Signal Temporal Logic Monitoring and Online Robustness Approximation

Research Project 2 May 17, 2021

Prof: Guillermo Alberto Perez Laurens Van Damme

Introduction

Verification is important in any kind of system to make sure it does what it is supposed to do. To do verification, a specification/requirements engineer needs to specify the properties of interest that need to be verified. These properties can then be checked in an automated manner with the help of a verification tool. An automated process is often much more reliable than manual inspection and verification. Formal methods became powerful mathematical tools used to do specification and verification. A big advantage is that you can go beyond simple properties like safety and stability. With these formal methods it is possible to check much richer specifications.

Temporal Logic (TL) is such a formal method, more specifically a language for specifying acceptable behaviors of reactive systems. It has already been used to define rich time-dependent constraints for control systems in a wide variety of applications ranging from biological networks to multi-agent robotics [1, 2, 3]. A temporal logic-based specification formalism [4] has even been adopted by hardware industry as specification language. There are two categories of TL techniques: automata based are computational heavy and perform *formal verification*, e.g. automatic model checking, and optimization based which became more popular in recent years and performs *monitoring*. Monitoring provides much more flexibility since it is not required to have a model of the system, only a process that generates traces of observations is necessary. This way it can also be used on systems that are viewed as black boxes. An analogy can be made with testing in the software context under the name *run-time verification*, i.e. monitoring by tests on individual simulation traces. When a test passes, it continues to the next one, else it is abandoned and the software thus failed the tests or the required properties. The formula is in this case in the shape of a small test program.

Signal Temporal Logic (STL) is a TL formalism for specifying properties of dense-time quantitative signals. Many real-life applications are working with continuous data and in those situations a yes or no as answer to a property that has to be verified is not enough. The yes/no answer is rather only satisfying in a discrete context. With that reason we focused on the optimization based approach. With STL it is possible to leverage the definition of quantitative semantics to compute the robustness estimate. This results in an optimization problem since the robustness has to be as high as possible, because the higher, the "more satisfied" the property is.

The main contribution of this article is the presentation of an easy to use tool written in Python for STL monitoring. Both validation with Boolean semantics and Robustness estimate computation with quantitative semantics are supported. The code can be found on github: `https://github.com/LaurensVanDamme/STL-monitoring`

The rest of this article is organized as follows. Section 1 reintroduces STL in a brief fashion and also indicates how the two semantics are linked together. The tool itself is explained in Sections 2 & 3. Here we go deeper into some implementation choices that were made and also discuss the used algorithms to perform all the STL operations. Finally Section 4 adds some experiments that were performed with the tool.

1 Signal Temporal Logic

This section reintroduces the STL framework in brief. We adopted most of the conventions from *Efficient Robust Monitoring for STL* [5].

A *signal* is a function $\mathbb{T} \to \mathbb{D}$, where the time domain $\mathbb{T}$ is the set $\mathbb{R}^+$. For $\mathbb{D}$ there are two possibilities. $\mathbb{D} = \mathbb{B}$, these signals are called *Boolean signals* because $\mathbb{B} := \{0, 1\}$, or in words `False` and `True`. $\mathbb{D} = \mathbb{R}$, these signals are called *real-valued signals*. An *execution trace* w is a set of real-valued signals $\{s_1^w, ..., s_n^w\}$, which can be "Booleanized" through a set of *threshold predicates* or *Boolean "filters"* in the form $s_i \geq n$ with $n \in \mathbb{R}$. The basic formulae of STL are defined by the grammar:

$$\varphi := \texttt{True} \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \mathbf{U}_I \varphi$$

where $I = [a, b]$ is a closed, non-singular interval of $\mathbb{R}^+$ with $0 \leq a < b$. The three operators present in the grammar are the basic STL operations. We can also define other usual operators as *syntactic abbreviations* or *shorthands*:

$$\varphi \vee \psi := \neg(\neg\varphi \wedge \neg\psi) \quad \varphi \to \psi := \neg(\varphi \wedge \neg\psi)$$
$$\diamond_I \varphi := \texttt{True}\mathbf{U}_I \varphi \qquad \Box_I \varphi := \neg \diamond_I \neg\varphi$$

With w being a trace of time domain $\mathbb{D}$. The formula φ is said to be defined over a time interval $dom(\varphi, w)$.

1.1 Boolean Semantics

For a trace w, the validity of an STL formula φ at a given time $t \in dom(\varphi, w)$ is set according to the following inductive definition:

$$\begin{array}{lll} w, t \models \texttt{True} & & \\ w, t \models \neg\varphi & \text{iff} & w, t \not\models \varphi \\ w, t \models \varphi \wedge \psi & \text{iff} & w, t \models \varphi \text{ and } w, t \models \psi \\ w, t \models \varphi \mathbf{U}_I \psi & \text{iff} & \exists t' \in t + I : w, t' \models \psi \text{ and } \forall t'' \in [t, t'] : w, t'' \models \varphi \end{array}$$

For a given formula φ and execution trace w, we define the satisfaction signal $\chi(\varphi, w, .)$ as follows:

$$\forall t \in dom(\varphi, w) : \chi(\varphi, w, t) := \begin{cases} 1 & w, t \models \varphi \\ 0 & otherwise \end{cases}$$

Computing the satisfaction signal $\chi(\varphi, w, .)$ of a formula φ is done by computing the satisfaction signal $\chi(\psi, w, .)$ for each subformula ψ of φ. This is a recursive procedure on the structure of the formula. The formula is validated if $\chi(\varphi, w, t_0) = 1$.

1.2 Quantitative Semantics

For a trace w, the robustness estimate $\rho(\varphi, w, t)$ of an STL formula φ at a given time $t \in dom(\varphi, w)$ is defined in the quantitative semantics by induction as follows:

$$\begin{array}{lcl} \rho(\texttt{True}, w, t) & = & \texttt{True} \\ \rho(\neg\varphi, w, t) & = & -\rho(\varphi, w, t) \\ \rho(\varphi \wedge \psi, w, t) & = & \min\left[\rho(\varphi, w, t),\ \rho(\psi, w, t)\right] \\ \rho(\varphi \mathbf{U}_I \psi, w, t) & = & \sup\limits_{t' \in t+I} \min\left[\rho(\psi, w, t'),\ \inf\limits_{t'' \in [t, t']} \rho(\varphi, w, t'')\right] \end{array}$$

Note that if the above inductive rules are applied to χ with

$$\chi(s_i \geq 0, w, t) := \begin{cases} 1 & s_i^w(t) \geq 0 \\ 0 & otherwise \end{cases}$$

it comes down to using Boolean signals again and an equivalent characterisation of χ is obtained. In quantitative semantics this is not the case, because s_i will evaluate to a real value representing the distance to satisfaction or violation. Thus it will not evaluate to `True` or `False`. This eventually propagates in the use of operations on $\mathbb{R}$ in the formula and not their logical counterparts that are used on $\mathbb{B}$.

2 Implementation

In this section, the details of the implementation will be discussed. The algorithms used to check the satisfaction of a STL formula are discussed later on in Section 3. Python was the chosen language because it is easy to understand and supports a lot of useful libraries. To parse the given STL formula and check for syntax errors, the tool of choice was ANTLR [6]. It is written in Java, but supports Python code generation. ANTLR generates a tree from a given STL formula and this tree is then used to create the self-made STL Tree. Wiht the help of the STL tree, it is possible to check if the given signals are validated by the given STL formula or compute their robustness estimate in a Boolean or quantitative semantic respectively. The rest of this section is structured as follows: First, the methods of providing the input signals or execution trace are discussed in Section 2.1, Section 2.2 describes the grammar used by the ANTLR tool and the STL Tree is explained in Section 2.3. Please note that when the validation by an STL formula is mentioned, it is also possible to compute the robustness estimate as it all just depends on the semantic context.

2.1 Signals as input

The signals that need to be validated by the given STL formula should be provided in a `CSV` file. The file consists of one or more comma-separated columns which represent one signal or its timestamps. The first row is the header row, where the names of the signals and its timestamps are placed. The names of the signals can be a combination of lowercase or uppercase letters and numbers. The name must start with a letter and can not include any spaces. All the rows underneath represent the measurements of the signal per time step, so one row is one time step. The signals do **not** have to be of the same length, but keep in mind that if the signal is not long enough to check if it validates the formula, then the validation will not work. An example of some signals without any properties is shown in Table 1.

`Signal1`	`Signal2`	`Signal3`
0.23	2	5000
0.57	10	10000
1.00	15	5000
0.61	23	10000
0.23		5000

Table 1: A simple example of an input `CSV` file of which the content is an execution trace.

Definition 2.1. A signal s is said to be finetely piecewise-linear continuous if there exists a finite sequence $(t_i)_{i \leq n_s}$ such that:

- the definition domain of s is $[t_0, t_{n_s})$
- for all $i < n_s$, s is continuous at t_i and affine on $[t_i, t_{i+1})$

It is also possible to define the timestamps of the signals. The header of these columns should be the name of the signal ending with `_t`. If they are not provided, like in the example of Table 1, they are added automatically. The added time steps are one per value in the signal, meaning if the signal has 4 values, the time steps will be $[0, 1, 2, 3]$. Just like `Signal4` in Table 2. So the column `Signal4_t` with the time steps of `Signal4` shouldn't be added, since it would be added like that automatically. The possibility of adding the time steps yourself allows you to use decimals number as time steps and provide certain segments of signals easier. **The signals are assumed to be *finitely piecewise-linear continuous* as defined in Definition 2.1, so you can provide the first and last time step and skip the rest in between.** The values in between time steps are then easy to calculate. Both advantages are used to define `Signal5` in the example shown in Table 2.

Signal4_t	Signal4	Signal5_t	Signal5
0	4	0	0.004
1	12	10.2	0.004
2	34	10.7	4.000
3	64	21.78	0.003

Table 2: An example of an input `CSV` file with the timestamps defined manually and some time steps that are decimal numbers.

2.2 Grammar

The grammar for the STL formulas handles the basic operators, defined in section 1, but also accepts the shorthands. The shorthands are also allowed, because these notations are often used and easier to understand. As explained in section 2.3, these shorthands will eventually be replaced by their equivalent notations consisting of the basic components. The grammar uses easy notations of all the used symbols and does not only allow the STL operations but also some basic mathematical ones. At the end of this subsection an example is provided with some explanation.

2.2.1 Symbol Representations

As the symbols $\Box$, $\Diamond$ and $\neg$ aren't that easy to type and most of the time not supported on keyboards, representations of these symbols have been made. This way writing the formula in a file will be much easier. The representations of these symbols and other supported ones are shown in Table 3.

2.2.2 Mathematical Operators

Allowing mathematical operators provides the possibility for easy manipulation of the signals before they are validated by the STL formula. Six of these operators are also the Boolean "filters" that are used in the Boolean semantic context like explained in Section 1. Since they

Meaning	Symbol	Representation	Representation Description
Until	$\mathbf{U}$	U	capital U
Always	$\Box$	[]	open and closed square bracket
Eventually	$\Diamond$	<>	open and closed angled bracket
Interval	$X_{[a,b]}$	$X\{a, b\}$	open and closed curly bracket with two numbers
Negation	$\neg$	$-$	minus
Or	$\bigvee$	V or |	capital V or pipeline
And	$\bigwedge$	$\wedge$ or &	caret symbol or ampersand
Implication	$\rightarrow$	$->$	minus and greater than

Table 3: Representations of symbols in the grammar.

are handled a bit differently, we split them in two groups: standard math operators and Boolean "filters". The grammar from Section 1 thus extends to:

$$\varphi := f \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \mathbf{U}_I \varphi \mid \varphi < \varphi \mid \varphi > \varphi \mid \varphi = \varphi \mid \varphi \leq \varphi \mid \varphi \geq \varphi$$

$$f := f + f \mid f - f \mid f * f \mid f/f \mid |f| \mid r$$

$$r \in \mathbb{R}$$

These operators should only be used before any STL operator is used. As mentioned above they are used for the manipulation of signals **before** they are validated. Before means that in the ANTLR tree, that represents the order of operations, these operators can not be present above an STL operator. Take for example the following STL formula:

$$\Box_{[1,2]} signal + 2$$

In this case it may seem the *Always* will be computed first, but the grammar will interpret that the addition has to be computed first. Because then the addition will be underneath the STL operator in the tree and is thus equal to:

$$\Box_{[1,2]} (signal + 2)$$

Standard Math Operators
The standard mathematical operators are very flexible, but they have strict placing in the STL formula. The supported ones are displayed in Table 4. They can be used with signals and numbers combined, meaning that you can add a signal to a signal, but it is also possible to add a constant number to a signal or even numbers with each other. When two signals are used, the operation takes place on the values of both signal of the same time step. So in case of an addition, the first values of both signals are added and that will become the first value of the resulting signal. If the two signals are not of the same length, the resulting signal will have the length of the shortest signal. With a constant number, it is added to every value in the signal.

Boolean "Filters"
The operators that are also used as Boolean "filters" are *comparison operators*, they are listed in Table 5. These operators always need a signal on the left side of the operator. On the right there can be both a number or a signal. A comparing operator returns a Boolean, meaning that the resulting signal will be a Boolean signal, i.e. a signal only consisting of ones and zeros. That is also why they are categorized as Boolean "filters". When two signals are used, the values of both signals of the same time step will be compared. When the signals do not have

Meaning	Symbol
Addition	$+$
Subtraction	$-$
Multiplication	$*$
Division	$/$
Absolute value	$\|.\|$

Table 4: The supported mathematical operators that are not Boolean "filters".

the same length, the resulting signal will have the length of the shortest provided signal. When comparing to a number, each value of the signal will be compared with that same number.

When using quantitative semantics, these operators can still be used. The difference is that now the STL operators do not need a Boolean signal, so in this case it is just another pre-processing step.

Meaning	Symbol
Larger than	$>$
Smaller than	$<$
Equal	$=$
Larger than or equal	$>=$
Smaller than or equal	$<=$
Unequal	$!=$

Table 5: The supported mathematical operators that are categorized as Boolean "filters".

2.2.3 Example

Here an example is shown of how a STL formula should be written to be used by the implementation. Take the following formula:

$$\square_{[300,2500]}(((10 + y) \leq 30) \wedge ((|x| > 0.5) \rightarrow \diamond_{[0,150]}\square_{[0,20]}(z \leq 0.5)))$$

This formula is made for a Boolean semantic context because all the signals are changed to a Boolean signal with Boolean "filters" before they are used in the STL operators. That transformed formula that can be interpreted by the used grammar looks like this:

```
[]{300,2500}(((10 + y) <= 30) & ((|x| > 0.5) -> <>{0,150}[]{0,20}(z<=0.5)))
```

2.3 STL Tree

ANTLR creates a tree, so why is this *STL Tree* necessary? The advantages of making your own tree is that you have full control over it. It is easy to implement (recursive) functions that you need to process the signals and it can be presented in your own preferred way(s). Also, ANTLR generates a tree based on the given grammar, meaning it has too much information that is not needed. The STL Tree will therefore become a simplified version of that the ANTLR tree, because the unnecessary nodes will be deleted and some nodes will be replaced with others.

2.3.1 Structure

While the ANTLR Tree can consist of more than sixty different types (all tokens and rules in the grammar) of nodes, the STL Tree only has twelve different types. All these nodes are the

leafs of the tree in Figure 1 that represents the structure of the STL Tree.
All nodes eventually inherit from *Node*, this class keeps all the information to keep the tree together and structured. While all other nodes represent some part of the STL formula, the *ContentNode* will always be the top of the tree. You can see it as the whole formula itself, because all other parts are linked to this node, being its children.

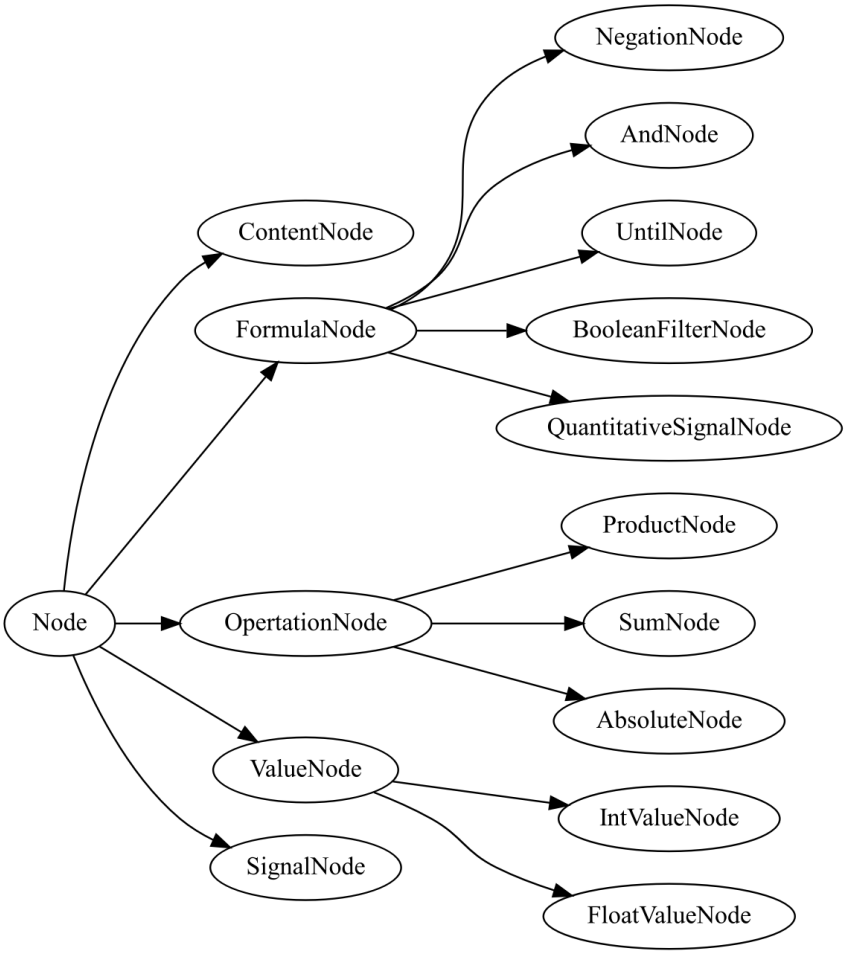


Figure 1: An overview of the structure of the STL Tree.

The *SignalNode* doesn't do anything more than reading the signal from the provided `CSV` file. Both the *ValueNodes* and *OperationNodes* are very straightforward, representing their values and operations respectively. Since addition and subtraction is the same operation but with a different sign, they are combined into one node. The same idea is valid for the product and division. The three nodes that represent the basic STL operations will be discussed in Sections 2.3.2 & 3 and the *BooleanFilterNode* has been explained above in Sections 1 & 2.2.2.

The only node left is the *QuantitativeSignalNode*, this node enables the use of signals without putting them in Boolean "filters". In a Boolean semantic context it applies the following rule:

$$b_i = \begin{cases} 0 & s_i < 0 \\ 1 & s_i \geq 0 \end{cases}$$

where s_i is the i^{th} value of the given signal s and b_i is the i^{th} value of the resulting Boolean signal b that can then be used in the STL operators using the Boolean semantics.
In a quantitative semantic, the node allows the signal to stay exactly as it is provided and

calculates the derivatives of each time step of the signal so they can be used in the computation of the robustness using an Until operator. More details on this in Section 3.2.3.

2.3.2 Shorthands

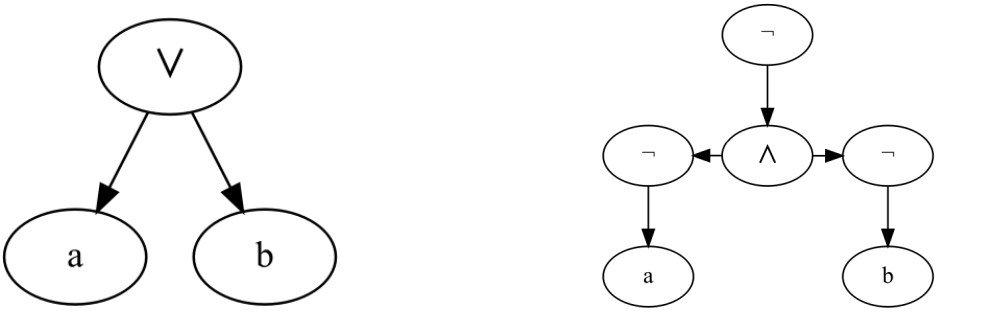
All the shorthands discussed in Section 1 are accepted by the grammar, but the tree only has three possible nodes that execute STL based functions and a bunch of nodes that represent numbers, mathematical operators, etc as explained above. These three nodes represent the basics of STL, namely the *Until Node*, *And Node* and *Negation Node*. So in the conversion of ANTLR tree to STL Tree, the shorthand nodes are replaced with their equivalent notations consisting of the three basic STL nodes.
The node with a `T` is often present in the figures that will be referred to in Table 6. This node represents an always/constant true signal and is present here for the sake of clarity. It is not included in the actual tree, but in fact a dummy signal is used in the implementation. These figures represent the steps that are taken in the tree to transform a shorthand into its equivalent notation.
There are four shorthands that are accepted and thus also have a transformation in the tree. The figures that display those transformations can be found in Table 6.

Shorthand	Figure
Or	2
Implication	3
Eventually	4
Always	5

Table 6: The supported shorthand STL operators linked with the figure that displays their transformation in the STL Tree.

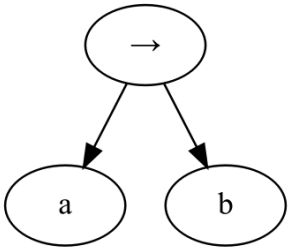
2.3.3 Double Negation

The transformation mentioned in Section 2.3.2 will often end up putting two negations right after each other. Since this is time and resource consuming for no reason, these double negations will be deleted. The moment a negation is added, there will be a check to see if the next node isn't a negation too. If this is the case, no negation node is added and the second one is deleted.

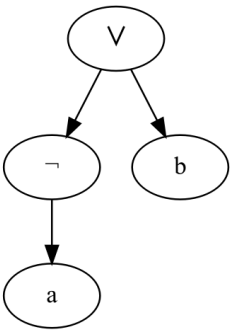


(a) Tree representation of $a \vee b$ (b) Tree representation of $\neg(\neg a \wedge \neg b)$

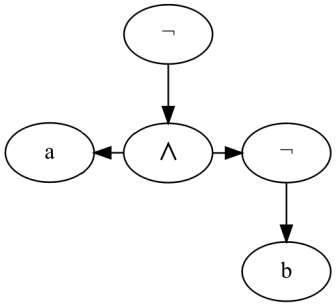
Figure 2: Steps taken in the tree to replace the *or* with its equivalent notation consisting of the basic components.



(a) Tree representation of $a \rightarrow b$

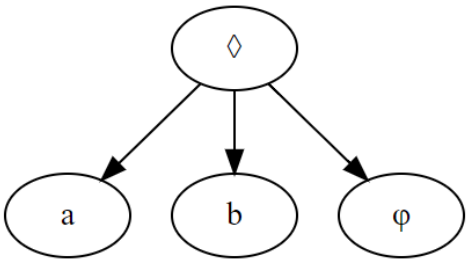


(b) Tree representation of $\neg a \vee b$

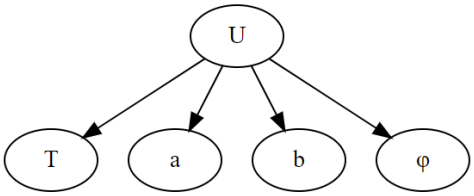


(c) Tree representation of $\neg(a \wedge \neg b)$

Figure 3: Steps taken in the tree to replace the *implication* with its equivalent notation consisting of the basic components.

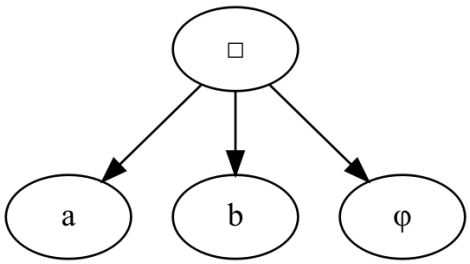


(a) Tree representation of $\Diamond_{[a,b]}\phi$

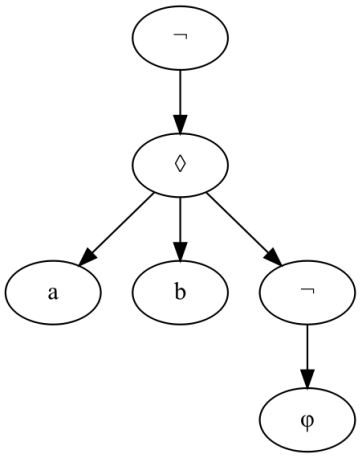


(b) Tree representation of $T\mathbf{U}_{[a,b]}\phi$

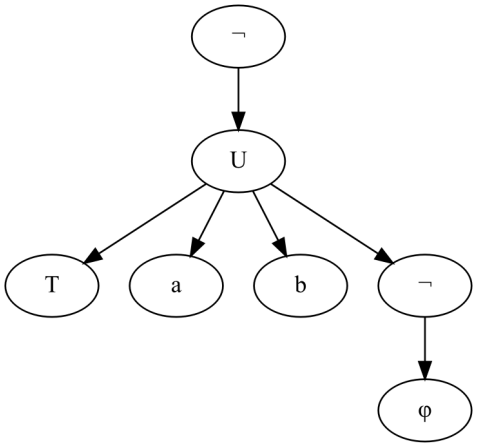
Figure 4: Steps taken in the tree to replace the *eventually* with its equivalent notation consisting of the basic components.



(a) Tree representation of $\Box_{[a,b]}\phi$



(b) Tree representation of $\neg\Diamond_{[a,b]}\neg\phi$



(c) Tree representation of $\neg(T\mathbf{U}_{[a,b]}\neg\phi)$

Figure 5: Steps taken in the tree to replace the *always* with its equivalent notation consisting of the basic components.

3 Algorithms

The algorithms that implement the behavior of the STL operators are explained in this section. It is important to notice that only three operators (negate, and, until) are discussed as the shorthands from Section 1 are transformed into their equivalent notations with the basic STL operators. These transformations are explained in Section 2.3.2. This results in only using three different algorithms, instead of one for every possible operator even if that operator is some syntactic abbreviation.
Section 1 indicates how STL formulas are used. The satisfaction of a formula can be checked by computing the satisfaction of each subformula, this is a recursive procedure on the structure of the formula. Therefore the general algorithm will parse the STL Tree (Section 2.3) bottom up. While parsing, the algorithms of the nodes are executed to compute the satisfaction of the sub-formula that the current sub-tree represents. This mimics the behavior of the recursive procedure.
Please note that all the subscripts in this section do **NOT** indicate indexes, but time steps. An example, r_t is the value of signal r at time step t and **not** the t^{th} value of the signal.

3.1 Validation

The validation of input signals with a STL formula happens in a Boolean semantic context. You know the signals are validated if after parsing the tree bottom up, the first time step of the resulting Boolean signal has the value `True`. Boolean signals only consist of zeros and ones that represent `False` and `True` respectively. The rest of this subsection will discuss the used algorithms for the different operators using the zeros and ones, instead of `False` and `True`.
Since we are working with continuous signals, we presume that the value of a defined time step is present until the next defined time step. So the y value is continued constantly to the right from every x position, i.e. the interval $[x_i, x_{i+1})$ has the value y_i.

3.1.1 Negation

A negation in Boolean semantics applies the following rule:

$$b_t = \begin{cases} 0 & , \quad s_t = 1 \\ 1 & , \quad s_t = 0 \end{cases}$$

where b is the result of the negation and s is the input. Both signals are Boolean signals. Figure 6 shows output of a negation example in Boolean semantics from the presented tool.

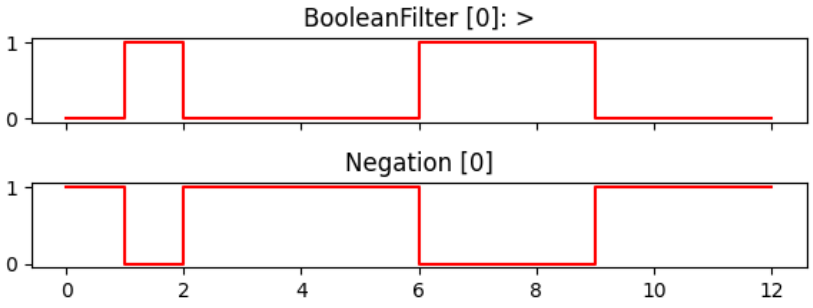


Figure 6: Output from the presented tool of a negation example in Boolean semantics. The used formula in this example is: $\neg(s > 0)$

3.1.2 And

A logical AND in Boolean semantics applies the following rule:

$$b_t = \begin{cases} 0 & , \quad s1_t = 0 \quad s2_t = 0 \\ 0 & , \quad s1_t = 0 \quad s2_t = 1 \\ 0 & , \quad s1_t = 1 \quad s2_t = 0 \\ 1 & , \quad s1_t = 1 \quad s2_t = 1 \end{cases}$$

where b is the result of the logical AND, $s1$ and $s2$ are the inputs and all signals are Boolean signals. Figure 7 shows output of a logical AND example in Boolean semantics from the presented tool.

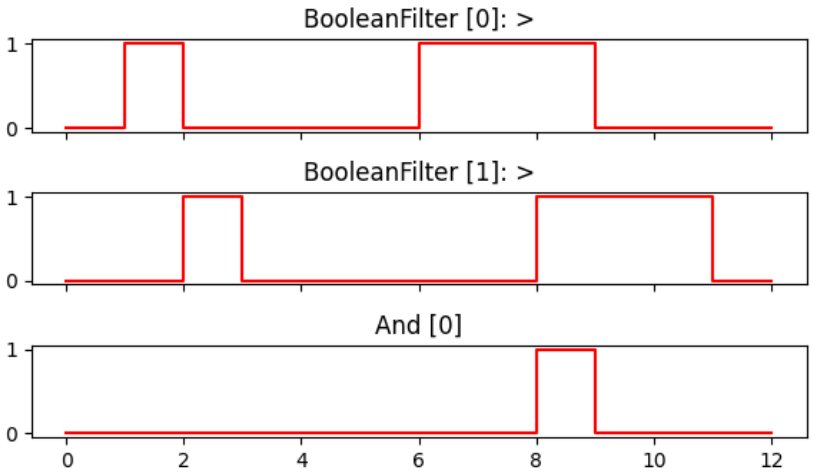


Figure 7: Output from the presented tool of a logical AND example in Boolean semantics. The used formula in this example is: $(s1 > 0) \wedge (s2 > 0)$

3.1.3 Until

The algorithm performing the until operation is based on the work of *Maler et al.* [7]. The computation of $s2\ \mathbf{U}_{[a,b]}\ s1$ consists of three operations:

1. Intersection of the two signals: $s2 \cap s1$
2. Back shifting of (1) with the given indices: $(1) \ominus [a, b]$
3. Intersection of (2) and the first signal: $s2 \cap (2)$

As mentioned in the work of *Maler et al.* [7], when back shifting non-unitary signals (signals with multiple `True`-intervals that are interrupted by `False`-intervals) it leads to wrong results. Therefore we need to make all combinations of the `True`-intervals of both signals, apply the until operation on them and then add them together with the help of an *or* operation. The complete algorithm thus looks like:

function UNTIL(s1, s2, a, b)
$result \leftarrow \bar{0}$
$true\_intervals\_s1 = s1.true\_intervals$
$true\_intervals\_s2 = s2.true\_intervals$
for all $inter\_1 \in true\_intervals\_s1$ **do**
for all $inter\_2 \in true\_intervals\_s2$ **do**
$intersection \leftarrow inter\_1 \cap inter\_2$
$intersection \leftarrow intersection \ominus [a, b]$
$intersection \leftarrow intersection \cap inter\_1$
$result \leftarrow result \vee intersection$
end for
end for
return *result*
end function

The output of an example using the until operator in Boolean semantics from the presented tool is shown in Figure 8. The step by step process of the same example is also visualized in Figure 19 and the data that was used can be found in Table 8, both are added in the Appendix.

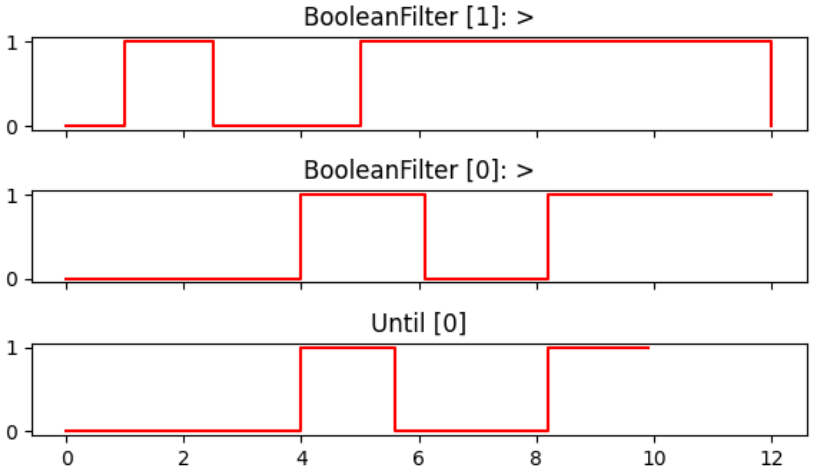


Figure 8: Output from the presented tool of an until example in Boolean semantics. The used formula in this example is: $(s2 > 0)\ \mathbf{U}_{[0.5,2.1]}\ (s1 > 0)$

3.2 Robustness Estimate Calculation

The algorithms used to compute the robustness estimate of signals in a quantitative semantic context are delineated in this section for every operator. The STL formula is considered satisfied in a quantitative semantic context when the robustness estimate of the first time step of the resulting signal is bigger than zero. The higher the estimate, the more robust the given signals are according to the formula and vice versa. If this estimate is smaller than zero, it is assumed that the signals aren't robust. When the estimate is zero, it is unclear if the formula is satisfied or not.

As the robustness is a real value and not a Boolean, it represents the distance to the satisfaction or violation. This eventually propagates in the use of operations on $\mathbb{R}$ and not their logical counterparts that are used on $\mathbb{B}$. **Keep in mind that Section 2.1 also indicates that it is assumed the signals are *finitely piecewise-linear continuous*.**

3.2.1 Negation

A negation in quantitative semantics applies the following rule:

$$r_t = -s_t$$

where r is the result of the negation and s is the input. Both signals consist of real numbers. Figure 9 shows output of a negation example in quantitative semantics from the presented tool.

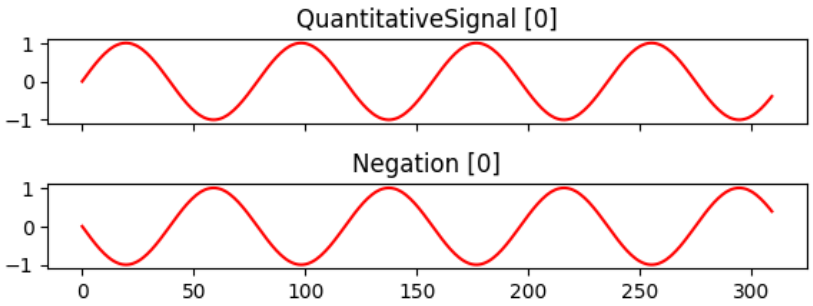


Figure 9: Output from the presented tool of a negation example in quantitative semantics. The used formula in this example is: $\neg s$

3.2.2 And

An AND in quantitative semantics applies the following rule:

$$r_t = min(s1_t, s2_t)$$

where r is the result of the logical AND, $s1$ and $s2$ are the inputs and all signals consist of real numbers. Figure 10 shows output of a logical AND example in quantitative semantics from the presented tool.

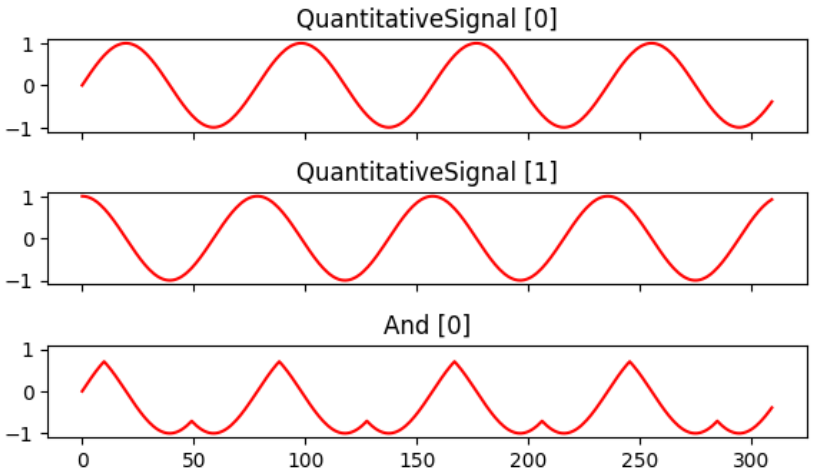


Figure 10: Output from the presented tool of a logical AND example in quantitative semantics. The used formula in this example is: $s1 \wedge s2$

3.2.3 Until

Two different algorithms were inspected for the computation of the until operator in quantitative semantics.

The first method follows the same logic as the above operators, the use of operations on $\mathbb{R}$ and not their logical counterparts. It is based on a couple of articles [8, 9] about the robustness with STL and applies the following rule:

$$r_t = \max_{k \in t \oplus I} \left[\min(s1_k, \min_{k' \in [t,k]} s2_{k'}) \right]$$

where the signal r is the result of $s2\ \mathbf{U}_I\ s1$ with the interval $I = [a, b]$ on which the until operation is performed. Thus the `max` is performed on all time steps $k \in [t+a, t+b]$, **NOT** the k^{th} time step. The input signals $s1$ and $s2$ consist of real numbers.
Figure 10 shows output of an example with the until operator in quantitative semantics from the presented tool. The input data is the same as in the Boolean semantics example and can be found in Table 8, this also shows that the same data can be used in both semantics.

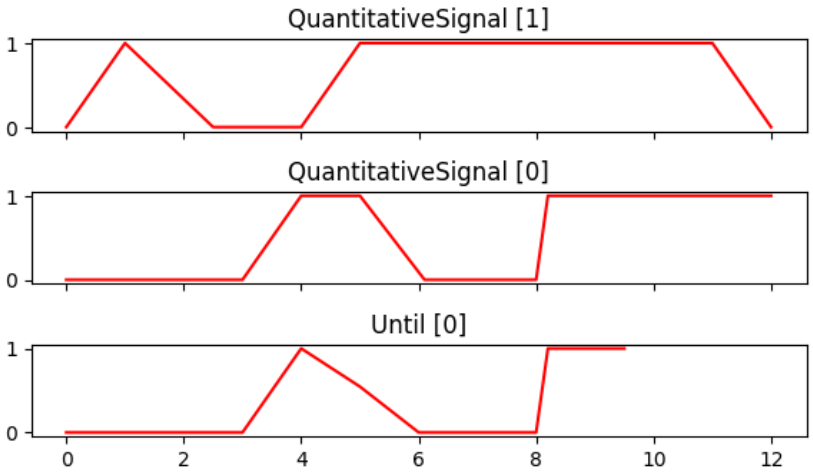


Figure 11: Output from the presented tool of an until example in quantitative semantics. The used formula in this example is: $s2\ \mathbf{U}_{[0.5,2.5]}\ s1$

The second method was first presented in *Efficient Robust Monitoring for STL* [5]. They claim that their method's time complexity is linear to the length of the signal and thus much faster and more efficient than the competition. They achieved this using streaming algorithms, more specifically the *Lemire algorithm* [10]. In essence, they split the signal in different segments and compute the robustness of each segment to eventually end up with the robustness of the whole signal. For the pseudo-code I refer to the original article [5]. However, while implementing the algorithm using the article, some mistakes and remarks were found. Therefore we were not able to reproduce a fully functioning algorithm. Let us first explain the use of the Lemire algorithm and then the remarks are illustrated in Section 4.1.

Each segment goes trough a couple of operations, which operations depend on the derivative of the current segment. We know the derivative of each segment since we assume that the signals are *finitely piecewise-linear*. One of these operations is the *eventually* operation. The algorithm for this operator is based on the Lemire algorithm, an example is shown in Figure 14. It makes use of a sliding window that has a size equal to $b - a$ where the interval of the until is $I = [a, b]$.

The front and the back of the window are time steps $t + a$ and $t + b$ respectively, so changing t makes the window slide. The sorted list M contains the time steps that are in the window and for which the following rule applies:

$$s_{M_i} > s_{M_{i+1}} \tag{1}$$

where s is the input signal of the eventually operator, so s_{M_i} is the value of signal s at time step M_i and that is the i^{th} time step in the list M. If a new time step j is added to M and this rule is not valid, then all time steps k where $s_j \geq s_k$ will be deleted from M.
The sliding window makes such jumps that the front is equal to the first time step in M or the back of the window is equal to a known time step in signal s, whichever causes the smallest jump. The equivalent notation with t is:

$$t = \min(\min(M) - a, t_{i+1} - b) \tag{2}$$

where t_{i+1} indicates the next know time step of s that has not been added to M yet. Each time the window slides, two operations can be performed depending on the locations of the front and back of the window:

- The front of the window is equal to the first time step in M:
  The resulting signal is computed over the interval from the value of t at the last deletion from M to the current t and the first time step from M will be deleted. The current t will thus be the beginning of the interval for the next time the front of the window is equal to the first time step in M.

- The back of the window is equal to a known time step in s:
  The back of the window is the new time step that will be added to M.

This continues until the front of the window is equal to the last known time step of signal s. The resulting signal will then be fully computed.

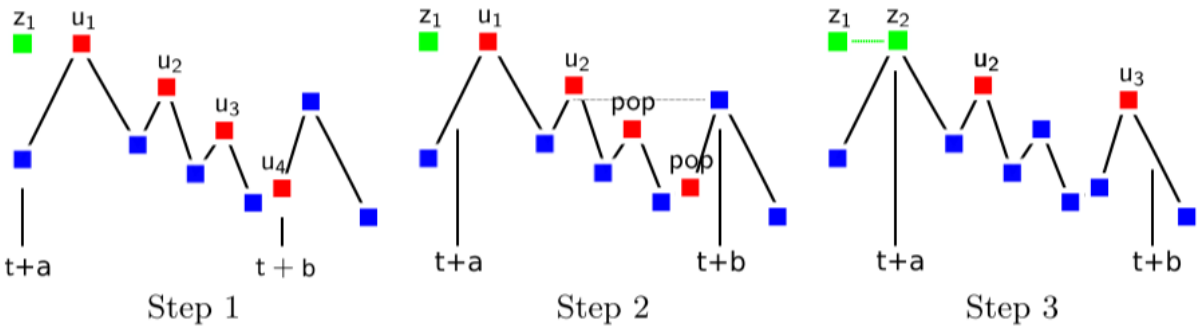


Figure 12: Steps of the Lemire algorithm. Note that $a = 0$ in this example, so $t = t + a$. The horizontal axis is the time and the vertical axis is the value of the input signal s. The first eight steps where the time steps in the interval $[t + a, t + b]$ are added have been skipped. M now contains time steps u_1, u_2, u_3 and u_4, because $t + a$ is equal to the first known time step of s and thus it is deleted from M. The resulting signal z is therefore also computed at that time step. Also, $t + b = u_4$ so it is added to M. At step 2, a new time step appears at $t + b$ which removes u_3 and u_4, because $s_{u_4} \leq s_{u_3} \leq s_{t+b}$ following Equation 1. At step 3, $t + a$ reaches u_1 which is removed from M and z is computed for the interval $[z_1, z_2]$. As the value of t at the last deletion from M was z_1 in the first step and now $t = z_2$.

4 Experiments

This section presents multiple experiments that were performed. They are meant to check the robustness of the algorithms, see how it scales to larger data etc. Also a case study is presented about the use of the tool with Reinforcement Learning in a cart-pole environment.

4.1 Efficient Robust Monitoring for STL

As already mentioned in Section 3.2.3, some remarks have to be made about the algorithm used for the robustness calculation of the until operation in a quantitative semantic context that has been presented in *Efficient Robust Monitoring for STL* [5].

4.1.1 Unknown Segments

In general with the whole algorithm it is very unclear how they handle the unknown parts of signals. Since they assume that the signals are *finitely piecewise-linear continuous*, all values of the signal from the first till the last time step are known. When performing the algorithm however, you will encounter often that, for example, a disjunction has to be taken of two segments that do not have interleaving time steps. There are two options: the result is an empty signal or the segments are stuck together. In the second approach, values are assigned to unknown parts of the signal. *Maler et al.* [7] indicates that what is not known, can't be computed. This approach is favoured by us.

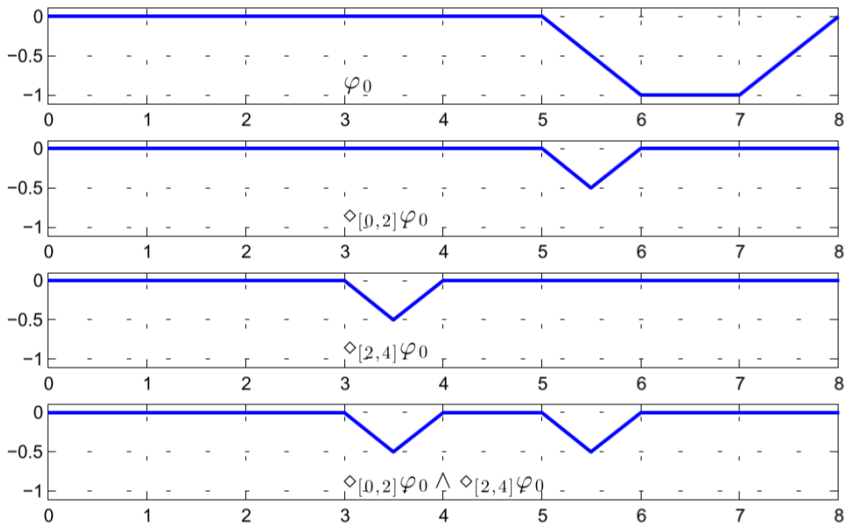


Figure 13: Original example of the article *Efficient Robust Monitoring for STL* [5].

Figure 13 shows an example of the original article. The signal in the second graph continues until time step eight. This seems impossible, because the eventually operator looks 2 steps into the distance and the signal is only eight time steps long. So in fact, the eventually can only be computed till time step six. The same is valid for the third graph. In the fourth graph the two signals are combined, even over the part where there is technically no information about $\Diamond_{[0,2]}\varphi_0$.

4.1.2 Result Computation Positioning

In the eventually algorithm, the computation for the values of z is positioned at the very end of the while loop, so after the addition and deletion of the time steps in M. First of all, computing the values of z in a certain interval every iteration is not wrong but unnecessary. The beginning of the interval will not change until a time step gets deleted from M, i.e. the computed values of z in the beginning of the interval will be overwritten every time until a deletion takes place. This consumes time and resources for no good reason, therefore it is better to only compute the values of z when an item gets deleted from M.
Related to the same problem, if the computation is done after the beginning of the interval is changed. The new interval is then only one element, as the interval will be $[t, t]$.

4.1.3 Eventually Without a Result

In the until algorithm, the signals are split up in multiple segments to improve the time complexity. The segment always consist of two time steps with their values, that represent a segment of the full signal. The evaluation operation is performed on this segment (or a variation after some edits). In the beginning of the eventually algorithm, t is always smaller than the first time step of the input signal. Because the window's back $(t + b)$ should be equal to the first time step to add it to M. If the two time steps in the segment are closer to each other than the window size and the value of the first time step is smaller than the value of the second time step, only one deletion will take place. This is visualized in Figure 14, where the deletion happens in step 2.

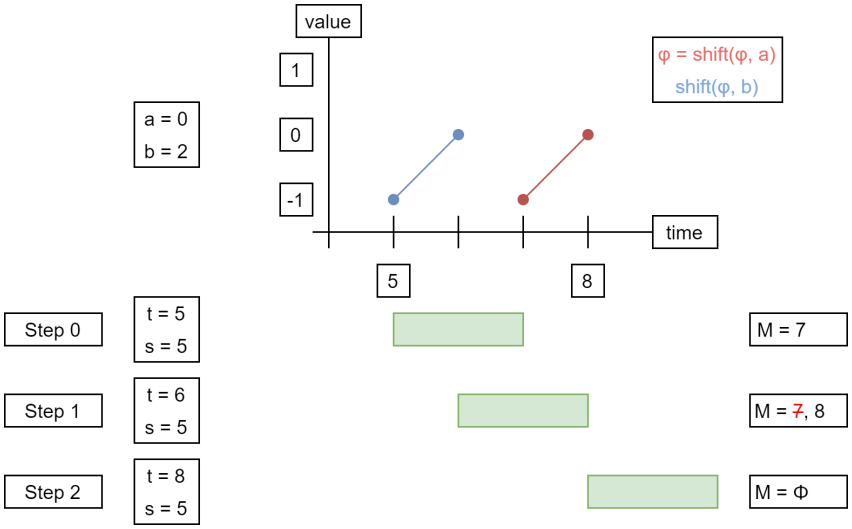


Figure 14: An example of the eventually algorithm on the segment with interval $[7, 8]$ of the example in Figure 13. It represents the process with the green bar being the window of the Lemire algorithm. The result will be empty, because $s < t_0 = 7$ and therefore no values of the resulting signal z will be computed.

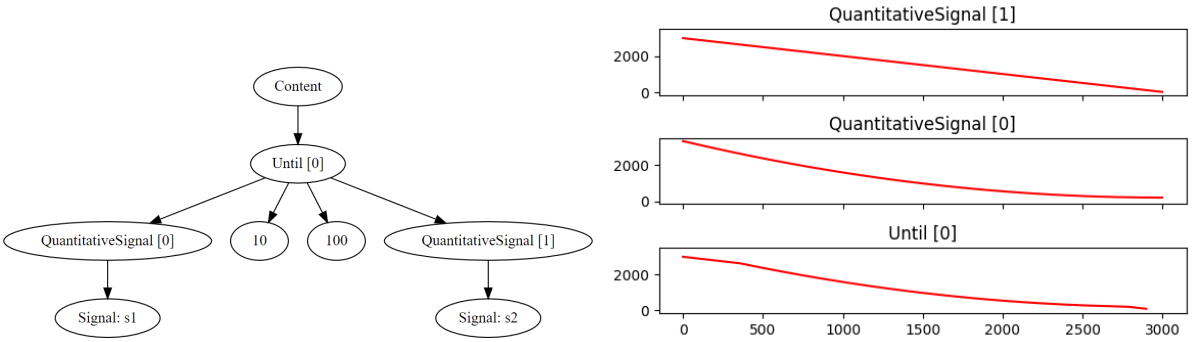
At first glance this doesn't seem a problem, but when all these conditions are true, the following happens. When the second time step is added, the first time step will be deleted from M, because its value is smaller than the new one. The beginning of the interval of values of z that should be computed, is thus still smaller than the first time step of the segment. The condition to compute the values of z is that the beginning of the interval is bigger than or equal to the first time step of the segment. Therefore there won't be any computation and the function will return an empty signal, causing problems in later stages.

4.1.4 Wrong Comparison

When adding a new time step to M in the eventually algorithm, they compare the values of the already present time steps to the wrong value. When a time step is added, it means that the back of the window is at that time step and thus is equal to $t + b$. In their pseudo-code, they compare it to the value of time step $t_{i+1} - b$, which is equal to t according to Equation 2 in Section 3.2.3. Therefore the " $- b$" should be left out.

4.2 Comparing Until Operation Algorithms

The until operation is clearly the most resource intensive of all STL operators. This subsection presents a comparison of different implementations and algorithms to see how the tool performs. As there aren't many implementations on STL in python to compare to, we stumbled upon the Python library called *STLInspector* [11]. The library is not maintained anymore and fully works on Python 2. Python 2 is discontinued for more than a year at the time of writing this article, thus it wasn't possible to compare our implementation to theirs.



(a) STL Tree of: $s1\ \mathbf{U}_{[10,100]}\ s2$

(b) Output graphs

Figure 15: The output of the tool for the example that is used in the comparison of the until operation algorithms that were implemented.

A comparison of the two methods that were implemented has been made. Note that since our second method based on *Efficient Robust Monitoring for STL* [5] is not fully functioning as explained in Section 4.1, the used example is made specifically that it does not encounter any of the problems mentioned. The output of the tool, in Figure 15, was almost exactly the same with both algorithms and therefore only included here once.

Figure 16 goes into further detail of the compared example. The signals used are always decreasing to avoid the problem of Section 4.1.3. Both signals are fairly easy mathematical functions that can be found in the legend on the left graph of Figure 16. The right graph shows that the results of both algorithms are as good as exactly the same, which is expected because they handle the same operation in different ways. The second method should be faster than the first method, but as can be seen in Table 7 this is not the case. Seeing the amount of extra time that is needed for double the amount of values in the signal, does make you realize why it is important to have an algorithm that has a linear complexity. Clearly our implementation of the second method is not able to do what was claimed by the original authors. There are multiple possible reasons. As the second method doesn't fully function, you could argue that there are some parts that weren't understood or implemented correctly.

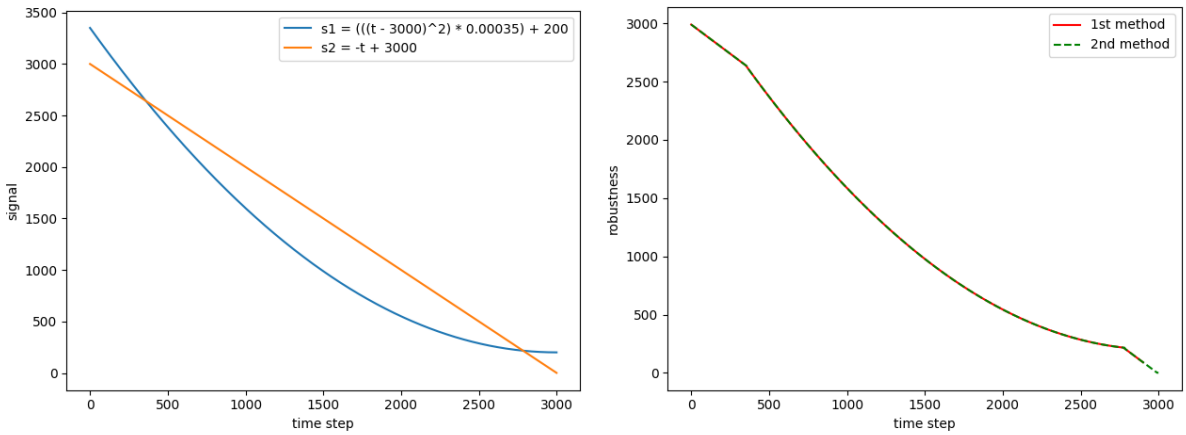


Figure 16: A closer look at the input signals (left) and result (right) of the two different algorithms for the until operation in quantitative semantics.

Method	# values	Total time	Until time
1	3000	33.23s	33.01s
2	3000	46.19s	45.97s
1	6000	130.06s	129.84s
2	6000	179.30s	178.98s
1	9000	274.84s	274.60s
2	9000	403.67s	403.39s

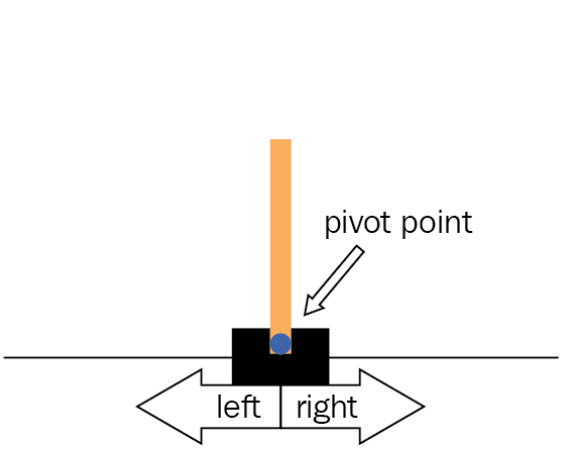
Table 7: Time comparison of the two implemented methods for the until operation in quantitative semantics. The second method is based on *Efficient Robust Monitoring for STL* [5]. The *Until time* column represents the chunk of time from the *Total time* column that was used to finish the until operation.

4.3 Real World experiment

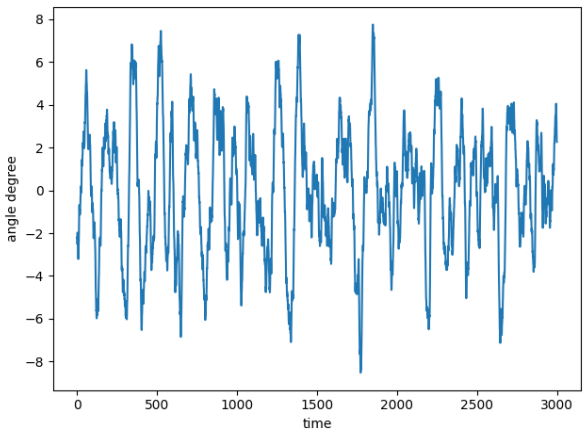
Since the presented tool is made for monitoring and not formal verification, it can be used on systems that are considered black boxes. A hot topic at the time of writing this article is the moral about Deep Learning algorithms. The problem with DL methods is that even the creator of the network does not know how the trained model works. For instance how it decides which class the input belongs to in a classification exercise. Thus, it can be said with certainty that these agents trained with DL methods are black boxes.

As a use case we considered the cart-pole environment. Here you have a cart on a string with a pole on it (Figure 17a) and the intention is that the pole does not fall down. This can be achieved by pushing the cart left or right. When the pole goes past 12 degrees left or right from the vertical axis, it is impossible to get it up again. When this happens, it is considered that the the episode is done. Each episode the pole starts from a perfect vertical position.

To get a system, or in this case an agent, that is able to hold the pole as perfectly vertical as possible, a Reinforcement Learning agent [13] is trained and used. While the agent is performing the actions to keep the pole vertical, the angle of the pole was extracted every time step. These measurements eventually create a real-valued signal, like in Figure 17b, which is the required data to perform monitoring with a formal method like STL.



(a) Visualization of the cart-pole.



(b) The extracted angles of the pole of one episode.

Figure 17: The used cart-pole environment from OpenAI [12], on which the Reinforcement Learning agent is trained and from which a real-valued signal of angles is extracted.

The property that needs to be verified on this example is a typical stabilization property. The output of the angle should never go out of the interval $[-12, 12]$ because then it is considered that the pole will fall, except for the first 150 time steps when the agent starts moving the cart. To make sure the agent is able to hold the pole fairly steady, the angle of the pole should only leave the interval $[-4.5, 4.5]$ for a brief period. We define brief as: the angle has to return into the interval for at least 30 time steps within the next 150 time steps from the moment the angle left the interval. As we don't want it to go constantly very hard left and very hard right, the "at least 30 time steps" is included. This way it has to have a steady period for a short time. In Boolean semantics, the formula then looks like:

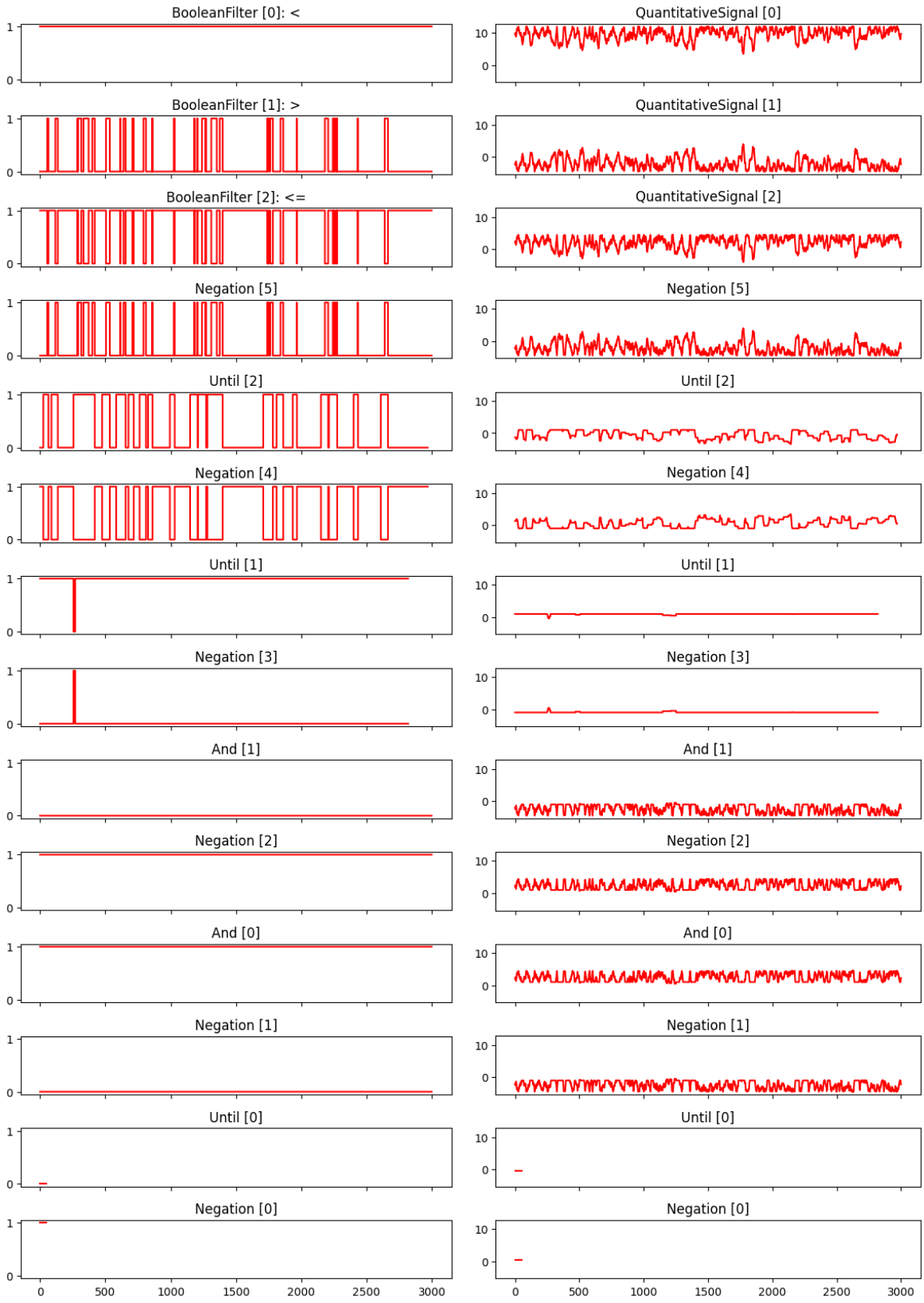
$$\square_{[150,2950]}((|s| < 12) \wedge ((|s| > 4.5) \rightarrow \diamond_{[0,150]}\square_{[0,30]}(|s| \leq 4.5)))$$

Here s is the provided signal from Figure 17b and the tree generated from the formula can be found on Figure 20 in the Appendix. The result of the validation process can be found in Figure 18a and it indicates that this episode indeed is satisfying the formula. However in a Reinforcement Learning context, we are interested in how good or bad the agent did. Just knowing if the agent did achieve what is asked for is not enough. Therefore it is also very interesting to perform the same experiment in a quantitative semantic context.

When working with quantitative semantics, you want to avoid comparing operators, or Boolean "filters", as much as possible. If they are used, the signal is Booleanized and that causes information loss. Thus, using the same formula is not the best option. It is possible to simplify these comparisons. In quantitative semantics, it is considered that a positive value means that the formula is validated. So for instance $x > n$ becomes $x - n$ and it will have the same result in different semantics. Using this simplification, the formula looks like:

$$\square_{[150,2950]}((12 - |s|) \wedge ((|s| - 4.5) \rightarrow \diamond_{[0,150]}\square_{[0,30]}(4.5 - |s|)))$$

The tree generated from the formula can be found on Figure 21 in the Appendix. Figure 18 puts the output of both semantics next to each other. It's clearly visible that every time a signal is negative in the quantitative semantics, the Boolean variant is `False` on the same time steps. Both semantics claim that the formula is satisfied in this example. The quantitative results looks very close to zero, but is effectively around 0.5 over all the 50 time steps.



(a) Boolean semantics

(b) Quantitative semantics

Figure 18: The output of the presented tool of the cart-pole environment example using a formula to check a stabilization property of the angle of the pole on the cart.

5 Conclusion

Temporal Logic and more precise Signal Temporal Logic are formal tools used for verification and became much more popular due to their possibility to specify rich specifications. The idea has been adopted in the hardware industry, is present in software systems and a wide variety of applications. We introduced a tool that can handle STL formulas and compute both the validation in a Boolean and robustness in a quantitative semantic context. While the implementation of the tool only uses the most basic operators of STL, it supports the most popular syntactic abbreviations, making it easy to use in specifying required properties of a system.

Our performed experiments show that there are some remarks about the method to compute the until operation introduced in *Efficient Robust Monitoring for STL* [5] that have to be investigated. While the method didn't fully function, it was possible to compare its outputs to other methods and it showed us that the method does indeed work. However, we were unable to recreate the claim that it has a linear time complexity.

The tool eventually did prove its use, as the use cases of it were explored in the area of Reinforcement Learning. We know for a fact that this is only the top of the iceberg of possibilities and look forward to see how STL and our tool will be used in the future.

References

[1] Vasumathi Raman, Alexandre Donzé, Dorsa Sadigh, Richard M Murray, and Sanjit A Seshia. Reactive synthesis from signal temporal logic specifications. In *Proceedings of the 18th international conference on hybrid systems: Computation and control*, pages 239–248, 2015.

[2] Yashwanth Annpureddy, Che Liu, Georgios Fainekos, and Sriram Sankaranarayanan. S-taliro: A tool for temporal logic falsification for hybrid systems. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 254–257. Springer, 2011.

[3] Tichakorn Wongpiromsarn, Ufuk Topcu, and Richard M Murray. Receding horizon temporal logic planning. *IEEE Transactions on Automatic Control*, 57(11):2817–2830, 2012.

[4] Ilan Beer, Shoham Ben-David, Cindy Eisner, Dana Fisman, Anna Gringauze, and Yoav Rodeh. The temporal logic sugar. In *International Conference on Computer Aided Verification*, pages 363–367. Springer, 2001.

[5] Alexandre Donzé, Thomas Ferrere, and Oded Maler. Efficient robust monitoring for stl. In *International Conference on Computer Aided Verification*, pages 264–279. Springer, 2013.

[6] Terence Parr. *The definitive ANTLR reference: building domain-specific languages*. Pragmatic Bookshelf, 2007.

[7] Oded Maler and Dejan Nickovic. Monitoring temporal properties of continuous signals. In *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*, pages 152–166. Springer, 2004.

[8] Iman Haghighi, Noushin Mehdipour, Ezio Bartocci, and Calin Belta. Control from signal temporal logic specifications with smooth cumulative quantitative semantics. In *2019 IEEE 58th Conference on Decision and Control (CDC)*, pages 4361–4366. IEEE, 2019.

[9] Lubos Brim, Tomas Vejpustek, David Šafránek, and Jana Fabriková. Robustness analysis for value-freezing signal temporal logic. *arXiv preprint arXiv:1309.0867*, 2013.

[10] Daniel Lemire. Streaming maximum-minimum filter using no more than three comparisons per element. *arXiv preprint cs/0610046*, 2006.

[11] Stlinspector 1.1.4. https://pypi.org/project/STLInspector/. Accessed: 2021-03-30.

[12] Cartpole-v0. https://gym.openai.com/envs/CartPole-v0/. Accessed: 2021-04-02.

[13] jordanlei. Deep reinforcement learning with cartpole in pytorch. https://github.com/jordanlei/deep-reinforcement-learning-cartpole, 2020.

Appendix

s1	s1_t	s2	s2_t
0	0	0	0
1	1	0	1
0	2.5	0	2.1
0	3.5	0	3
0	4	1	4
1	5	1	5
1	6	0	6.1
1	7.155	0	7
1	8	0	8
1	9	1	8.2
1	10.02	1	10
1	11	1	11
0	12	1	12

Table 8: The data used in the examples of the until operator in both the Boolean and quantitative semantics in Sections 3.1.3 and 3.2.3.

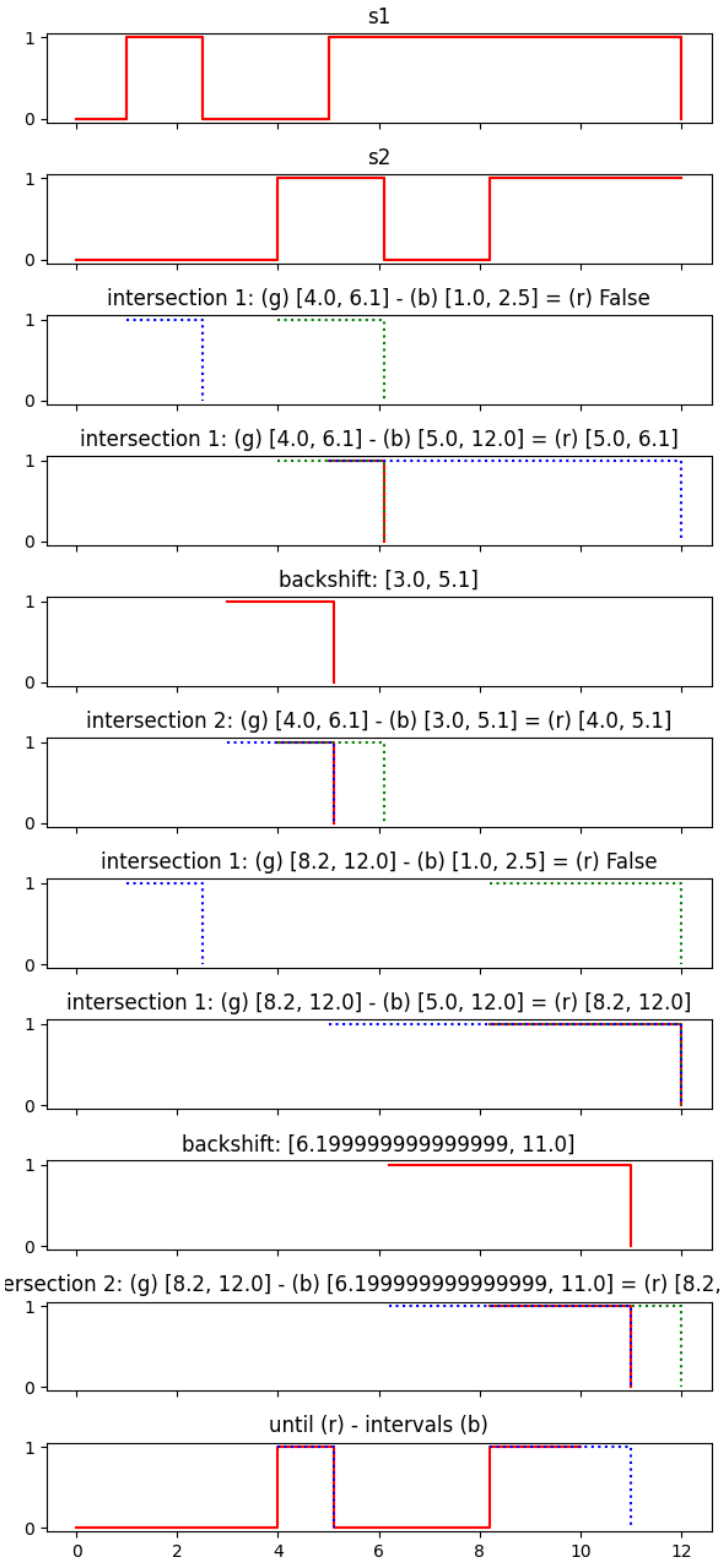


Figure 19: Visualization of all the steps in the algorithm used to compute the result of an until operation in Boolean semantics of an example as is explained in Section 3.1.3. The used data is shown in Table 8 and the formula in this example is: $s2\mathbf{U}_{[0.5,2.1]}s1$

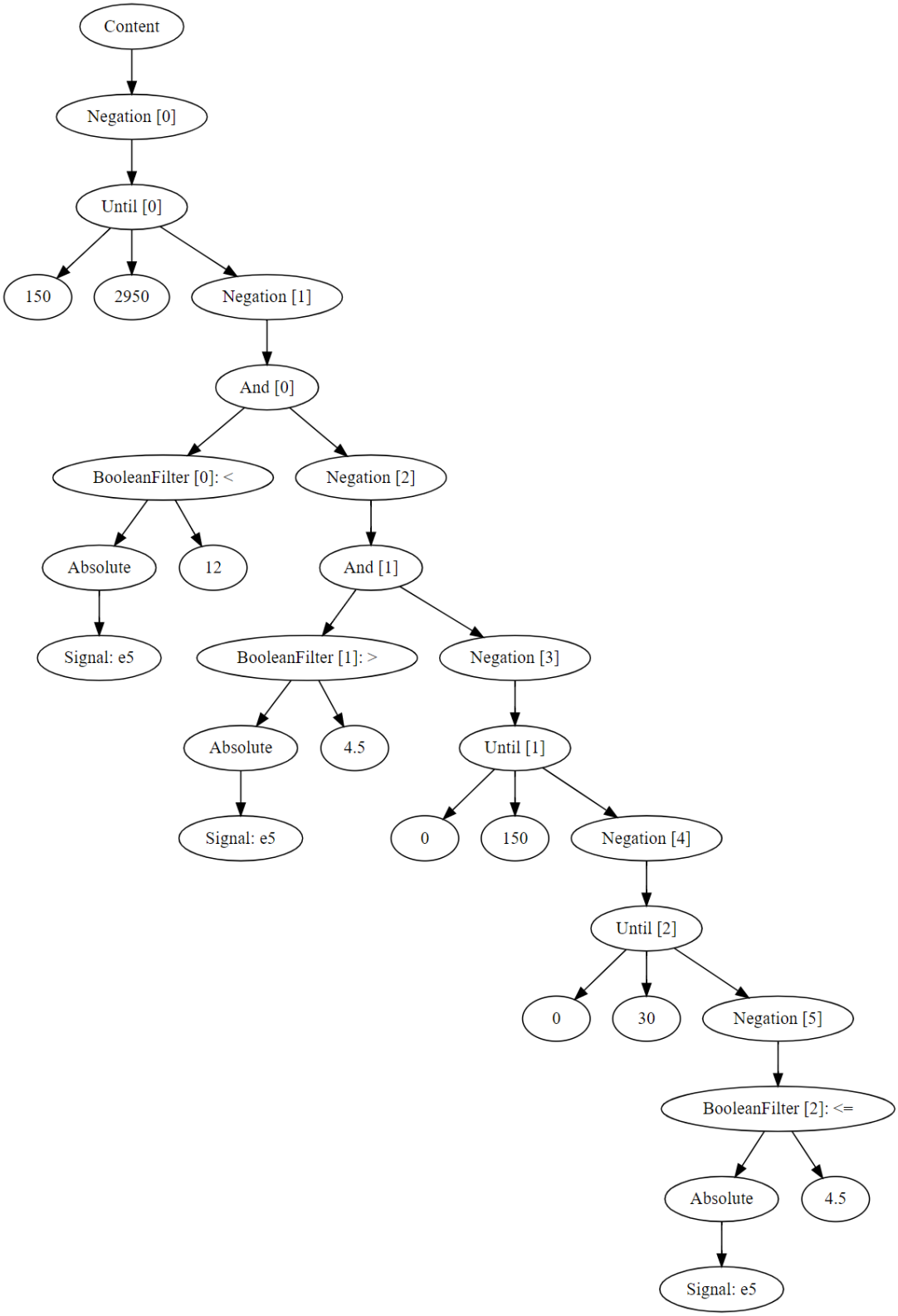


Figure 20: The STL Tree generated and used by the presented tool for the experiment in Section 4.3 using a Boolean semantics. The used formula is:

$$\square_{[150,2950]}(((|s| < 12) \wedge ((|s| > 4.5) \rightarrow \diamond_{[0,150]}\square_{[0,30]}(|s| \leq 4.5)))$$

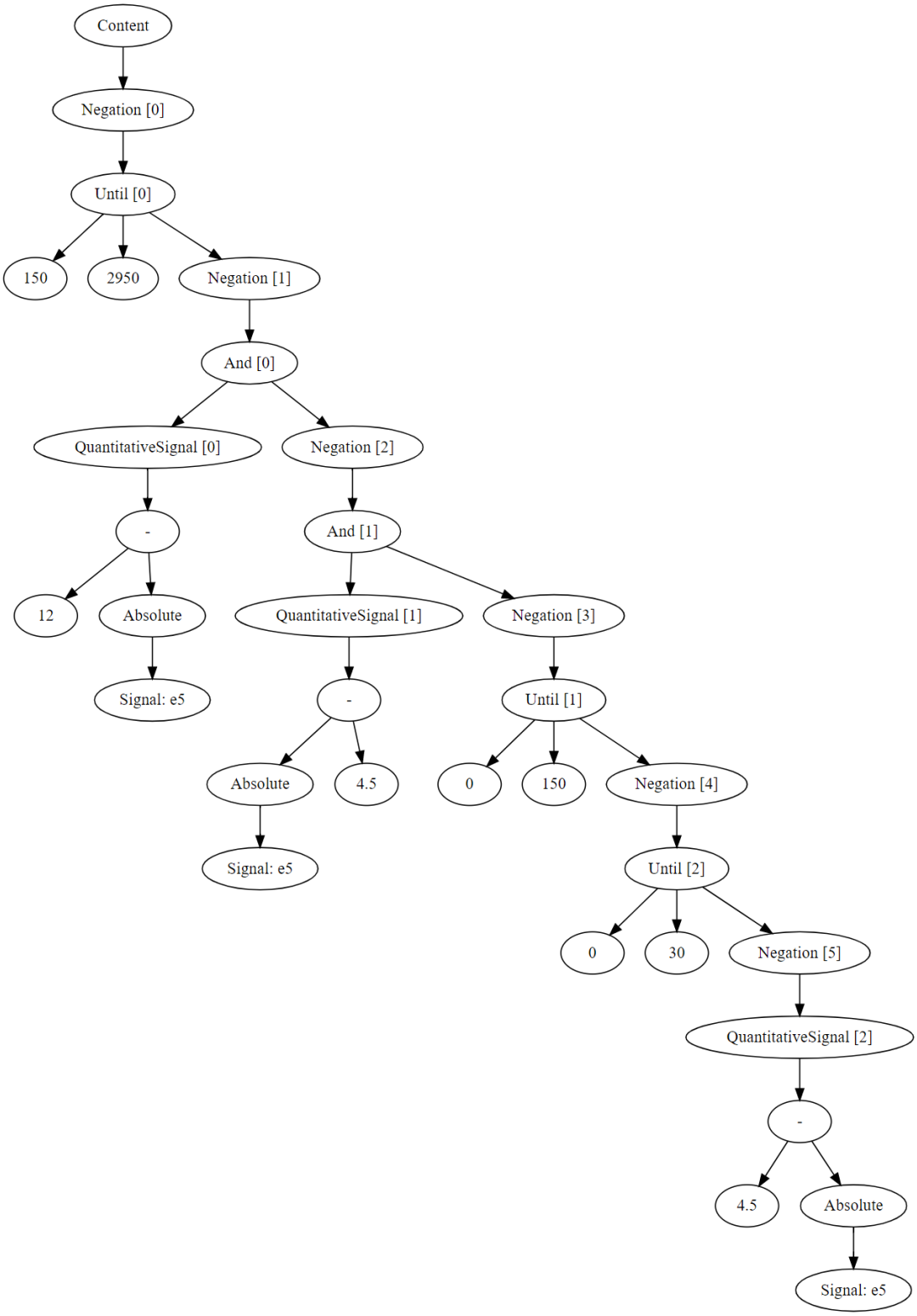


Figure 21: The STL Tree generated and used by the presented tool for the experiment in Section 4.3 using a quantitative semantics. The used formula is:

$$\square_{[150,2950]}((12 - |s|) \wedge ((|s| - 4.5) \rightarrow \diamond_{[0,150]} \square_{[0,30]}(4.5 - |s|)))$$