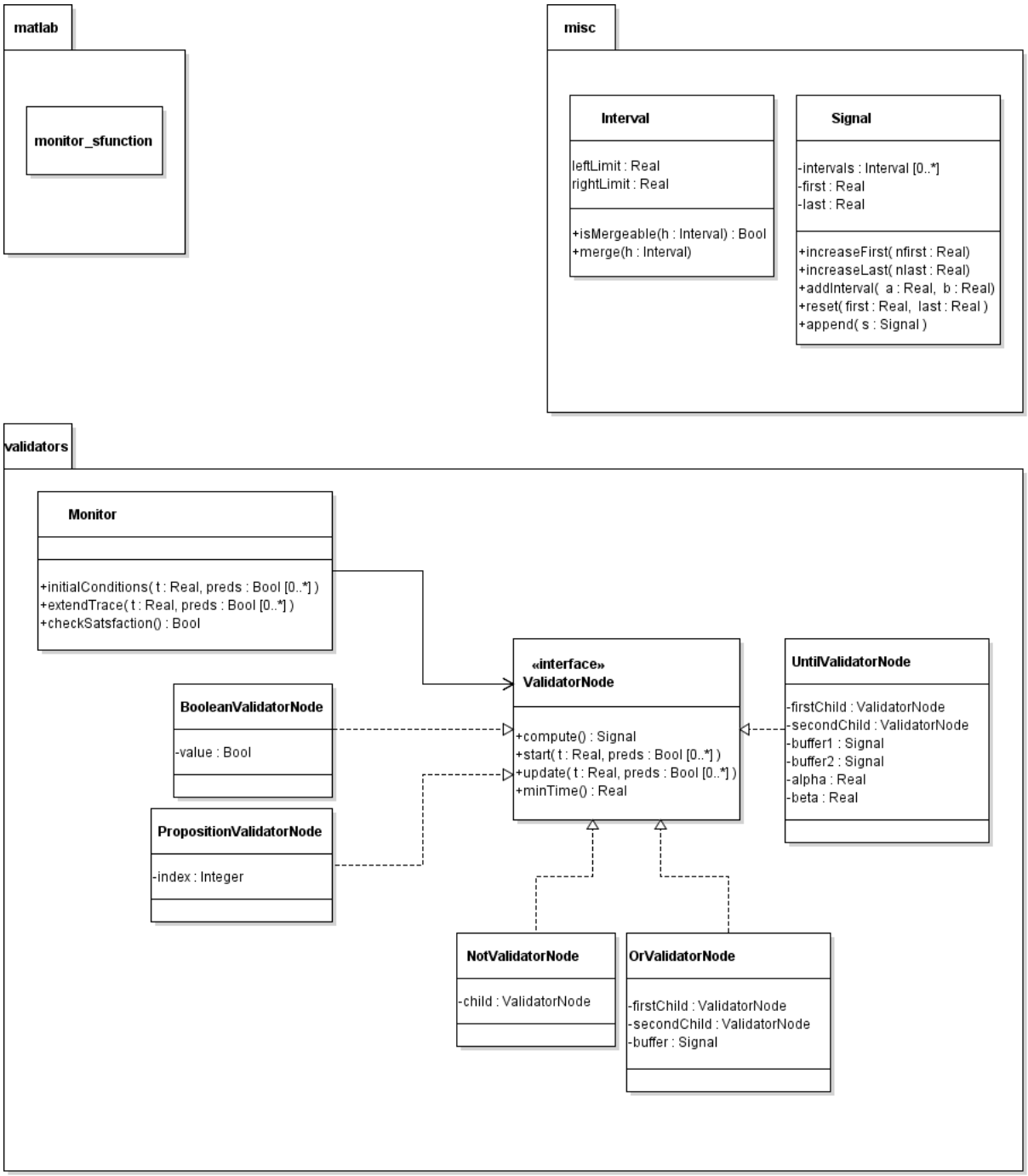


Figure 16: Object Diagram of a typical tree of *ValidatorNode* instances.

In this particular case, we have that these instances can be used to check satisfaction of the formula: $true\ \mathcal{U}_{[0,3.2]}\ (P_0 \vee P_1)$

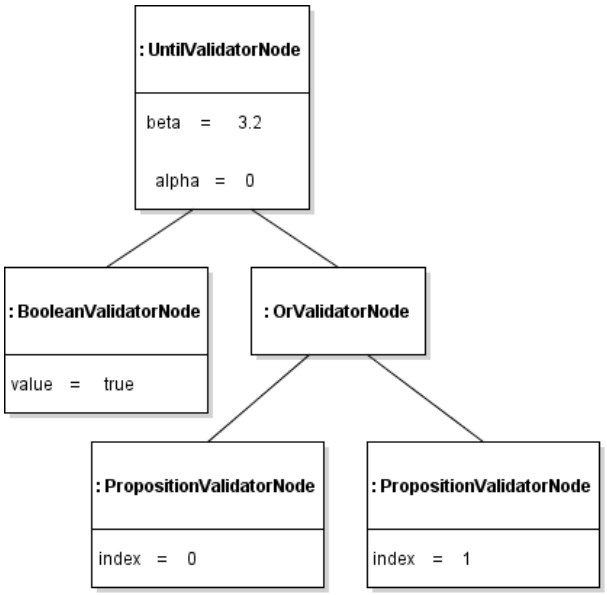
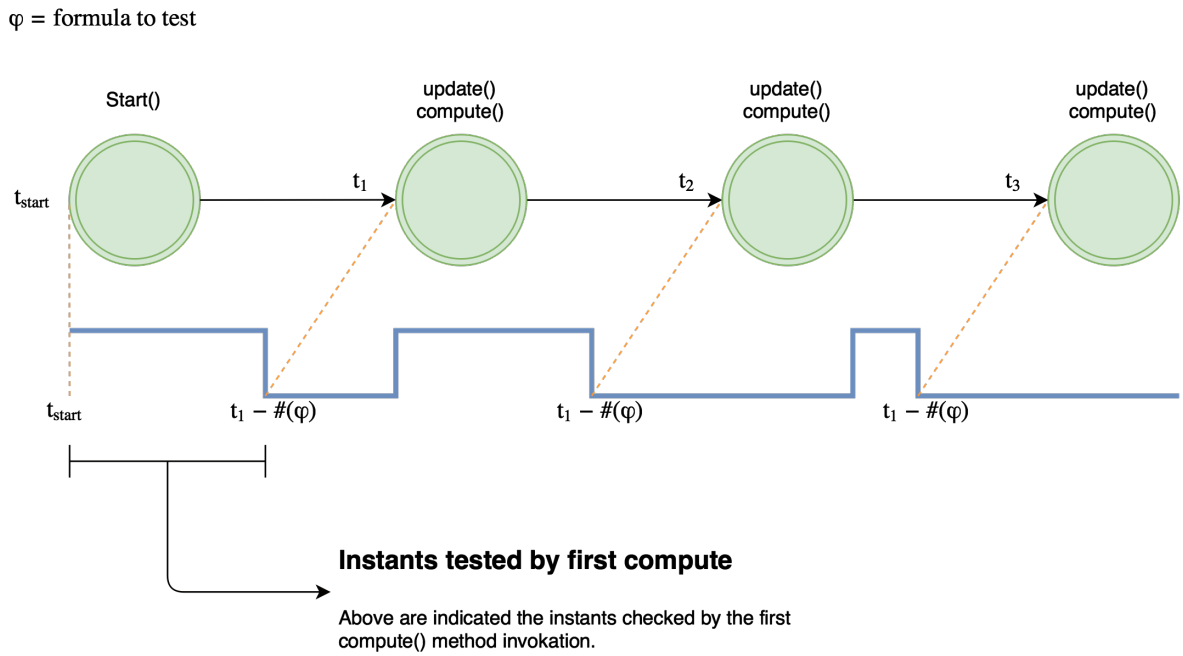


Figure 17: Sequence of method calls typically invoked by a *ValidatorNode* instance.



description of such interface. We describe which methods would be typically invoked during our monitor execution (in the model simulation).
Suppose to have obj, instance of *ValidatorNode*, whose sub-tree represents an $MITL_{[a,b]}$ formula ϕ. A sequence of methods calls like:

1. $obj.\texttt{start}(t_0, p_0)$
2. $obj.\texttt{update}(t_1, p_1)$
3. $o_1 = obj.\texttt{compute}()$

   ...

$(2k-1)$. $obj.\texttt{update}(t_k, p_k)$

$2k$. $o_k = obj.\texttt{compute}()$

$$(t_i \in \mathbb{R} \text{ and } p_i \in \mathbb{B}^n)$$

Produces a sequence of boolean signals $\{o_i\}_{i=1..k}$ (each returned by an $obj.\texttt{compute}()$ call), which indicates the formula satisfaction for successive adjacent intervals (each "delayed" by $\#(\phi)$). Formally we say that for all o_i it holds that:

$$\begin{aligned} \text{dom}(o_i) &= [\max(t_o, t_{i-1} - \#(\phi)), t_i - \#(\phi)) \\ \forall t \in \text{dom}(o_i) : (s,t) \models \phi &\iff o_i(t) = 1 \end{aligned}$$

With $s = (s_1 \dots s_n)$ a signal obtained interpolating the values passed as input to the `start` and `update` calls. A formal definition of s is the following:

$$\forall j = 1 \dots n : s_j([t_{i-1}, t_i)) = 1 \iff \pi_j(p_{i-1}) = 1$$

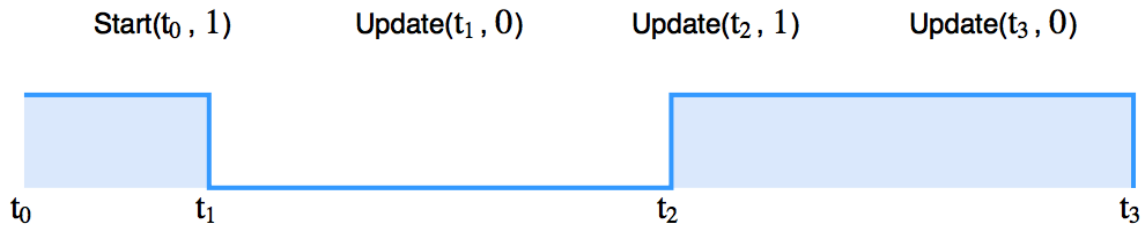
Observation:
The signal s can be seen as a piecewise constant (left-endpoint) interpolation of the points:

$$(t_0, p_0), (t_1, p_1) \dots (t_k, p_k)$$

A visual example of such a signal (with $n = 1$) can be seen in Figure 18.

The computation of the signals o_i is done using the algorithm described in Section 2.3.2. However because signals are "shifted" some signal buffers are necessary, used to "delay" the different signals in order to safely compute the formula satisfaction (remember what was stated in Section 2.3.1).

Figure 18: Example of signal reconstructed from `update` calls.



Methods Description Let ϕ be the formula represented by a tree of *ValidatorNode*. Then we give now a brief explanation of the methods' behaviour if called on the root of such tree:

`minTime`**:** This method returns the value of $\#(\phi)$.

`start`**:** This method starts the monitoring process, it is used to initialize buffers and fields of the *ValidatorNode* instances inside the tree.

`compute`**:** This method compute ϕ satisfaction signal for instants inside the interval[15]: $[\max(t_0, t_{i-1} - \#(\phi)), t_i - \#(\phi))$.

`update`**:** This method updates the *ValidatorNode* instances' state, so that the next call at compute will work as expected.

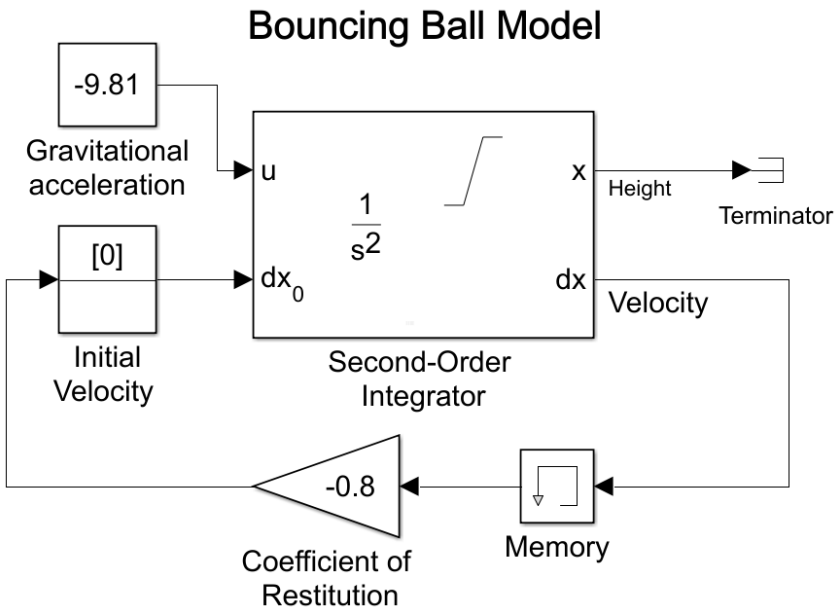
Monitor output Having briefly described *Monitor Library* we can explain how the S-function determine its output. The S-functions, during execution, maintains an instance of *Monitor* and update it using its method, such object is going to delegate most of its method to a *ValidatorNode* instance (that we will call val), representing the formula under verification. The S-function updates val at each major time-step, invoking `update`(t, p) with the current simulation time (t) and the predicates' current values (i.e. value currently assumed by the n-dimensional boolean input signal). For each `update` call a `compute` invocation follows, determining the formula's satisfaction for successive time intervals, as described before. Note that we need to only check satisfaction for the first simulation instant, which means that we just need to wait until the time-step where the output interval produced by a `compute`() has a non-empty domain.

Note:

The time we need to wait is $\#(\phi)$, for all the precedent simulation time the monitor simply output 0.

[15]We are assuming that the last two call at `update` had t_{i-1} and t_i as first arguments.

Figure 19: Bouncing ball model



5 Formulae Examples

We are going to show, as an example of how an user could use our property monitors, some properties defined over a *Simulink* model. Such model is a representation of a bouncing ball. Of the these properties, we are going to show for what instants satisfaction should be expected, specifically, we will determine the properties' satisfaction for the first simulation instant (the one checked by our monitors).

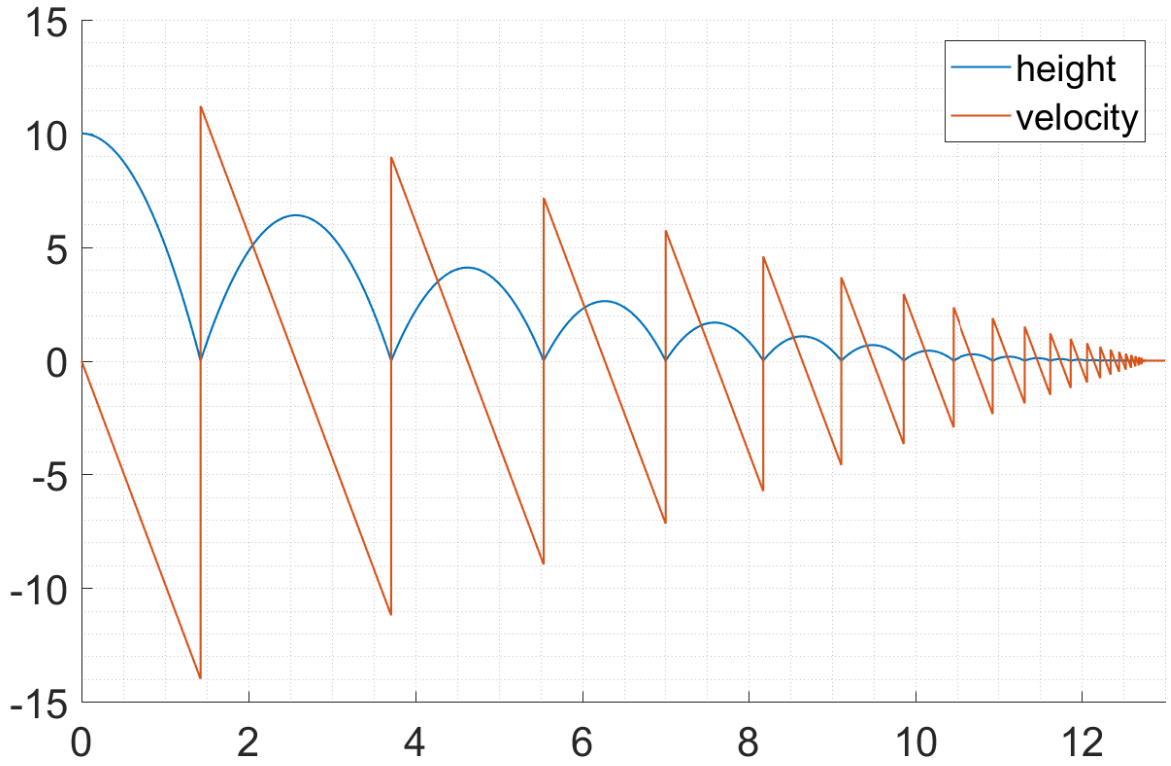
5.1 Bouncing ball Example

Let us consider a simple model representing a bouncing ball (Figure 19). In this model we are interested in two variables: *velocity* (indicated inside the properties with name v) of the ball and its *height* (indicated with h). The height's unit of measurement is in meter, while velocity is represented in meter per second. In Figure 20 we can see how the ball's height and velocity change during model simulation. The ball's initial velocity is 0 (as we can se from the graph), while its initial height is 10 meters. Simulation starts at 0 and end at the instant 25.

5.1.1 Will stop

We called this property *Will Stop*, it's used to verify that in the future (after 12 and before 13 seconds), for at least a certain amount of time, the ball's velocity is equal to 0. In other words, after some amount of time, our ball must stop moving, hence the

Figure 20: Bouncing ball simulation plot

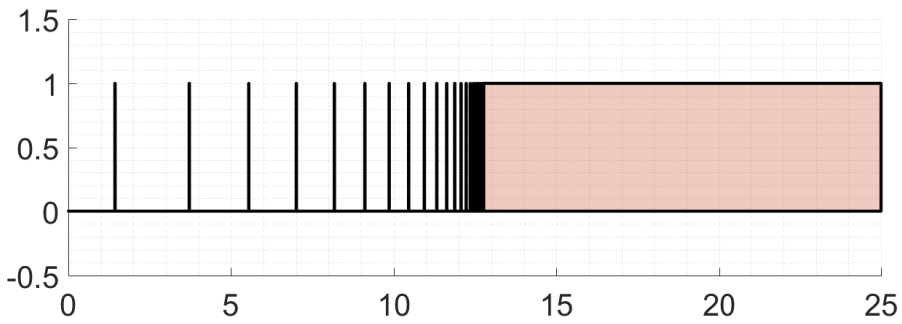
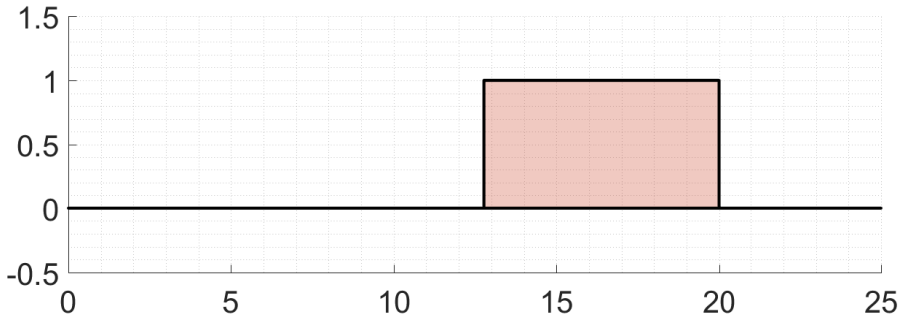
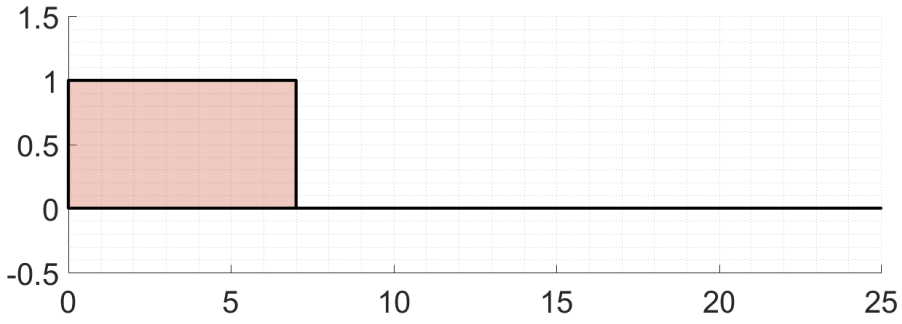


property's name. An *STL* formula which describe such a property is the following:

$$\Diamond_{[12,13]} \Box_{[0,5]} v = 0$$

Observation:
Note that $\#(\Diamond_{[12,13]} \Box_{[0,5]} v = 0) = 18$, so our simulation interval, $[0, 25]$, is sufficiently long for our property monitor to determine the formula's satisfaction.

Table 2: *Will stop* sub-formulae evaluation

$v = 0$	
$\Box_{[0,5]} v = 0$	
$\Diamond_{[0,13]} \Box_{[0,5]} v = 0$	

We can see by looking at Figure 20 that velocity around simulation time 12.7 (which we will call β) assumes value 0, and maintains such value until the end of simulation,

at 25. So, the first part of the formula ($\Box_{[0,5]}\, v = 0$) its true from β until 22 seconds. For this reason we have that the complete formula is true (for $t = 0$), since it exists a $t' \in [t + 12, t + 13]$ such that $\Box_{[0,5]}\, v = 0$ is true, such t' is exactly β. This means that the bouncing ball model satisfies the property *Will Stop*.

5.1.2 Two bounces

We define now a property that will be true if the ball bounces two times over a constant threshold (we choose it to be 4). Both the bounces must be in range of 1.5 seconds from each other. Also the ball, during its first bounce, must not stay above the threshold for more than 1.5 seconds. The formula describing such property is the following:

$$\phi = \Diamond_{[0,1.5]}(h \geq 4 \wedge \Diamond_{[0,1.5]}(h < 4 \wedge \Diamond_{[0,1.5]}\, h \geq 4)$$

For simplicity's sake we describe ϕ through the following sub-formulae:

$$\begin{aligned}
\phi_1 &= \Diamond_{[0,1.5]}\, h \geq 4 \\
\phi_2 &= \Diamond_{[0,1.5]}(h < 4 \wedge \phi_1) \\
\phi_3 &= \Diamond_{[0,1.5]}(h \geq 4 \wedge \phi_2)
\end{aligned}$$

$$(\phi = \phi_3)$$

Note:
More generally, we can describe a sequence of n bounce using this recursive formula:

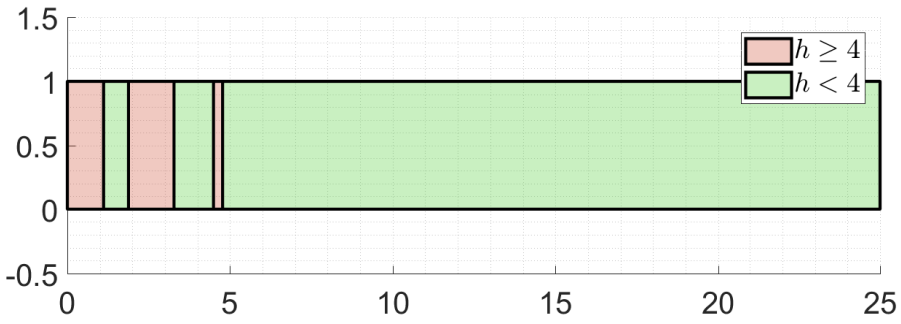
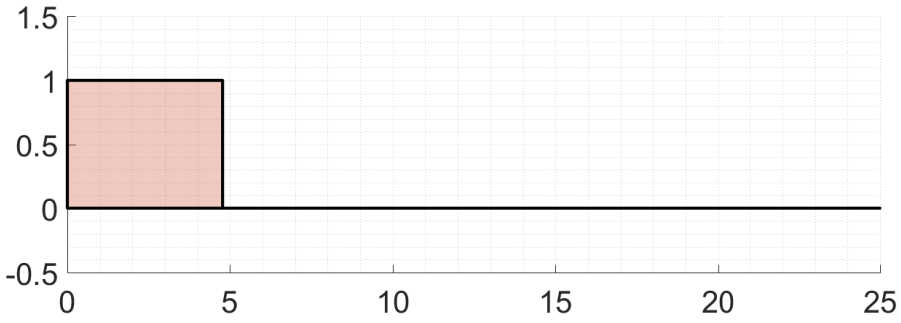
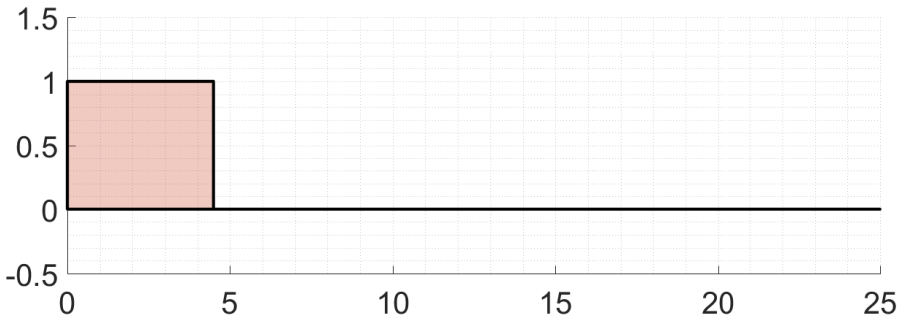
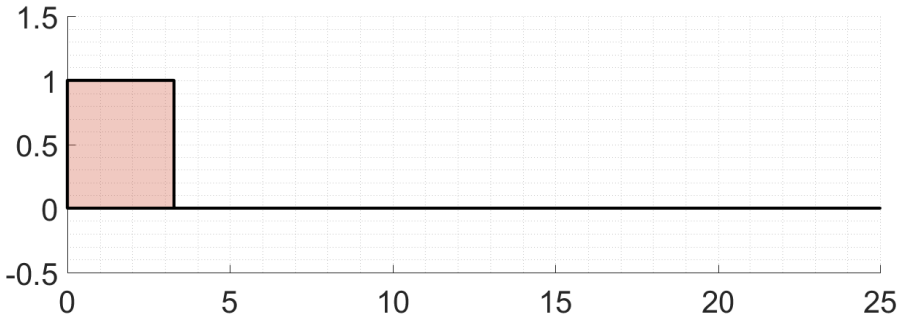
$$\begin{aligned}
\phi_1 &= \Diamond_{[0,1.5]}\, h \geq 4 \\
\phi_i &= \Diamond_{[0,1.5]}(h \geq 4 \wedge \Diamond_{[0,1.5]}(h < 4 \wedge \phi_{i-1})
\end{aligned}$$

Where ϕ_i indicates a formula which is true if at least i bounces exceed the threshold 4.

We can see from Figure 20 the bounces where height is greater than 4 are three, so we can expect the property to holds. The Table 3 shows when each of the above sub-formulae holds. We Indeed can see that, for 0, ϕ is satisfied.

Observation:
Note that $\#(\phi) = 6$, so the simulation interval is long enough for our property monitor to determine ϕ's satisfaction.

Table 3: *Two bounces* sub-formulae evaluation

$h \geq 4$, $h < 4$	
$\Diamond_{[0,1.5]}\, h \geq 4$	
$\Diamond_{[0,1.5]}(h < 4 \wedge \phi_1)$	
$\Diamond_{[0,1.5]}(h \geq 4 \wedge \phi_2)$	

5.1.3 Until property

We describe now an arbitrary property, used to give an example of a non-trivial formula containing an until operator. We will see that such property holds, regarding the simulation shown in Figure 20. The formula is defined as follows:

$$\phi = (\Diamond_{[0,1.5]}\, h \geq 4)\ \mathcal{U}_{[0,3.5]}\ (\Diamond_{[0,10]}\, \Box_{[0,1]}\, v = 0)$$

We divide such formula in different sub-formulae:

$$\phi_1 = \Diamond_{[0,1.5]}\, h \geq 4$$
$$\phi_2 = \Box_{[0,1]}\, h = 0$$
$$\phi_3 = \Diamond_{[0,10]}\, \phi_2$$
$$\phi_4 = \phi_1\ \mathcal{U}_{[0,3.5]}\ \phi_3$$

$$(\phi_4 = \phi)$$

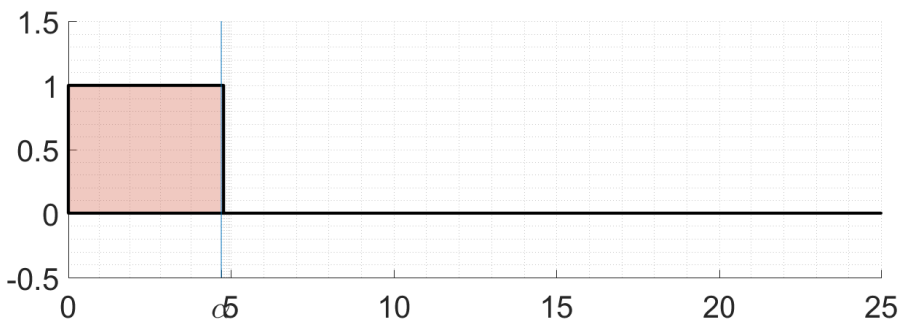
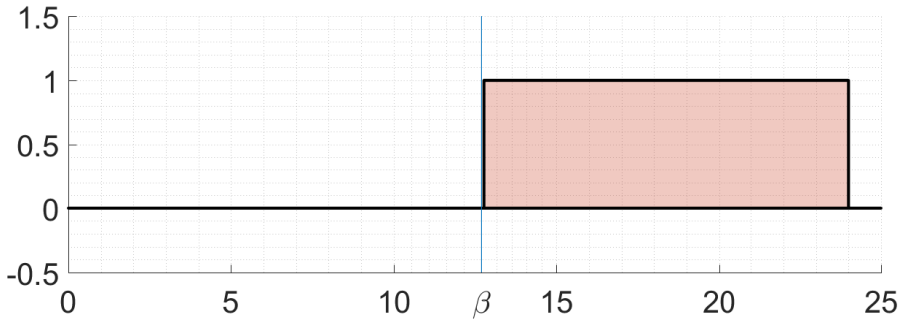
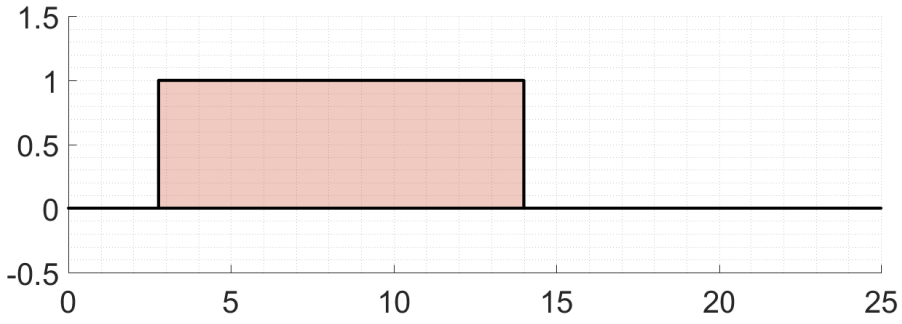
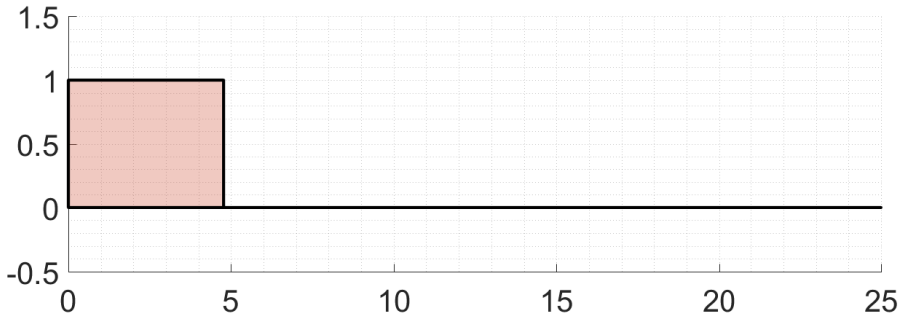
Observation:
Note that $\#(\phi) = 14.5$, so the simulation interval is sufficiently long for our property monitor to determine the formula's satisfaction.

The formula $\phi_1 = \Diamond_{[0,1.5]}\, h \geq 4$ holds if, within a range of 1.5 seconds, the ball's height reaches a value greater than or equal to 4. This property holds until the last instant, during the third ball's bounce, where the height is greater than 4. We call such instant α ($\alpha \sim 4.7$).

On the other hand, ϕ_2 and ϕ_3 can be thought as properties similar to *Will stop*. Let us call β the simulation time where velocity become constantly equal to 0 ($\beta \sim 12.7$). Then it can be seen that ϕ_2 is true for each instant greater than β (Table 4). Such statement yields, since the globally part of ϕ_2 is true from β to 24. This means that for each instant greater $\beta - 10$ (and lower than $25 - \#(\phi_3)$) ϕ_3 yields.

Combining these two signals we can obtain the signal for ϕ_4, which is true for the instants $[0, \alpha)$. This means that the formula ϕ holds during simulation.

Table 4: *True Property* sub-formulae satisfaction

$\Diamond_{[0,1.5]}\, h \geq 4$	
$\Box_{[0,1]}\, h = 0$	
$\Diamond_{[0,10]}\, \Box_{[0,1]}\, h = 0$	
$\phi_1\, \mathcal{U}_{[0,5]}\, \phi_2$	

6 Conclusions and Future Work

In this document we presented `OnMonGen`, a system which produces as output a *Simulink* library of sub-systems. Such systems are able to monitor the satisfaction of a specified temporal formula, during a simulation. This kind of monitors can be useful in automatic verification of *Simulink* models, if approaches like *Run-Time Verification* are used. Our goal was, partially, to give an on-line[16] implementation of the monitor described in [3]. In its current state, `OnMonGen` can be expanded and improved in several ways. We will describe here a list of the main improvements that can effected to the system:

Redundant operators implementation In its current state, `OnMonGen` implements only the operators: $\neg$, $\vee$ and $\mathcal{U}_{[\alpha,\beta]}$, while the other are emulated building equivalent formulae. This behaviour guarantees an easy maintenance, readability and testing of the code, but cause a more inefficient way of checking formulae, since it requires more complex syntax trees and uses algorithms more complicated than required. If we want to achieve the generation of more performing and efficient monitors, then implementing the redundant operators $\wedge$, $\Box_{[\alpha,\beta]}$ and $\Diamond_{[\alpha,\beta]}$ could be an easy and valid solution. It should be also noted that determining the satisfaction signal of a formula of type $\Diamond_{[\alpha,\beta]}\ \phi$ or $\Box_{[\alpha,\beta]}\ \phi$ its easer than a formula of type $\phi_1\ \mathcal{U}_{[\alpha,\beta]}\ \phi_1$.

Enable simulation state saving Our current monitors do not allow the saving of a simulation state: usually it is possible, during a simulation, to save the state of the model in an object of type *SimState*. Using the words written in the *Simulink®* *User's Guide*[1] we have that:

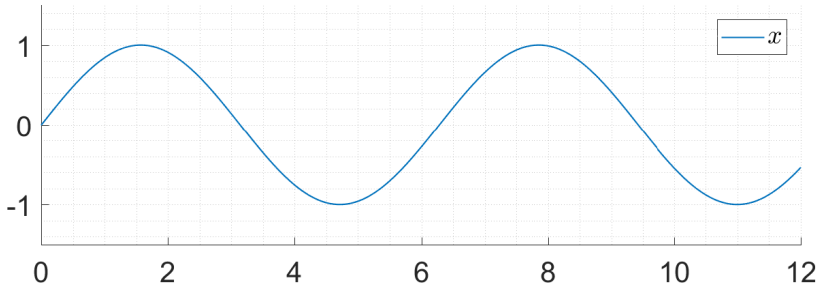
> *A SimState is the snapshot of the state of a model at a specific time during simulation.*

Unfortunately such a feature requires every *S-function* in the model to implement a specific procedure (called *mdlGetSimState* in the *C++* source code), which for reason of time was not realized. A simple but nice improvement to `OnMonGen` would be the implementation of said function. This could be decisive in case of big systems that requires to run for long periods of time and can not be easily re-simulated.

Removing virtual functions As we already saw *Monitor Library* contains an interface: this means that in the *MEX*-file source code are present some *C++* *virtual* functions. A virtual function can be more expensive (regarding execution time) than

[16] In this contest with *on-line* we refer to something directly connected to a model, differently from, for example , off-line monitors, which work on simulation traces after the model's simulation.

Figure 21: Trace μ plotting



non virtual functions, this is caused by the dynamic (or lazy) binding used to determine the function to be invoked during the program execution. A way to enhance the monitor performance could consist of a refactoring of the *mex-file* source code, removing those virtual functions. Such a result can not easily achieved, but if done correctly could boost the monitors' performance.

Not delayed monitors The monitors generated by the `OnMonGen` delay judgement of a simulation trace until the first instant where they can be sure of correctly determining a formula satisfaction regarding the given trace. To be precise this instants is the simulation time of the time-step following the instant $t_s + \#(\phi)$, where t_s is the start time of the simulation. An improvement then would be the ability for the monitor to output a result exactly when it is possible to correctly determine the satisfaction of the formula. This can be done, but requires some sophistications.

An example could be the formula:

$$\phi = \Box_{[0,10]}\, x \geq 0$$

Consider the case of a simulation trace μ, containing a single variable x which represents $sin(t)$ (as can be seen in Figure 21). We have that at the instant $\alpha \sim 3.1416$ it possible to determine that $\mu, t_s \not\models \phi$ since x, for α, is less than zero. Unfortunately our monitor doesn't compute the actual satisfaction until the simulation time $\#(\phi)$.

References

[1] Simulink User's Guide, march 2017. *Release 2017a*.

[2] Rajeev Alur, Tomás Feder, and Thomas A. Henzinger. The benefits of relaxing punctuality. *J. ACM*, 43(1):116–146, 1996.

[3] Oded Maler and Dejan Nickovic. Monitoring temporal properties of continuous signals. In *FORMATS/FTRTFT*, volume 3253 of *Lecture Notes in Computer Science*, pages 152–166. Springer, 2004.