Academic year
2021-2022

Learning an Efficient Online STL Robustness Monitor

Pieter Hendriks

Master's thesis
Master of Science in computer science: computer networks

Supervisor
prof. dr. Guillermo A. Pérez, University of Antwerp

Cosupervisor
Ritam Raha, University of Antwerp

University of Antwerp
Faculty of Science

Abstract

A system formalized through STL can be monitored using the robustness value. Computing this value is a linear complexity operation in the size of the input. In online contexts, this verification can lead to performance issues for the monitored system. Through an extensive rework of a previously introduced offline STL robustness computation tool, we significantly increase its performance. After guaranteeing its correctness through the introduction of a comprehensive test suite for this tool, we use it to generate a large, labeled robustness data set. The existence thereof enables the training of a neural network robustness estimator. Using convolutional layers, we enable the estimator to handle signals of variable lengths to allow more flexible use. In our validation set, the estimator performs admirably. Using the output of the estimator as an indication to decide if full robustness computation is necessary can result in significant performance improvements for a monitored system. Entirely replacing the traditional computation is not advisable, since no guarantees can be made about formula satisfaction using any estimation. A system where they work in tandem obtains the best of both worlds: guarantees on the computed robustness value when necessary, improved performance when possible. We present an example implementation of such a combined system and use it to illustrate the attainable performance improvements.

Abstract

Een systeem dat door middel van STL geformaliseerd is, kan gemonitord worden door van de robuustheidswaarde gebruik te maken. De berekening van deze waarde is een operatie met lineaire complexiteit in de grootte van de input. Wanneer deze verificatie uitgevoerd wordt op een actief systeem, kan dit resulteren in significante degradatie van de performantie. Door een uitgebreide herwerking van een eerder beschreven programma voor de berekening van deze robuustheidswaarden (in de context van inactieve systemen) wordt een grote verbetering in performantie behaald. Nadat de correctheid van deze functionaliteit gegarandeerd werd door de introductie van uitgebreide geautomatiseerde tests, gebruiken we deze om een grote hoeveelheid aan robuustheidswaarden te generen. Deze data staat ons toe om een artificïeel neuraal netwerk te gebruiken om een robuustheidswaarde schatter te ontwerpen. Convolutieve lagen staan de schatter toe om om te gaan met invoer van variabele groottes voor gebruik in algemene context. De robuustheid-schatter behaalt binnen onze validatietests een indrukwekkend resultaat. We stellen dat de geschatte waarde als indicatie kan dienen om te bepalen of de volledige robuustheidswaarde berekening uitgevoerd moet worden of dat deze simpelweg weggelated kan worden. Op deze manier kan gebruik van deze schatter voor significante performantieverbeteringen zorgen bij actieve systemen die gemonitord worden. De berekening volledig vervangen kan niet aangeraden worden, aangezien er geen garantie over de satisfactie van een formule kan gegeven worden door een geschatte waarde. Een systeem waarin de traditionele berekening en de schatter die wij presenteren samenwerken, behaalt het beste van beide: wanneer nodig kunnen garanties over de berekende robuustheidswaarde gemaakt worden en in andere gevallen wordt betere performantie behaald. We implementen een monitor waarin deze samenwerking gebruikt wordt en illustreren daarmee de performantieverbeteringen die behaald kunnen worden.

Contents

1 Introduction

1.1 Specification & verification

System specification and verification are important steps in system design. These steps are innately related, as requirements for the system are set during specification and these requirements are exactly what must be verified. Verification tools allow automated verification of systems, which is preferred to manual verification (since humans are quite error-prone). The use of formal methods for specification and verification creates an environment where requirements can be unambiguously stated, and later verified, even when these requirements are non-trivial (i.e. more complex than, for example, up-time and stability).

Multiple approaches can be taken when a system must be verified. First, to completely guarantee system correctness, *formal verification* is the main method. Formal verification involves proving, mathematically, that no undesired behavior can result from the system. It is a computationally intensive task and does not scale well to complex systems. In fact, for a lot of classes of Cyber-Physical Systems (CPS) the verification problem is undecidable [1]. It is, however, used for core functionalities for safety-critical systems, such as air traffic control [2] [3], compilers [4], OS kernels [5] (and more).

Alternatively, in case no strong correctness guarantee is required, one can observe the system's behavior and determine if those outcomes are in line with its specification. This is *runtime verification* or *monitoring* [6]. A major advantage of this method is that no insight into the system's internals is required. A system can be monitored simply through observation of its in- and outputs (and knowledge of the desired behavior, of course). The downside of this method is that only the correctness of the observed cases is guaranteed — it cannot reasonably guarantee that there does not exist an edge case that results in undesired behavior. The lower computational requirement of this method has made it more popular recently for most use-cases [7].

1.1.1 Monitoring

Within monitoring, we distinguish two cases: offline and online. Offline monitoring occurs after the system being monitored has fully performed its task, online monitoring, on the other hand, occurs while the system is still running. Using the full output trace, an offline monitor can determine whether the system behaved within the specification. These offline monitors are generally easier to implement than their online counterparts and do not have any stringent performance requirements [6], since they do not have an impact on the runtime performance of the system.

In some contexts, offline monitoring is insufficient. As an example, consider a monitor for the movements of a robot arm. One sensible thing that may be specified is that the arm should not hit unexpected objects that appear within its range of motion. In the case of an offline monitor,

this specification would not actually provide any safety benefits: the monitor would tell you that the arm had broken its specification after the fact, possibly having hurt someone in the process.

An online monitor would be more suitable here, since the arm's behavior can be modified based on the monitor's results. If the monitor detects the arm is close to breaking its specification (hitting something, in this example), the arm could simply stop and wait for operator intervention (or some other error recovery method). In this way, an online monitor provides benefits over the offline alternative; the runtime behavior of the system can be modified based on the monitor's observations.

As noted, offline monitors tend to be easier to implement than online monitors. The main challenges online monitors face are insufficient information and performance. An online monitor can encounter situations where in order to definitively state whether a system is breaking the specification or not is through some information that is not yet available. An online monitor must somehow account for this, for example by assuming a worst case: if any sequence of events exists to make the system break its specification, within some limited time period, assume that it will. This kind of approach is generally favored in systems that are critical to safety. Other approaches are assuming a best case (if any sequence allows success, assume success), neutral (compute best guess based on current information using truncated semantics) and more [8].

Problem statement [1] The performance concerns, similarly, are system dependent - in case the system's performance is not critical, the monitor's impact on the performance will similarly be non-critical. However, in cases where the performance is important, the impact must be minimized. To achieve this, research has been done to improve the performance of online monitoring techniques such that they can be implemented in real-time [9]. Our work introduces a novel approach for the optimization of online monitoring by attempting to eliminate a significant portion of the computations.

1.2 Signal Temporal Logic

Many languages for formal specifications exist. The one we will be focusing on is Signal Temporal Logic (STL). This formalism builds on Linear Temporal Logic (LTL) [10] by additionally introducing real-valued variables (signals) and the concept of dense (continuous) time [11]. LTL, itself, is an extension of classical Boolean logics, featuring the addition of discrete time and temporal operators [12].

Temporal operators work on the input signals, as well as considering a interval in time. The ability to model time in that way makes temporal logics ideal for modeling reactive systems [7]. Something similar to "after x, y must happen within 2 seconds" is not easily expressible in purely Boolean logics, but trivial in temporal logics.

A real-valued variable has more meaning than the simple true/false we see in Boolean logics. Consider, for example, a formula that must make decisions based on some observed temperature. In order for a Boolean logic to handle such a naturally continuous value, its range must be discretized (e.g. through use of binning). Once the space is discrete, the various bins can be enumerated and a Boolean formula for such a decision created. Depending on the amount of bins, though, that may result in a lot of necessary variables in the Boolean formula.

A real-valued logic introduces a significant simplification for this example. Rather than requiring

[1] Further expanded in the rest of the introduction, we focus specifically on online STL robustness monitoring rather than online monitoring in general

discretization of the space and multiple variables to indicate the bin, direct comparison of the observed temperature is possible to obtain the desired result.

Just like the value for any given variable, in Boolean logics formula satisfaction is a Boolean value — a formula is either satisfied or it is not. In real-valued logics, there is more flexibility. Satisfaction remains a binary value, of course. However, by using a real value, it is possible to give an indication how far away from any boundary condition the current state of the variables is. This value is called the *robustness* [13]. A positive robustness indicates formula satisfaction, a negative value indicates dissatisfaction. The magnitude of the robustness is a measure of the change required to the variables in the formula to reach the boundary condition it specifies.

1.2.1 Robustness

Our intuition for an implementation of a monitor using STL is that whenever the robustness, computed for the STL formula describing the desired system behavior, is positive, the system is behaving within the bounds of its specification. Monitoring, then, boils down to computing the robustness and ensuring that it remains positive (or, if the formula is not exactly the system's specification, within acceptable bounds which are situation dependent).

Basic algorithms to compute the robustness for all STL operators can be derived directly from the syntax of the logic [7]. These are computationally inefficient in some cases, so work has been done to optimise them [7]. The syntax-based algorithms, and their optimisations, are all offline monitoring algorithms.

RTAMT is an implementation of an STL monitor [14], with a tool to interact with the Robot Operating System (ROS) [15]. This link with ROS allows use in the monitoring of CPSs. RTAMT supports discrete- and continuous time interpretations of the STL specification, as well as offline and online monitoring. However, in the case of online monitoring for continuous time STL, the implementation is incomplete; there is no support for *eventually*, *until*, or *always* in that context.

For other real-valued temporal logics, similar efforts to implement online monitors exist [16].

Other work has made strides to create implementations of these operations in online contexts [1] [8]. The main issues these implementations face, similarly to other online monitoring work, are handling the cases where insufficient information is available to fully evaluate a formula and the performance impact on the running system.

Predictive monitoring has also been introduced; it is an extension of online monitoring wherein a predicted range wherein the robustness value likely falls is computed [17]. The algorithm further provides a probabilistic guarantee on the correctness of the computed range.

The robustness computation is computationally expensive. The best known offline algorithms achieve a time complexity that is linear in the size of the input signals for the computation of the robustness for a given formula. Unfortunately, the algorithms achieving this efficiency do not easy translate to online contexts, where the complexity is higher as a result; the range of time specified for operators is an additional factor for the complexity in online contexts. This is further detailed in Chapter 2.

We build on a previous work that implemented the syntax-derived algorithms for offline monitoring [18]. An attempt to adapt the work presented in *Efficient Robust Monitoring for STL* [7] was made, but unfortunately unsuccessful. We correct the behavior in the error cases they

identified To facilitate this correction, we extensively refactor the code making up this tool and introduce a test suite. Following the introduction of the test suite, additional problems were identified which we have also addressed.

These corrections were required to generate a correct data set to train our estimation tool (further described below), since the robustness computation using the syntax algorithm was too slow to reasonably generate large data sets. During our work, we further identified a few minor cases where the syntax algorithm deviates from the expected solution (which were not mentioned in the original work [18]). The corrections we present are, thus, a requirement for the generation of a correct data set.

As part of our corrections, we have re-implemented the timed eventually operation. Translating the *Efficient Robust Monitoring for STL* [7] implementation of this operation resulted in incorrect behaviour in our test cases. Likely, this was the result of an error in our implementation. After considerable effort failed to directly identify the source of the bug, the most viable strategy for identification of our mistake became re-implementation of the algorithm based on the source material [19] and comparing the implementations.

At that point, performing the re-implementation without attempting to correct what we had at the time is simpler. A second contribution is the implementation of this algorithm. We base ourselves on the same sliding window min-maxing implementation [19], but re-do the modifications to support a window containing a variable amount of sample points. We present the full implementation and the underlying concepts.

1.2.2 Learning a Robustness Estimator

Efficient known algorithms for STL robustness computation achieve linear complexity results in offline evaluation [7]. If these algorithms, without further modification, are used in an online context, their performance decreases dramatically. In an online context, a new robustness value must be computed whenever a new input sample point is introduced. To achieve best performance, we can limit the length of the signal passed into the computation based on the time intervals in the formula (values outside any interval have no impact on the result). The algorithm can then be used on the relevant subset of the signal whenever a new data point comes in.

However, this effectively makes the complexity of the algorithm dependent on the time intervals in the formula again (since they now determine the size of the input signal). In online contexts, performance improvements for this type of evaluation is desirable. Other work has modified these algorithms for online contexts [8] [1].

Reason for feasibility study: We introduce a novel approach for improved performance in online monitoring. Assuming one can generate an estimate of the robustness for some given formula and input (and have some knowledge about the expected error), the magnitude of that robustness estimate may be used to forego robustness computation entirely. Of course, this is inadvisable in safety-critical applications, but in other contexts, significant performance gains could be achieved.

This is exactly the approach we take: in this work, we introduce a method for robustness estimation. The goal of this system is, as mentioned, for it to be used in conjunction with the existing algorithms. It should serve an indicative function for the system being monitored; when the generated estimate is below acceptable values (or some other criterion, such as unexpected

variation), the system can fall back on the full robustness computation. When the estimate is high (and indicates a low variance), the robustness computation can be omitted.

In this way, we hope to improve the performance of online STL monitoring systems when no (or relatively little) unexpected behavior occurs, while maintaining accuracy when it matters (by then requiring that the robustness is computed in the traditional way). For systems that often operate near the boundary of their specification, this approach is intuitively sub-optimal; the computation of an estimate will take time and often result in the conclusion that the full computation is necessary. If this happens sufficiently often, performance may actually be worse after the introduction of an estimator.

Clearly, the intent is not to fully replace the existing algorithms. Rather, we wish to supplement them. By eliminating the expensive robustness computation where possible, online STL monitoring can gain a significant boost in performance through the implementation of an estimator.

We further note that any future improvements in online monitoring performance will not influence the results we present, unless they are so significant that computation of an estimate becomes slower than the full computation. Improvements made to the computation of online robustness values can simply be adapted into the implementation of the monitoring algorithms; the implementation of the approximation tool and the robustness computation algorithms are entirely separate. The performance gain we present is obtained simply through eliminating unnecessary computations.

Estimated robustness values will deviate from the expected values. Intuitively, we expect these deviations to be less important when the magnitude of the robustness is high (since inherently, the sign of the robustness carries more meaning than the magnitude). Problematic, though, are deviations where the sign of the estimation does not match the sign of the actual robustness. We would expect this to be less frequent as the magnitude of the robustness increases — a slight deviation will not change the sign of a value when the value's magnitude is high. In cases where magnitude is low (and thus there is a higher likelihood of the estimate having the wrong sign), the existing algorithms can be used to guarantee correctness of the computed robustness.

This workflow limits the performance impact of the expensive robustness computation by eliminating it where possible, based on the value of a robustness estimate. When guarantees about the robustness are required (i.e. when the estimate is low in magnitude), the known algorithms are used to make this guarantee. Of course, this entire line of reasoning relies on the existence of a good approximation. Without a good approximation, we cannot reliably skip robustness computations and so this work would offer no performance benefit (and perhaps even result in performance degradation).

Recall that the steps necessary for the robustness computation can be derived directly from an STL formula. This, in essence, means that the formula defines a function from its input domain to the range of real numbers (the efficient algorithm simplifies its computation, but the function remains identical). Approximating the robustness computation is then simply function approximation.

Function approximation techniques exist that achieve arbitrary non-zero errors given enough time and data [20]. Achieving near-perfect accuracy is not part of the scope of this work; rather than attempting to optimize our approximation to perfection, we aim to show that the approximation for this robustness function is possible without egregious errors using a reasonably sized data set and that the resulting estimator achieves significant runtime performance gains over the fastest known algorithms.

1.2.2.1 Estimator Design

Function approximation is a well-studied problem and many techniques for approximating functions exist. They are typically divided into three categories: supervised learning, unsupervised learning and reinforcement learning [21]

Reinforcement learning can be easily excluded; the concept for reinforcement learning is studying how an agent, acting in an environment, may optimize its actions to achieve the best possible outcomes. However, conceptually, this does not quite match what we are looking for. The output of our estimator does not really influence the next states it will observe, whereas in reinforcement learning, the agent's actions directly influence the environment (and thus its following observations). While unsuitable for the robustness estimator, we do use reinforcement learning in our work; we consider the training process of reinforcement learning agents to generate example data sets.

Unsupervised learning, then, typically finds patterns in data as in, for example, k-means clustering. The very nature of unsupervised learning prevents the approximation of a specific target function; since the learning is unsupervised, the techniques focus on attempting to find patterns in the data. However, there is no way to guide the learning process — that would be supervision. Since we have a known target function in our use-case, these methods are unsuitable.

Applicable techniques: This leaves us with supervised learning. Supervised learning attempts to approximate a target function based on a data set containing sample input-output pairs of the target function. This category encompasses learning methods such as linear regression, neural networks [22] [23], decision forests [24], gradient boosted trees [25], support vector machines (SVM) [26] and more. Each of these latter four is a universal approximator (i.e. can approximate any continuous target function down to an arbitrary non-zero error, given sufficient time, data and inner complexity).

A neural network, very simply stated, is a series of linear combinations of variables mixed with non-linear function evaluations, where the value of the parameters in the linear combinations can be modified during training. The output of neural networks is naturally continuous and thus ideal for regression problems. By interpreting the output of a neural network through the use of a softmax activation function, the output can be interpreted as probabilities that a certain input is a member of a certain class (in which case each output node represents a single class).

A decision forest is a collection of decision trees, wherein each tree is created by analysis of some subset of the training data using bagging. A variation of this, random forests, further restricts the features that can be used to construct each tree to a random subset. By constructing multiple trees in this way, and using some combination of their outputs as the eventual result (in classification problems typically the mode, in regression problems typically the mean), this technique mitigates the over-fitting issues often seen in simple decision trees.

Gradient Boosted Trees (GBT), similar to a decision forest, is an ensemble method that combines the outputs from individual trees. Boosting is a machine learning algorithm intended to reduce bias and variance in supervised learning [27]. The basic idea is to use *weak learners* (which are very simple prediction models) and combining them sequentially. The goal is to always have the next learner correct for the error left after the previous learner's evaluation. At each step, the goal is to filter out the examples that particular learner would get correct and pass any others to the next learner. In GBT, these weak learners are typically decision trees of a specific size. The sequential operation is a major distinction between GBT and decision forests (where evaluation

can occur in parallel).

SVM is a technique that, essentially, finds the optimal hyperplane separating two categories of data. The optimal hyperplane is the one separating the data such that the margin (i.e. distance) between the plane and the data points of each category is maximal. To categorize data into more than two categories, the problem is split up into multiple binary classification problems. An extension known as Support-Vector Regression also allows the handling of regression problems (as in this work).

While we cannot state that one of these methods is better than the others a priori (by the No Free Lunch theorem [28]), we limit this study to one learning method. We note that the application of our estimator is not known ahead of time. The STL formula (and thus the target robustness function) may be of arbitrary complexity. A comparative study has found that neural networks tend to perform relatively better, when compared to other methods, when data becomes more complex [29]. Combined with neural networks' well-known affinity for large data sets (which we are able to generate following the optimisations of the STL robustness computation tool), this has lead to the choice to implement our estimator using an Artificial Neural Network (ANN). Within neural networks, there are still multiple options to consider. These are discussed later.

Choice of experiment: To create the required amount of data, we use OpenAI gym environments [30] [31]. These are environments designed to train reinforcement learning agents to solve a specific task. We choose two of them (cartpole and mountain car) and, by training a reinforcement learning agent on the environments, generate a large data set. Since the criteria for success in the environments are clearly defined, we can create STL formulas describing these restrictions. From the environment outputs during the training process and the STL formulas we create, we can compute the robustness for every step of training. This results in a large set of labeled data suitable for supervised learning using a neural network.

Since the signal input does not, in general, have a predictable amount of input elements, we cannot easily use a simple fully-connected, feed-forward architecture. Typical approaches for enabling variably sized inputs for this class of neural network is to pre-process the input in such a way that the input does not exceed a specified maximum size. In case the input is larger than this size, some values must be cut (or combined). In case the input is smaller than this size, a number of the input values will be some default value (often 0).

Other neural network architectures are better suited to input size variance, though. One such architecture is the Recurrent Neural Network (RNN). RNNs build on the traditional architecture with the addition of a memory element; for any node in the network, its previous outputs can be considered as inputs during computations. The exact amount of time these values are re-used is a tunable hyperparameter. For these networks, the 'memory' implementation can also vary — two significant implementations are Long Short-Term Memory (commonly referred to as LSTM) [32] and Gated Recurrent Units (commonly known as GRU) [33].

For these networks, the input should generally be fed in as a sequence of items. For an online monitor, where new values are received one-by-one, this is very natural. The introduction of the memory elements increases the computational complexity of the model, though, resulting in a slightly slower computation. Combined with the fact that these networks are relatively difficult to train, this has resulted in the decision against the use of RNNs in this work.

Instead, we consider a Convolutional Neural Network (CNN). In this architecture, kernel convolutions are applied to a set of inputs, generating a set of outputs. This is a shared-weight operation that operates like a sliding window over the input, allowing it to deal with inputs

of varying sizes. Of course, a longer input will generate a longer output - we normalize that through the use of pooling somewhere in the network architecture, to obtain a fixed number of values.

From there on, we use fully-connected feed-forward layers that will eventually generate our two-valued output. This output is interpreted as the parameters for a Normal distribution. In this way, we obtain some measure of uncertainty as part of the robustness estimate. This may be an important factor in the full computation omission decision.

An alternative implementation would be to begin with a pooling layer, without the initial convolutional layers, and use fully-connected feed-forward layers only. We have decided against this architecture because the architecture will feature many more parameters than the current architecture (due to the parameter re-use in convolutional layers) and thus be significantly harder to train. Given sufficient training, though, such an architecture can likely obtain similar results to our architecture.

CNNs typically see use in image processing [34], natural language processing [35] and time series forecasting [36] (among others). In those contexts, their application is very natural since the way the convolution matrix slides over the input is very similar to how a human might approach that kind of problem. In our case, the sliding window closely matches what the algorithmic solution for the problem is so we expect this type of ANN to be an effective way to learn the function.

As far as we are aware, our work is the first to consider ANNs to approximate the robustness function defined by an STL formula. This technique offers a potential benefit in the performance of online STL robustness monitoring. By using the estimation to decide whether a full computation is required (for example, when the estimation is near the specification boundary), this system can result in significant performance improvements by eliminating a lot of unnecessary robustness computations. The exact improvement that can be achieved is implementation dependent.

This work serves as an initial study into the viability of this type of approximation to optimize the performance of online STL monitors. Further work will be required to consider whether the results we find generalize to the far more complex environment of real-world applications.

2 Signal Temporal Logic

We briefly introduce the notation and semantics used in this work. Most are identical to those used in *Efficient Robust Monitoring for STL* [7].

2.1 Signal

Signals are variables whose values vary over time, which makes them a function of time as in Definition 2.1.1. Boolean signals trivially fall within this definition, as they simply limit the range to the set $\{0, 1\}$ (alternatively, some equivalent True-False representation, such as extending $\mathbb{R}$ with the set $\{\top, \bot\}$ and using these new values to represent True and False, respectively).

Definition 2.1.1 (Signal)
A signal s is function $s : \mathbb{T} \to \mathbb{R}$. We add the simplifying assumptions that signals are finite, piecewise linear and continuous (Definition 2.1.2). To exactly represent a signal, then, we use a (sorted, ascending by time) series of $\{t, v\}$ tuples ($t \in \mathbb{T}$, $v \in \mathbb{R}$) called *checkpoints* or sample points.

We further define the derivative for signals to allow interpolation computations. These are values in $\mathbb{R}$, since they are the derivative of a linear segment (i.e. the slope of that segment). This extends the definition of a signal to $\{t, v, d\}$ triples. For two sequential checkpoints in a signal, $\{t_0, v_0, d_0\}$ and $\{t_1, v_1, d_1\}$ we define the derivative d_0 as the difference quotient $\frac{v_1 - v_0}{t_1 - t_0}$.

Definition 2.1.2 (Finite, Piecewise Linear and Continuous)
A signal s, with n_s checkpoints, is finite, piecewise-linear and continuous (f.p.l.c.) if a sequence t_i ($i \in \{0, 1, ..., n_s\}$ and $\forall t_i : t_i \in \mathbb{T}$) exists such that:

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

We call the sequence t_i the *time sequence* of the signal s.

From the derivative defined in Definition 2.1.1, it follows that $\forall t' \in [t_i, t_{i+1}] : s(t') = v_i + d_i * (t' - t_i)$. For $t' = t_i$, $t' - t_i = 0$, so this trivially equals v_i as expected.

For $t' = t_{i+1}$, we get $s(t') = v_i + d_i * (t_{i+1} - t_i)$. After expanding d_i to $\frac{v_{i+1} - v_i}{t_{i+1} - t_i}$, the multiplication and the denominator cancel. The subtraction and addition of v_i also cancel, simply leaving us with $s(t') = v_{i+1}$ as expected.

Finally, assuming the last checkpoint in the signal is $\{t_n, v_n, d_n\}$, we state that d_n is undefined. Since the signal is not defined after this point, defining a value for the derivative at this point is

nonsensical.

We assume signals are finite, piecewise linear and continuous (Definition 2.1.2). An STL formula can operate on multiple input signals, the entire set of input signals is named a *trace* (Definition 2.1.3).

Since a signal is a function mapping $\mathbb{T} \to \mathbb{R}$, we can view a trace as a function mapping $\mathbb{T} \to \mathbb{R}^n$ (where $n \in \mathbb{N}$ is the amount of signals in the trace). However, this isn't quite complete.

For some signals s_0 and s_1 in the input trace w, there may exist some $t \in s_0$ that is outside the domain of s_1. As a result, we cannot assume that signals are all defined over the entire input domain represented in a trace. To represent a trace as a function, then, we must extend $\mathbb{R}$ with a value to represent undefined signals as in Definition 2.1.3.

Definition 2.1.3 (Trace)
A trace is defined as a set of real-valued signals $w = \{s_1, s_2, ..., s_n\}$ $(n \in \mathbb{N})$.

Equivalently, we could add an indicator value for undefined values to $\mathbb{R}$ to construct $\mathbb{R}_u$. Then, a trace can be viewed as a function $w : \mathbb{T} \to (\mathbb{R}_u)^n$.

Further, we introduce a specific notation for signals with a constant value, defined over all of $\mathbb{T}$, in Definition 2.1.4.

Definition 2.1.4 (Constant Signal)
We define a constant signal s as the constant function $s(t) = v, \forall t \in \mathbb{T}$. We denote constant signals using their value as $\overline{\mathbf{v}}$.

2.2 Specification

We allow signals to be modified using arithmetic operators, the other operand can either be another input signal or a constant value (in which case it is part of the formula). We define the grammar for these operations in Equation 2.1. We write abs(s) for the absolute value of a signal in this definition for clarity, since the symbol used in the application is the same as the separator for the grammar rules. An absolute value in an STL formula should actually look like $|s|$ instead.

$$\begin{aligned} \psi = & s_{\text{in}} \in w | \overline{\mathbf{c}} | (s) | s_0 + s_1 | s_0 - s_1 | s_0 * s_1 | s_0 / s_1 | \text{abs}(s) \\ & \text{where } c \in \mathbb{R} \end{aligned} \tag{2.1}$$

The arithmetic operations are defined point-wise, following the creation of *comparable* signals (Definition 2.2.1).

Definition 2.2.1 (Comparable Signals)
Two signals s_0 and s_1, with time series t_i^0 and t_i^1 respectively, are considered comparable if their time series are identical.

For any two signals, we can compute comparable signals by limiting their domain to the intersection of their domains and computing interpolated (extra) sample points where time stamps do not match within the intersected domain. Exact implementation of this operation is discussed in Section 4.1.

We additionally define Boolean threshold formulations that allow a quantitative signal to be transformed into a Boolean signal following an arbitrary specification in Equation 2.2. Mixing the Boolean and quantitative semantics of the application is not recommended.

$$\psi = s_0 > s_1 | s_0 \geq s_1 | s_0 < s_1 | s_0 \leq s_1 | s_0 = s_1 | s_0 \neq s_1 \tag{2.2}$$

The grammar for the STL operators making up a formula is (minimally) stated in Equation 2.3. The other operations that make up STL are defined in terms of the ones in Equation 2.3 later.

$$\begin{aligned} \varphi :=& \neg\varphi | \varphi_0 \wedge \varphi_1 | \varphi_0 \mathbf{U}_\mathrm{I} \varphi_1 \\ & \text{where } \mathrm{I} = [a, b] \subset \mathbb{T} \text{ and } a \leq b \end{aligned} \tag{2.3}$$

Logical negation ($\neg$) and logical conjunction ($\wedge$) require no further explanation; they are exactly as in traditional Boolean logics (there are some computational differences - but we get to that later).

$\mathbf{U}_I$ is the *Until* operator; for the operation to be satisfied, its left-hand operand must remain *True* until, at some future point in time, its right-hand operand becomes *True* - once the right-hand operand has become *True*, there is no more requirement on the left-hand operand. The interval I is the relative time range over which the operator operates. If we denote the time for which the formula is being evaluated as t_0, then the until operation with interval $I = [a, b]$ will use exactly the values (from both signals) that fall within the interval $[t_0 + a, t_0 + b]$ in its computation.

From these basic elements, we define the other parts of the STL formalism (using s_1 and s_2 to denote any given input signals).

$$s_1 \vee s_2 := \neg(\neg s_1 \wedge \neg s_2) \tag{2.4}$$

$$s_1 \implies s_2 := \neg(s_1 \wedge \neg s_2) \tag{2.5}$$

$$\Diamond_I s_1 := \overline{\mathbf{True}} \mathbf{U}_I s_1 \tag{2.6}$$

$$\Box_I s_1 := \neg \Diamond_I \neg s_1 \tag{2.7}$$

- Equation 2.4 defines the *logical disjunction* (as in classical logics)
- Equation 2.5 defines *logical implication* (as in classical logics)
- Equation 2.6 defines the *eventually* operation (meaning, at some point in the interval, the operand will become true)
- Equation 2.7 defines the *always* operation (meaning that the operand must hold for the entirety of the associated interval)

Further, for all operators that have been defined over an interval, we also define an *untimed* version of the operator. This, implicitly, sets the interval $I = [0, +\infty)$. They are noted identically to the timed versions above, but the interval I will be omitted. This is a slight extension of previous work, where only certain operators had explicitly defined separate timed and untimed variations [18] [7].

2.3 Robustness

The *robustness* is a real value describing the distance from the boundary condition for the current state of the variables is. It varies with the current values of the variables and so is, in effect, itself a signal. It is similar to the *satisfaction* for a Boolean formula. In case only small changes are required to change the formula's satisfaction, we expect a robustness that is (relatively) low in magnitude. Conversely, if large changes are required, a high magnitude robustness is expected. A negative robustness value indicates that the current values do not satisfy the formula, a positive robustness indicates that the formula is satisfied. In this way, the sign of the computed robustness value can be used to determine if a formula was satisfied by a certain input or not. The interpretation of a robustness that is exactly 0 is implementation defined.

The robustness of an STL formula at a given time essentially states how far from the boundary condition the current state of the variables is. We define the robustness (ρ) of an STL formula, given a specific trace (w = { s_1, ..., s_n }, $n \in \mathbb{N}$), a set of times (t) and a logical formula (φ) for the basic elements of the grammar defined above. For all derived operators, the robustness is computed using these elementary parts. The decomposition of the derived operators has been described in Section 2.2.

The definition of the robustness computation, for any input signals s_0 and s_1, for the basic building block operators can be found in Equation 2.8 through Equation 2.13. For the remaining operators, the robustness computation can be derived directly from their deconstruction.

$$\rho(True, w, t) := True \tag{2.8}$$

$$\rho(c, w, t) := c \quad (c \in \mathbb{R}) \tag{2.9}$$

$$\rho(\neg s_0, w, t) := \neg\rho(s_0, w, t) \tag{2.10}$$

$$\rho(s \geq 0, w, t) := s(t) \quad (s \in w) \tag{2.11}$$

$$\rho(s_0 \wedge s_1, w, t) := \min[\rho(s_0, w, t), \rho(s_1, w, t)] \tag{2.12}$$

$$\rho(s_0 \mathbf{U}_I s_1, w, t) := \sup_{t' \in t+I} (\min[\rho(s_1, w, t'), \inf_{t'' \in [t,t']} [\rho(s_0, w, t'')]]) \tag{2.13}$$

While in theory the robustness value can be any value in $\mathbb{R}$, in practice the value is often limited by the combination of the input signal(s) and the STL formula. We discuss this further in Chapter 5, where the range of the input signals as well as the STL formula are manipulated to ensure the computed robustness is limited to certain interval. Whenever we refer to a high or low robustness anywhere in this document, we mean relatively high or low within this situation-dependent interval of possible values.

2.3.1 Complexity

The computation of the robustness for an entire STL formula can be represented as a tree; operators can be nested, so the output of one operator may need to be interpreted as the input of another. Each operator in the formula will only be evaluated once, but we do need to ensure that at every step of the way, both the time to compute an operation is linear in the size of the input as well as the output being linear in the size of the input.

If an operator were to introduce a non-linearity in the size of its output, then any time this

operator was nested, the overall formula time complexity would be non-linear in the size of the original input signals. To prove that the complexity of the STL robustness computation is linear in the size of the input signals, we need to show that the computation of each individual operation is linear and that the size of the output resulting from those computations is also linear.

Below, we show that while the efficient algorithms introduced in *Efficient Robust Monitoring for STL* achieve linear complexity in offline contexts, that does not carry over to online contexts [7].

2.3.1.1 Offline context

The most complex operations in STL have, using the most efficient known algorithms, a linear complexity in the length of the signal (not significantly influenced by the size of the time interval) [7] in offline contexts. Using the naive, syntax-derived implementation, the complexity is further influenced by the size of the time intervals in the formula.

Intuitively, a minimal exact specification for a piecewise linear signal has sample points for exactly those places in the signal where the slope changes. All other points on the signal can, following our assumptions, be interpolated based on that data.

Changes in slope in an STL operation's result may occur in different (and additional) places than in the input signals. We can determine which points those are ahead of time, though: it is exactly the set of points present in the input signals combined with the set of intersections between them, potentially shifted in time.

Consider two signals s_0 and s_1 (without loss of generality, assume the amount of checkpoints in s_0, denoted n_{s_0} to be larger than the amount of checkpoints in s_1). Further, consider the set of times defined by the union of the time series of s_0 and s_1 (denoted T, sorted in ascending order).

We know that T has at most $2 * n_{s_0}$ elements (since the sum of elements from both signals must be smaller than or equal to that value). We also know that between any two sequential elements of T, both signals must be linear (this follows from the f.p.l.c.assumption). Trivially, then, between any two sequential times in T, at most one intersection between s_0 and s_1 can exist (since co-linearity is not counted as intersection). This means that there can be at most $2 * (n_{s_0} - 1)$ intersections between s_0 and s_1 which immediately imposes an upper limit on the size of the output signal for any STL operation that is linear in the size of the input signal.

From these observations, we can conclude that the evaluation of the robustness for an entire STL formula is linear in the size of the input signal.

2.3.1.2 Online context

Unfortunately, the complexity analysis from the previous section does not carry over to online contexts. The issue is that, in an online context, the entire signal is not available when the algorithm is executed. Instead, a new data point comes in and then the algorithm must compute the new robustness value.

To optimise the evaluation as much as possible, we can limit the size of the input signal. How much, though, is not immediately obvious. The largest interval in the formula seems like a good first candidate. However, consider the formula $\Diamond_{[30,50]} s \vee \Diamond_{[0,5]} s$. If we simply limit to that interval, we would limit the signal in such a way that the computation of the second eventually

is impossible. Instead, the limitation is the interval defined by the minimal lower bound of time intervals in the formula and the maximal upper bound of time intervals in the formula. In this case, that would be the interval $[0, 50]$. We'll call this the formula's *combined time interval*.

Since each step of the computation requires an input signal size defined by the formula's combined time interval, the size of that interval clearly influences the computation's complexity. Another factor in the complexity, trivially, is the size of the overall signal (defined by measurement frequency and runtime of the monitored system). The amount of sample points directly determines the amount of times the computation must be repeated. Trivially, then, the overall complexity (for a constant formula) is $\mathcal{O}(I * s)$ where I is the combined time interval of the formula and s the total amount of measurements from the monitored system.

3 STL Tool Refactor

Our work expands on a previously written tool [18]. As part of our work, we address a lot of the technical debt that accumulated during the development of this tool (through a thorough refactor of the code base) as well as correcting some inconsistencies and errors in the behavior. Further, we introduce a set of tests to the tool to allow automated verification of its correctness rather than relying on the manual test cases used previously [18]. Our version of the implementation is available on GitHub [37].

We additionally use a second repository which contains the original implementation and our implementation as submodules [38]. The second repository is mostly used to manage files relating to comparisons between the two implementations (for example, scripts to compare performances). This second repository is also later used for code that is not directly related to the STL tool implementation.

3.1 Existing Implementation

Use of the tool is relatively simple (and remains largely unchanged in our implementation); one provides the necessary input files and then runs a Python script (passing in the files as parameters). Two input files are required: the formula to be considered and the signals it operates on. Further, optionally, semantics can be specified using a parameter. These are either quantitative (default) or Boolean. Given this data, the tool computes the resulting robustness signal and displays it.

For the most part, the input formats defined in the original implementation are preserved [18]. We add the option to specify untimed operators in the STL formula through the omission of the time interval. The STL formulas that are accepted by our implementation are thus a strict superset of the original implementation, which means compatibility is maintained.

The tool uses an ANother Tool for Language Recognition (ANTLR) [39] grammar to build an Abstract Syntax Tree (AST) representing the formula semantics. From this structure, it is relatively simple to compute the robustness: the entire tree's robustness is equal to the robustness computation of the root node applied to the robustness signals of its children. This property applies to every sub-tree as well.

The robustness computation for the leaves is trivial: the leaves are always the input signals (or constants), their robustness is defined by the identity function (see Equation 2.9). Using the results from the leaves, then, we can compute the robustness signals for every node in the tree in a bottom-up manner.

By recursing the tree, we can compute the robustness for the formula. The robustness signal for the leaves is trivial: the leaves are always the input signals (or constants), their robustness is exactly the identity function (see Equation 2.9).

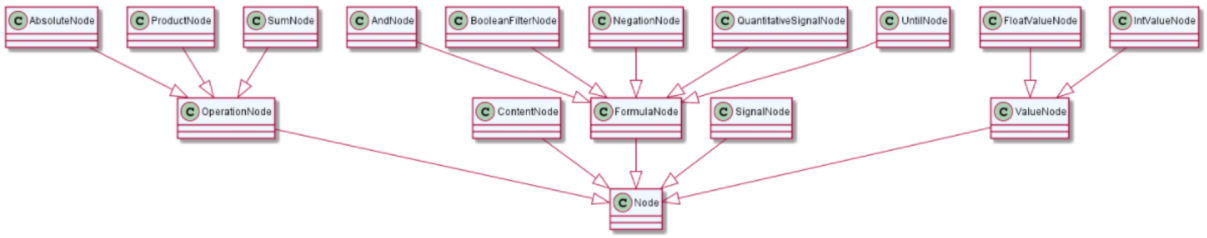

Figure 3.1: Tree node class structure for the pre-existing implementation

```
.gitattributes
.gitignore
antlr-4.8-complete.jar
customStlListener.py
formula.stl
README.md
stl.g4
stl.interp
stl.py
stl.tokens
STLLexer.interp
STLLexer.py
STLLexer.tokens
stlListener.py
stlParser.py
stlTree.dot
stlTree.py
stlUtils.py
stlVisitor.py

├──extra_scripts
    eventually.py
    plotter.py
    sinGenerator.py
    until_bool_steps.py

├──gen
    stl.interp
    stl.tokens
    stlBaseListener.java
    stlBaseVisitor.java
    stlLexer.interp
    stlLexer.java
    stlLexer.tokens
    stlListener.java
    stlParser.java
    stlVisitor.java
```

Figure 3.2: Folder and file structure of the pre-existing implementation

3.1.1 Architecture

In this section, we provide a brief overview of the architecture of the pre-existing implementation [18]. We focus specifically on those aspects of the implementation that seemed problematic and that we have addressed in our work.

Figure 3.1 shows the tree nodes used in the pre-existing implementation. Each of these nodes has a *validate* function, which is used to compute the robustness (and each node calls the *validate* function of its children as required). For every `Node` class, this function contains the implementation of the robustness computation for the associated operation. The base node class provides most of the data that each node needs - any similar operations get an intermediate class to inherit from that implements the operation similarities (e.g. for all binary operations).

Overall, the structure of the project files is rather flat as we can see in Figure 3.2. While that works while the project is small (which in this case, it is), it is bound to create issues as the implementation is expanded.

```
220     for i in range(i_1, len(s1[0])):
221         temp_1[0].append(s1[0][i])
222         temp_1[1].append(s1[1][i])
223         temp_1[2].append(s1[2][i])
224         temp_2[0].append(s1[0][i])
225         temp_2[1].append(temp_2[1][-1] + (temp_2[0][-1] - temp_2[0][-2]) * temp_2[2][-1])
226         temp_2[2].append(temp_2[2][-1])
227     for i in range(i_2, len(s2[0])):
228         temp_1[0].append(s2[0][i])
229         temp_1[1].append(temp_1[1][-1] + (temp_1[0][-1] - temp_1[0][-2]) * temp_1[2][-1])
230         temp_1[2].append(temp_1[2][-1])
231         temp_2[0].append(s2[0][i])
232         temp_2[1].append(s2[1][i])
233         temp_2[2].append(s2[2][i])
```

Figure 3.3: Hard-to-read code example in pre-existing implementation (stlUtils.py:220-233 [40])

The tree nodes are all implemented in the same file (*stlTree.py*), which makes navigating the code a little difficult, especially since the nodes contain the implementation of their validation functions. Over the entire implementation, multiple functions are also much too long to be read and understood. The main offender is the implementation of `UntilNode`.*validate*, which is 254 lines of code (LoC).

We note the lack of implementation of any structure to represent the signal data used by the system; they are represented as a list of lists. While that is technically sufficient (and clearly inspired by the mathematical descriptions), it creates code that quickly becomes unreadable (especially when other lists are also used). Combined with the lack of descriptive naming that often occurs in the pre-existing code base, this creates sections of code that are incredibly hard to understand when one is not already aware of what the code does. An example is shown in Figure 3.3.

3.2 Refactor

In this section, we briefly go over each of the issues we noted in the pre-existing implementation and describe our solution to the issue. These are behavior conserving changes that simply aim to improve the readability, maintainability and extensibility of the code.

3.2.1 Architecture

The overall structure of the project is unchanged. An ANTLR grammar is still used to generate an AST, which is the traversed to compute the robustness for a given set of input parameters. Slight modifications have been made where we believed the result to be clearer.

When we compare Figure 3.4 to Figure 3.1, it is quite clear that structurally that part of the program has remained largely unchanged. A few classes have been renamed (most notably, `BooleanFilterNode` to `ComparisonOperatorNode` which far more clearly reflects its function). Other than simple name changes, there is a single structural change.

We have entirely removed the `SumNode` and `ProductNode`. Instead, a single node class (named `BinaryOperationNode`, implementation similar to `ComparisonOperatorNode`, but for different operators), now represents all supported binary mathematical operations (sum, difference,

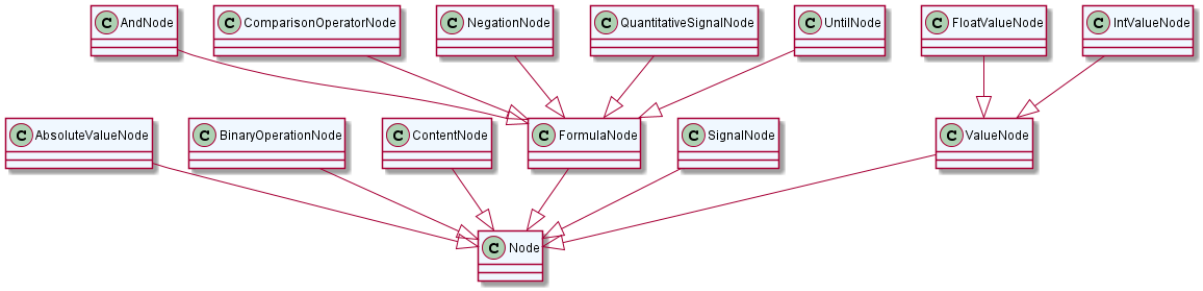

Figure 3.4: Structure of the AST tree nodes in the reworked code

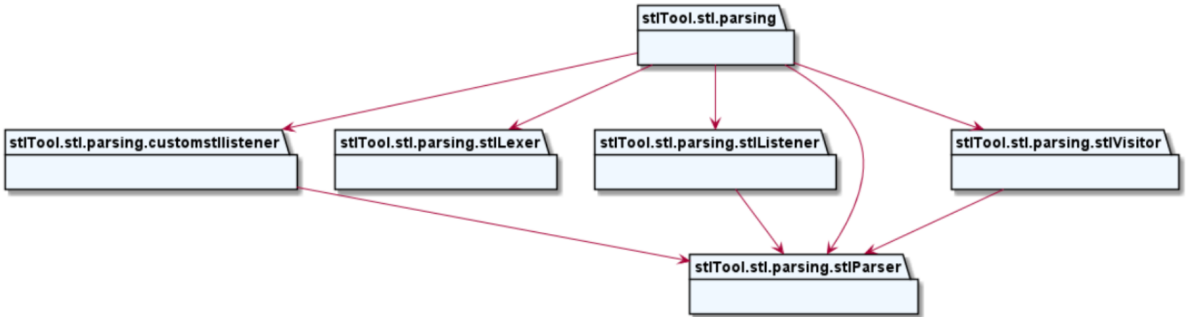

Figure 3.5: Structure of the STL parsing package after the refactoring

multiplication, division). The `AbsoluteNode` has been renamed to `AbsoluteValueNode` and is the only unary mathematical operation. In case more of those had been implemented, an additional `UnaryOperationNode` may have been introduced.

3.2.1.1 Parsing

The architecture of the parsing segment of the project is largely determined by the ANTLR framework [39]. Other than moving the relevant files into a separate package (see Section 3.2.3), no structural modifications have been made. The structure seen in Figure 3.5, thus, is essentially identical between the previous implementation and our work [18].

3.2.2 Componentization

3.2.2.1 Signal representation

This issue was the most notable when first attempting to understand the tool's implementation. Rather than use a data structure specifically to represent signals, a generic list structure was used (containing 3 other lists; times, values and derivatives). While this directly coincides with the mathematical definition for the signal, it is not easy to work with. Or, rather, when working with that representation extensively, it becomes hard to understand what is happening when reading the code.

Whenever a specific time, value or derivative needs to be accessed, two square-bracket operators are required. When these accesses are mixed with other lists present in the functions, things

```
10    for i in range(i_1, s1.getCheckpointCount()):
11      temp_1.addCheckpoint(s1.getCheckpoint(i))
12      temp_2.emplaceCheckpoint(s1.getTime(i), temp_2.computeInterpolatedValue(s1.getTime(i)))
13    for i in range(i_2, s2.getCheckpointCount()):
14      temp_1.emplaceCheckpoint(s1.getTime(i), s2.computeInterpolatedValue(s2.getTime(i)))
15      temp_2.addCheckpoint(s2.getCheckpoint(i))
```

Figure 3.6: Translation of the hard to read code example seen in Figure 3.3

quickly become confusing.

We mitigated this problem through the implementation of a `Signal` class, which now encompasses the signal representation as well as all representation-related functions. The `Signal` class manages a sorted (ascending, by time) list of `SignalValue` objects and provides an extensive interface for interacting with the data. This interface encompasses all operations required for the implementation of the STL operations and the clear method names ensure that any functions using this class are far more readable than they would have been using the old representation.

A `SignalValue` is a small wrapper around a triple: {time, value, derivative}. The main benefit it offers is the improved readability of getters and setters for each of the three values over the square-bracket access that would otherwise be needed. In this implementation, we assume that signal timestamps are always larger than or equal to zero. Timestamps can be pre-processed such that that assumption is not violated in general.

To illustrate the impact this has on the code readability, we have modified the previous bad example (Figure 3.3) to use the `Signal` class we implemented. Figure 3.6 shows the modifications to the code that can be made to support this operation using the `Signal` class. The variable names have been left unmodified to keep the comparison between the examples obvious. In our implementation, the variables would have been renamed for clarity if necessary. In this case, the segment was the cause of an error in behavior and so it has been eliminated entirely.

Signals must support the two semantics the tool uses: Boolean and quantitative. Rather than attempting to encompass both semantics within one class, we make the observation that Boolean semantics are, in essence, a special case of the quantitative semantics. We define a derived class `BooleanSignal` that implements any peculiarities in the Boolean behavior (for example, not using derivatives or intersection computations). All other behavior is inherited.

In the `Node` validation functions, a list of `Signal`s must be passed as an input parameter. We have implemented a small wrapper around `List` as well to manage these. Essentially, `SignalList` only adds an easy way to read in a list of `Signal`s from the input file the program expects.

The full structure of the stl.signals package, then, is shown in Figure 3.7.

3.2.2.2 STL operations

The STL tool placed the implementation of all STL operations in the AST `Node` classes (which were grouped together in *stlTree.py*). Since these implementations can be quite lengthy, this complicated these rather simple tree node classes unnecessarily. We have moved the implementation of the STL operations into their own module, from which the appropriate functions can be imported when needed.

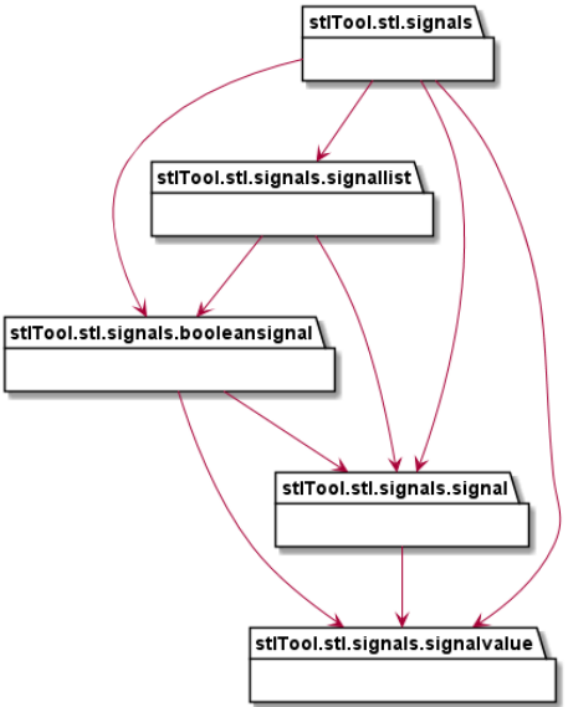

Figure 3.7: Structure between the various signal-related classes

This method extraction dramatically reduces the complexity of the tree nodes and removes any dependencies between them (outside of the inheritance relationships). Previously, these dependencies existed because we defined some STL operations as a sequence of other STL operations (Equation 2.4 through Equation 2.7). The operator inter-dependency is visualized in Figure 3.8.

While the actual dependencies between the operations remain unchanged (obviously, since their definitions were not changed), we do remove dependencies between `Node` classes that were the result of these operation dependencies. Additionally, any cases where operations might have been implemented multiple times were immediately corrected.

3.2.3 Project Restructuring

As seen previously (Figure 3.2), the project had very little structure. Essentially, it was a single folder containing all the code files. While that works (and is quite easy) as long as the project is small, the modifications we were making quickly resulted in that structure becoming unmaintainable. At that point, we decided to restructure the project entirely and introduce a hierarchy. The result of this can be found in Figure 3.9.

Some of these folders did not become necessary until some of the other refactoring operations were completed (for example, the *stl/operators* folder is a result of the changes discussed in Section 3.2.2.2).

The introduction of the hierarchy significantly simplified navigating the code when manually searching for a single file. Each folder contains the *___init___.py* file required to turn the folder into a Python module to manage importing all the correct files.

Figure 3.8: Python module dependencies within the operations package showing operator interdependency

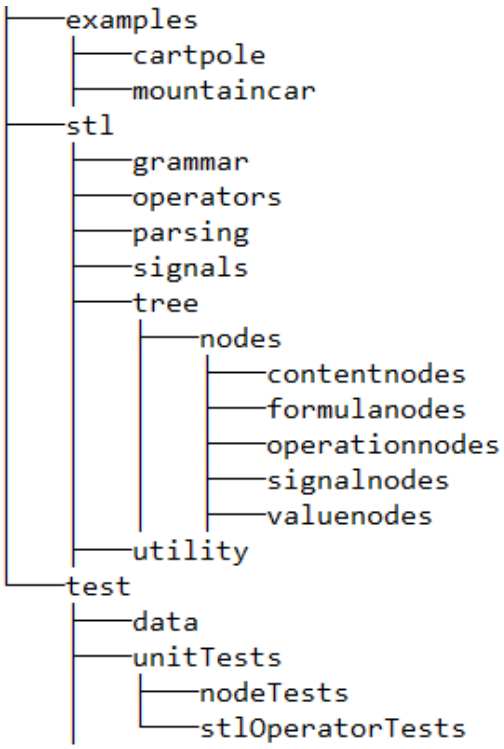

Figure 3.9: Folder structure of the modified implementation (files omitted for brevity)

3.2.4 Readability improvements

3.2.4.1 Remove code duplication

Some small tasks (such as value interpolation for signals), shared between multiple STL operations, were implemented multiple times (sometimes with slight differences, when one had encountered a bug and been corrected - but the other had not, yet). In our implementation, we extracted these into their own functions and ensured that code duplication was minimal.

For a lot of these, they became methods on the `Signal` class because they are rather innately related to the representation of a signal (value interpolation definitely falls under that, now being accessible through the *computeInterpolatedValue* method).

3.2.4.2 Type hinting

While specifying variable types in Python is not a language requirement, it does make reading and understanding the code significantly easier. It also improves analytic functions of most editors. We have added type hints throughout the code base, which (when combined with the introduction of the new data structures) makes for far more readable and easy to follow functions.

The availability of doc-strings on hover, for example, is a big improvement. Rather than having to manually figure out what type a variable is, then find the method in question to read its documentation, now we can simply place the cursor over the function we are trying to explore and our editor will display the information we need (Visual Studio Code in our case, though most editors should support this).

```
238 def getBooleanIntersection(a, b):
239     intersection =  [max(a[0], b[0]), min([a[-1], b[-1]])]
240     if intersection[0] > intersection[1]:
241         return False
242     else:
243         return intersection
```

Figure 3.10: Implementation of *getBooleanIntersection* in pre-existing codebase

While there were no cases in the pre-existing code where types were actually passed incorrectly, we did find some cases where the parameter types did not match what one would expect when reading the method name. We specifically refer to the function *getBooleanIntersection*, pre-existing implementation included in Figure 3.10.

If this function is encountered after having first seen the quantitative semantics (and the *getPunctualIntersection* implementation), one might assume this method fulfills a similar function for the Boolean semantics (i.e. limiting the signals to the same interval and then ensuring both have the same set of timestamps, but not doing any intersection computations). That, unfortunately, isn't the case.

The implementation, shown in Figure 3.10, shows that that is not the case. Instead, it turns out the function is rather badly named. As input parameters, it takes two pairs of floats, each specifying a single interval in time. The function, then, computes the intersection between the two intervals specified by the floating point pairs. This particular issue was corrected through our introduction of an explicit `Interval` class, rather than using generic structures like pairs of floats. The function was renamed (*computeIntersection*) and implemented as a method of the class.

3.2.4.3 Variable naming

In the old code, many variables used names that either a) mirrored naming from a particular paper or b) used names that conveyed no information at all (e.g. *temp*). Both categories are bad practice - the former only conveys useful information when the reader is simultaneously looking at that particular paper, the latter never conveys any useful information at all.

For both categories of poor naming, we renamed the variables to something more meaningful. Generally, this required a brief analysis of the surrounding code to fully understand what was going on. Considering the state of the code and its poor readability in its initial form, this was often non-trivial.

As a minor improvement, we have also renamed some functions and classes that did not adequately convey what they were doing. For example, the class `BooleanFilterNode` has been renamed to `ComparisonOperatorNode` (since it was the node implementing all the comparison operations). The result of the comparison operations is a Boolean value, of course, so the node's evaluation did function as a Boolean filter in that sense, but the name is slightly misleading. One might expect a `BooleanFilterNode` to simply apply to one signal and booleanize its values, rather than implement a comparison between two signals.

3.2.4.4 Separation of validation functions

The tool supports validating formulas in both Boolean and quantitative semantics. In the original implementation, both of these were handled in the same method, often with conditionals placed at multiple points in the functions to address the differences in semantics. This dramatically increased the complexity of the implementation, some featuring multiple checks on the semantics within a single function to differentiate the behavior between the two contexts.

To address this, we entirely separated the two semantics. Previously, each node would have to check a parameter to determine what semantic it was supposed to be evaluating. We eliminated almost all of these; we preserved the check in the `ContentNode`. `ContentNode` is a node class for the root of the AST — it cannot appear in any other place and is the root for all ASTs our implementation creates. By doing this, we maintain external compatibility.

The `ContentNode` is now the only class that checks the value of the semantics parameter, its *validate* method calls either *quantitativeValidate* or *booleanValidate* on its children, depending on its value. Further down the tree, the children will simply call the same function: a node's *quantitativeValidate* method only calls the *quantitativeValidate* method of its children and the same is true for *booleanValidate*. In this way, all nodes on the tree are able to operate under the correct semantics.

3.3 Testing

We have implemented a testing suite for the STL operations to ensure they are performing as expected, using Python's `unittest` package. Since the old implementation had errors, this seemed like the obvious choice when attempting to identify and correct all behavioral problems (as well as ensuring that we do not create any regression errors during further modifications). We introduced tests for each operation separately. These cover all the bugs we encountered over the course of this work, as well as a number of other cases that came to mind during development.

3.3.1 Structure

The tests are implemented as their own Python module. A file to run all of them is provided in the repository (*test.py*).

The test suite is split based on what is being tested. We have some tests aimed at the AST nodes and some aimed at the STL operations. Within the former, there is a file for each node we test. In the latter, there is a file for each operation being tested. These files define a class (derived from `unittest.TestCase`), holding a variable number of test cases (and sometimes, helper functions). When *unittest*'s main function is invoked, all test cases that have been imported (in case of the provided *test.py* file, all of them) are executed and the results are printed.

3.3.2 Coverage

Overall, we implemented a little over 100 test cases. The coverage for these tests is, on the STL operations package, 100%. For the supporting code (`Signal` class, AST nodes, ...), the coverage is not quite as comprehensive - coverage overall is around 80%. Most of the methods not being tested are relatively simple methods, like those handling file reads (for example, correctly setting

the operation for the `ComparisonOperatorNode` based on the token encountered in the input formula or the `Signal` initialization from file performed in `SignalList`).

3.4 Performance

We analyze the performance of our implementation in offline contexts, compared to the pre-existing implementation, both in runtime and memory use. Intuitively, we would expect our implementation to use slightly more memory as extra data is stored.

For example, we maintain both a list of `SignalValue`s to define the signals in our implementation as well as a separate list of times. This time series storage was added after profiling showed that constructing the list of times used a significant portion of the algorithm's runtime.

Due to the optimisations we have applied to our implementation, we hope to see an improvement in the runtime of our program when compared to the pre-existing implementation, even when comparing the syntax algorithm between both.

Comparing the efficient algorithm, we expect a significant further improvement, especially in formulas that have large window sizes (since this is a factor in the syntax algorithm's complexity, but not in the efficient algorithm's [7]).

All performance statistics were gathered on a Ubuntu 20.04 LTS system, using an AMD Ryzen 5900x CPU running at stock configuration. Throughout testing, lightweight monitoring software was used to ensure no unexpected CPU down-clocking occurred. No applications likely to result in a measurable impact on performance were running on the system during these measurements.

The full implementation of the performance comparison script is available in our comparison repository [38] as the file *comparePerformance.py*. To reproduce the full results presented in this section, the formula in each repository's *formula.stl* file must be specified appropriately. Additionally, in the new version of the tool, the algorithm type must be specified as a command line parameter. Exact description can be found in the project README on GitHub [37].

For all analysis in this section, where not explicitly specified, we are using Formula 3.1 for the robustness computation we are analyzing in the pre-existing implementation. This formula is, to be compatible with the changes mentioned in Section 4.1, slightly modified when using the current implementation to Formula 3.2. Any non-specified analysis of the current implementation is using Formula 3.2. This formula was presented in the previous work [18] as an example on a cartpole data set. We use that same data set, which can be found in the original repository [40] in the file *signals/results/angles_ac_cart-pole.csv*.

Using these two formulas, the two implementations obtain an identical result (save extra sample points in the current implementation, as described in Section 4.2). These extra sample points do not meaningfully influence the result signal, since the result signal in this case is a constant value.

$$\square_{[150,2950]}((12 - |e5|) \wedge ((|e5| - 4.5) \implies \Diamond_{[0,150]}\square_{[0,30]}(4.5 - |e5|))) \tag{3.1}$$

$$\square_{[0,2770]}((12 - |e5|) \wedge ((|e5| - 4.5) \implies \Diamond_{[0,150]}\square_{[0,30]}(4.5 - |e5|))) \tag{3.2}$$

	Pre-existing Syntax	Current Syntax	Current Efficient
Runtime (s)	520.5	130.0	1.5

Table 3.1: Comparison of the runtimes between algorithms, rounded to the nearest half-second

Further, some segments in the *comparePerformance.py* file are aimed purely at runtime performance, others at memory use. Which are aimed at runtime measurement and which at memory use measurement is indicated by in-file comments; it is important for the (exact) reproduction of these results that only the correct segments (that is, only memory use when recording memory use, only performance when recording performance) are uncommented.

3.4.1 Runtime

We record the runtime of the process by busy-waiting until it terminates, in the *comparePerformance.py* file (the timing segments should be commented when using the memory recording and vice-versa).

Using the experiment setup described previously (Section 3.4), we obtain the results shown in Table 3.1. We see a roughly factor 4 improvement from the pre-existing implementation of the syntax algorithm to our version of the syntax algorithm. For the efficient algorithm, then, we see a massive improvement (over 85 times faster). However, for the formula in question, this is somewhat expected: there is a rather large interval given for the first operator of the formula.

Since the complexity of the syntax algorithm is dependent on the size of the interval of the operators (and this is not the case in the efficient algorithm), a massive performance boost is expected. This case shows that the implementation is now suitable for computations where the time intervals required are rather large.

To perform a more sensible comparison, though, we limit the size of the intervals (and simplify the formula). For both implementations, we now consider the inner-most formula with a reduced interval of $I = [0, 1]$ as in Formula 3.3.

$$\Box_{[0,1]}(4.5 - |e5|) \tag{3.3}$$

Reducing the interval should, as much as possible, limit the impact of the size of the interval on the outcome of the computation. Intuitively, when the interval is smaller, we expect a much faster solution from the syntax algorithm implementations, while the solution found by the efficient algorithm should be found in approximately the same amount of time. For the efficient algorithm in case of Formula 3.3 compared to Formula 3.1 (or Formula 3.2), we do expect a speed-up simply because there are fewer operators to compute. However, that difference should be far less significant than the difference in performance for each of the syntax algorithms (especially since they, too, compute fewer operators).

As seen in Table 3.2, we see the expected massive speed-up of both versions of the syntax algorithm. The difference between the old and new implementations of the syntax algorithm, while far less significant, is still present - the current implementation appears to perform better even at lower window sizes.

The efficient algorithm has seen a minor performance increase, as expected.

	Pre-existing Syntax	Current Syntax	Current Efficient
Runtime (s)	1.5	1.1	0.9

Table 3.2: Comparison of the runtimes between algorithms, rounded to the nearest tenth of a second

	Pre-existing Syntax	Current Syntax	Current Efficient
Runtime (s)	0.72	0.74	0.72

Table 3.3: Comparison of the runtimes between algorithms, rounded to the nearest hundredth of a second

The other factor on algorithm runtime performance is the size of the signal. To illustrate, we will repeat this latest case (using the reduced window size) with a limited data set. To achieve this, we artificially limit the size of the input to an interval $[0, 300]$ using a manual modification of the signal initialization methods. Since the original data set defines inputs over the interval $[0, 3000]$, we might expect something close to a factor 10 improvement for each of the algorithms.

However, when looking at runtimes this small, a significant portion of the runtime is not actually the computation of the algorithms. The expected improvement, thus, is significantly smaller.

It is clear from Table 3.3 that the performances obtained when we further limit the signal size is close to identical between each of the algorithms. In fact, there is barely any improvement in these times when we disable the robustness computation altogether (runtime at roughly 0.70 for the old implementation, 0.72 for the current version): most of the time is spent reading the input data, initializing the signals and AST representing the STL formula. In this aspect, the current implementation performs slightly worse. The construction of `Signal` instances is more complex than the initialization of the list-of-lists data structure that was used previously. However, in almost all cases, a significant performance benefit is obtained which makes this more than worth it.

Our initial setup considered the full process time since it was easier to record the statistics in conjunction with the memory statistics. Memory statistics are much easier to record for an entire process than for a specific function call, so we simply recorded both metrics for the entire process.

In the example considered in Table 3.3, a large portion of the time spent is not part of the robustness computation. The data thus isn't really representative of the actual operation's computation time. For the other examples, this was not as much of a problem: the computation time of the robustness was significant enough that clear differences in the amount of time needed were present. Since that is no longer the case, we adjust our analysis for a last data point. We add timing statements directly to *stl.py* and *main.py* (in pre-existing and current implementation, respectively) to record exactly the time taken by the computation. These are simply lines recording the time surrounding the call to *validate*. These are not explicitly stored in the repository, but manually adding them for reproduction of the result is trivial.

Other than the location where we record the time taken, the experiment setup is exactly the same as previously described to obtain the results in Table 3.3.

From the computation times noted in Table 3.4, it becomes clear that the current implementation of the syntax algorithm performs more slowly than the original in this context. However, the efficient algorithm's improvements manage to close the gap.

	Pre-existing Syntax	Current Syntax	Current Efficient
Runtime (s)	0.01	0.04	0.01

Table 3.4: Comparison of the runtimes between algorithms, rounded to the nearest hundredth of a second

From profiling, it is clear that the reason our new implementations take longer in these simple cases is that we consider constant signals to be signals rather than simply values. This means that our initialization cost is slightly higher (since we construct a `Signal` instance, rather than using the constant value). Additionally, when these values are used in operations, we call the *computeComparableSignals* method which the pre-existing implementation does not do for constants. These two factors entirely encompass the performance difference we see in Table 3.4.

Since the implementation achieves better performance in all but the simplest cases, though, we do not consider this a problem. We achieve significant performance gains in cases where computations take a long time. A minor performance degradation, where in both implementations the result is obtained near instantly, is not an issue.

3.4.2 Memory Use

We recorded the memory use of the process running every 0.5 seconds while the process was computing the robustness for Formula 3.1. The computation is performed using the *comparePerformance.py* file (the timing segments should be commented when using the memory recording and vice-versa).

We launch the computation of the robustness, for a given set of input signals, semantic and formula, using subprocesses (from the package `subprocess`). The implementation stores the memory use for the subprocesses (measured using Python's `psutil`). After computing the memory use for the syntax algorithm in both implementations, as well as the efficient algorithm in the updated implementation, we obtain Figure 3.11.

The keen observer may notice that in Figure 3.11, the black line for the efficient algorithm is present in the legend, but not on the figure. This is an unfortunate consequence of the massive difference in program runtimes. We include Figure 3.12 to show this fact, where a small segment from Figure 3.11 is enlarged such that the efficient algorithm's memory use becomes visible.

From these figures, we see that the results match our intuition: the new implementation uses more memory than the old. Considering the modifications we have made, this was to be expected. Noteworthy is a further, minor, increase in case of the efficient algorithm. Presumably, this occurs because the efficient algorithm creates extra data structures to optimize the program runtime.

Another noteworthy feature is the simultaneous decrease in memory use between the two syntax algorithm implementations. We assume this is the result of garbage collection, but cannot rule out other causes.

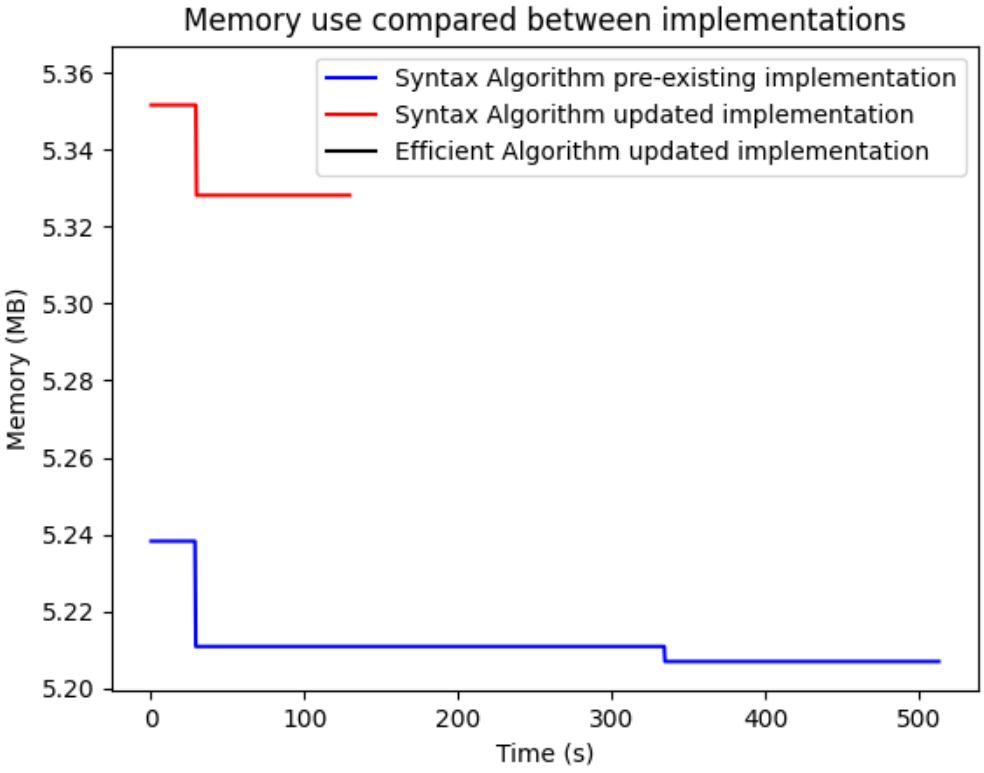

Figure 3.11: Comparison of the memory use between all working algorithms

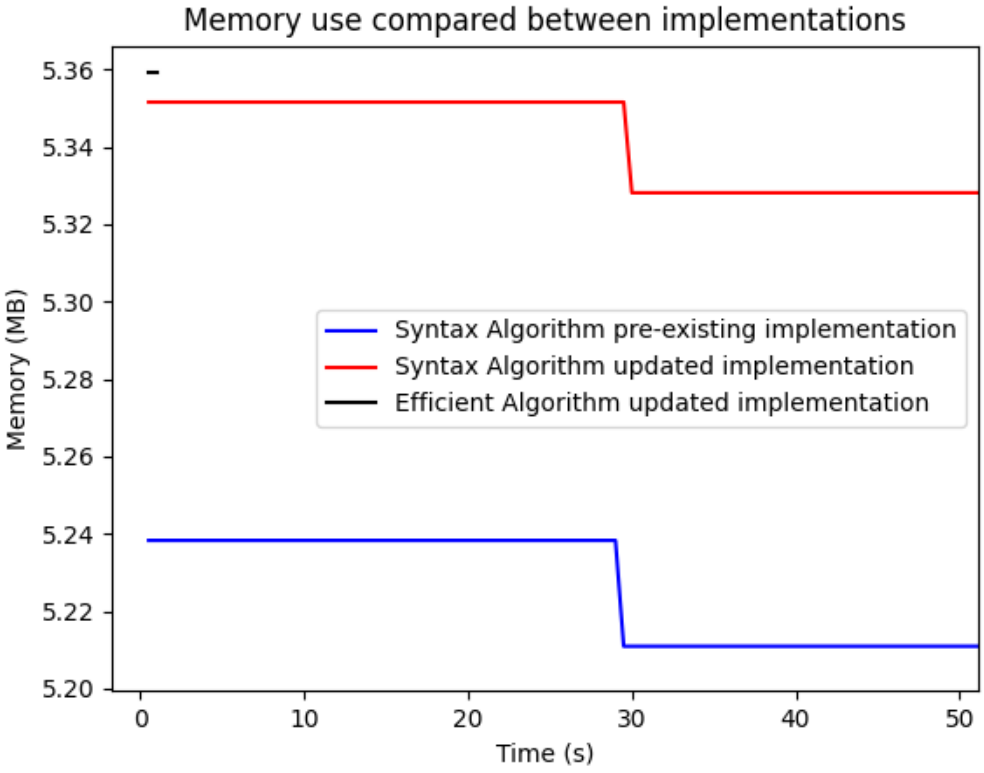

Figure 3.12: Comparison of the memory use between all working algorithms

4 STL Tool Functional Changes

This section addresses the changes made to the pre-existing implementation of the STL robustness computation tool [40] intended to modify the tool's behavior — generally in order to eliminate bugs or improve the performance. We detail the major improvements we have introduced and the corrections of the bugs noted in *Signal Temporal Logic Monitoring and Online Robustness Approximation* [18].

4.1 Computation of comparable signals

In the original implementation (presumably based on terminology in Efficient Robustness Monitoring for STL [7]), this was named *punctual intersection*. This does not quite capture what the operation actually does, though, so we have renamed it to the computation of *comparable signals*.

We briefly describe the operation: first, ensure that the time series for both signals is identical. For signals s_0 and s_1, denoting the set of times for any signal s as s_t, the set of timestamps that both must have is the set $\{s_{0_t} \cap s_{1_t}\}$. Second, find all points where the signals intersect and add that set of points to each signal (with the derivative equal to the derivative of the preceding checkpoint in that signal).

Next, we'll go into a bit more detail about how we actually achieve that implementation. We compute, for both signals, the interval on which they are defined. This can be trivially found through the time series; the lower bound of the interval is the first element of the time series, the upper bound is the last element. We then compute the intersection of this interval $I = [a, b]$. From both signals, we remove all checkpoints with a time value lower than a or larger than b.

Next, we ensure that in both signals a and b are part of the time series. If that is not the case, interpolated checkpoints must be added to fulfill that condition. This ensures that the amount of information lost is limited as much as possible.

Following the interval limitation, we iterate over the signals (one after the other). We'll describe the operation for iterating over s_0; the method when iterating over s_1 is identical, with the signals reversed. For each checkpoint $\{t_i, v_i, d_i\}$ in s_0, check if there is a checkpoint with time t_i in s_1. If there is not, add a new checkpoint (with interpolated value) to s_1. After repeating this operation with the signals reversed, the time series of s_0 and s_1 are identical.

This is not quite enough to encompass all possible places an STL operation's result signal might see changes in slope, though. This may additionally occur when the input signals intersect, which we have not yet accounted for. The last step, then, is to add these points as well by iterating over the signals and checking for any intersections. Whenever one is encountered, a new (interpolated) checkpoint is added to both signals.

In the pre-existing implementation, this method did not correctly limit the intersection to the interval over which both signals are defined. Rather than setting the interval of the resulting

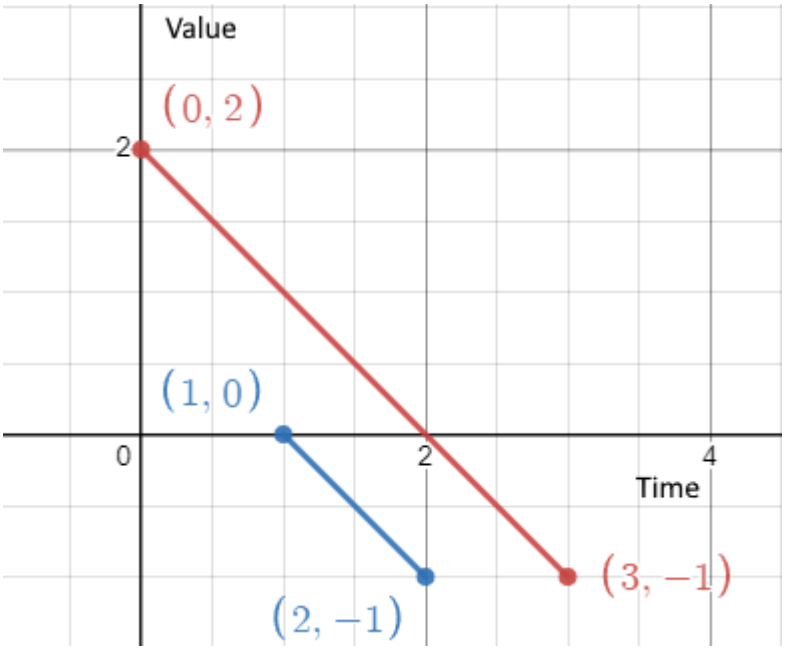

Figure 4.1: Example set up to illustrate the *computeComparableSignals* method

signals to $I_{lhs} \cap I_{rhs}$, they used $I_{lhs} \cup I_{rhs}$ (under the assumption that Signals are constant before their first timestamp and after their last).

This lengthens signals using segments of values that no information is available for.

Consider, as an example, the case of the two signals s_1 in red and s_2 in blue, shown in Figure 4.1. The domain of s_1 is $[0, 3]$ and the domain of s_2 is $[1, 2]$. To perform meaningful computation on these signals, the signals considered as operands for any operation should have their domain limited to the intersection of these two (which, in this case, happens to be equal to one of them: $[1, 2]$). The expected result is shown in Figure 4.2.

The original implementation did not behave correctly in this case. We assume the error in behavior is a result of a misinterpretation of the phrasing used in Efficient Robust Monitoring for STL [7]; regarding the definition of the *punctual intersection*, they state "We build the sequence $(r_i)_{i \leq n_z}$ containing the sampling points of y and y' when they are both defined, and the points where y and y' punctually intersect".

The pre-existing implementation appears to have misinterpreted that statement, considering the interval where *either* of the signals is defined rather than both them. This results in the interval $[0, 3]$ in our example, the incorrect result that the pre-existing implementation obtains is visualized in Figure 4.3.

4.2 Comparison operators

We use 0 to denote Boolean *False* value and 1 to denote *True*, as stated previously in Section 2.1. Since Boolean semantics are mixed with quantitative semantics in this section, we will explicitly state whenever a particular signal is using Boolean semantics within this section. Any time this is not stated, the semantics can be assumed to be quantitative.

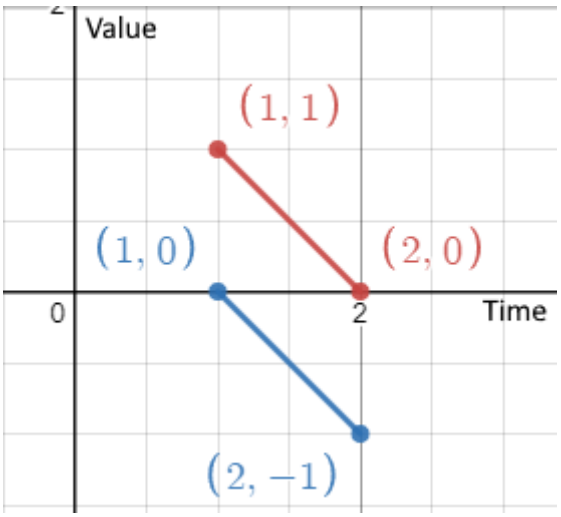

Figure 4.2: Correct computation for *computeComparableSignals* found in the current implementation (using the data in Figure 4.1

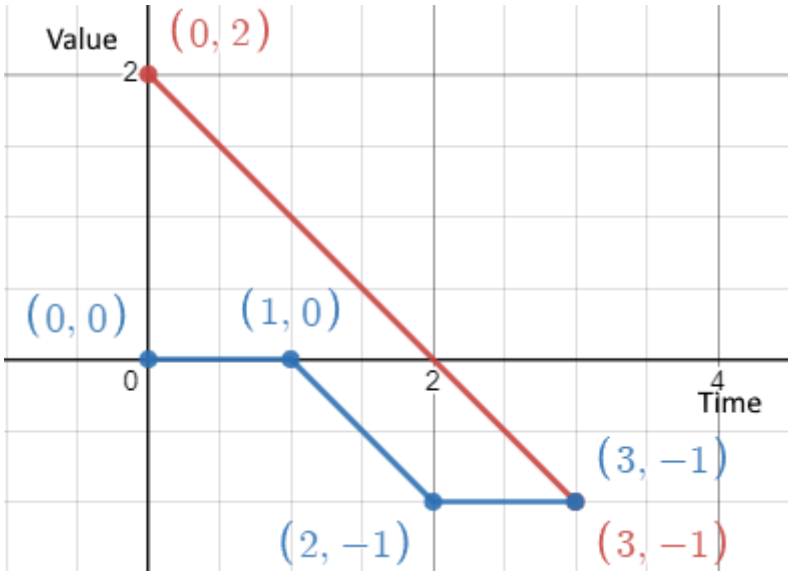

Figure 4.3: Incorrect computation for *computeComparableSignals* found in the original implementation (using the data in Figure 4.1

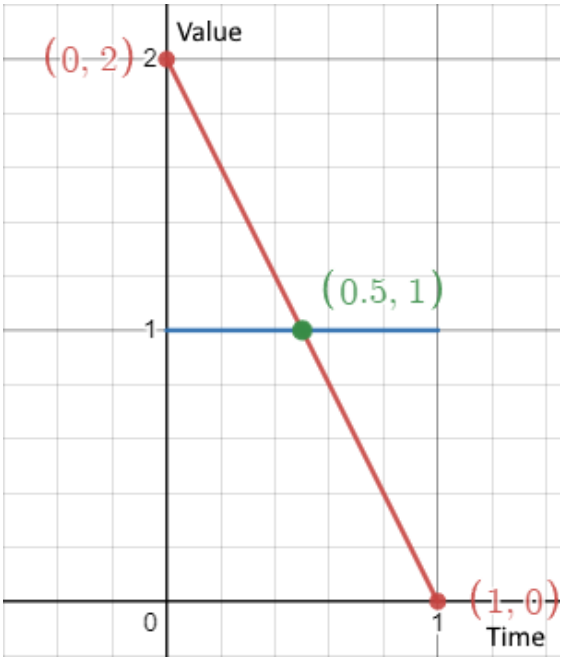

Figure 4.4: Two example signals to demonstrate the difference in behavior in the comparison operation

We noted an issue in the comparison operators in the case of comparison with a constant. As an example, consider signals x and y, both defined on the interval [0, 1], with $x = [\{0, 2\}, \{1, 0\}]$ and $y = \overline{\mathbf{1}}$. There is a point of intersection, where $t = 0.5$. This case is visualized in Figure 4.4. Consider the expression $x > y$.

The evaluation of $x > y$ requires the computation of the intersection at 0.5 in order to exactly represent the result signal. The old implementation would compute the result signal as $s = [\{0, 1\}, \{1, 0\}]$ (Boolean semantics). At first sight, that solution may seem good. However, in Boolean semantics, these signals are step functions. This translates to a function that is *True* on the interval $[0, 1[$ and *False* on the interval $[1, 1]$. Clearly, this is not the expected solution.

The true solution should be $s = [\{0, 1\}, \{0.5, 0\}, \{1, 0\}]$ (Boolean semantics). This, just like before, is a step function. In this case, though, the addition of the point at $t = 0.5$ means that we can now introduce the step required to exactly represent this signal. The result is visualized in Figure 4.5).

The key point is that the old implementation failed to consider intersections with a constant. While the previous work [18] did recommend using subtraction over comparison in quantitative semantics (e.g. preferring $x-5$ over $x > 5$, since it preserves the quantitative nature of the signal), the other binary arithmetic and comparison operations suffer from the same implementation defect so this is, in this case, not a solution.

Through the implementation of the `Signal` class, and using it for every signal (including constants) we ensure that all these cases are treated equally — our implementation will compute the intersections with a constant signal identically to the intersection computations between any other two signals. A potential optimization of our implementation introduces a special case here (since intersection with a constant can be more cheaply computed than the general case).

Combined with the corrections to the computation of comparable signals (Section 4.1), we have now fully corrected the errors seen in all binary operations. Since the intersections are computed,

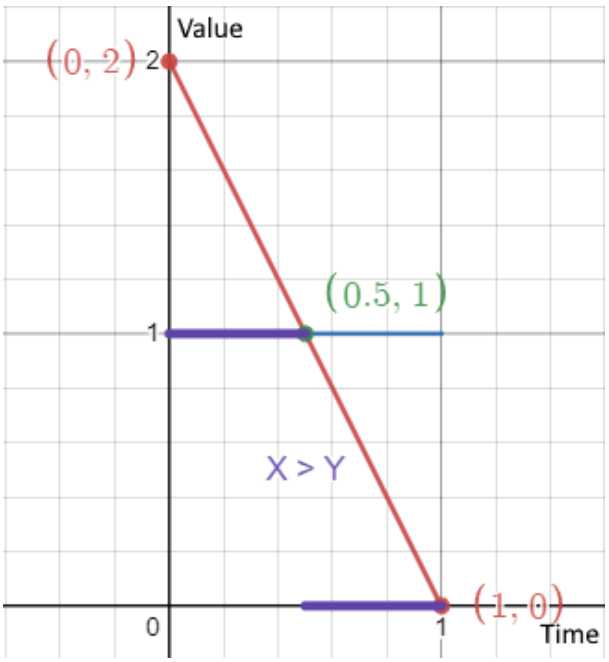

Figure 4.5: Two example signals to demonstrate the difference in behavior in the comparison operation

even for constants, and signals are correctly limited to the intersection between their domains, these operations can be implemented point-wise. The actual operation is trivially implemented: simply as the corresponding operator in Python, followed by a conversion to `int`.

4.3 Absolute Value

The absolute value implementation is rather trivial to do linearly; the intuitive method works. We simply iterate over each point of the signal and modify the value at every checkpoint to be the absolute value of that checkpoint.

4.4 Conjunction and Disjunction

The logical conjunction (AND) and disjunction (OR), in quantitative semantics, correspond to a minimum and maximum function, respectively. The implementation, similarly to the comparison operators, is trivial after proper signal pre-processing. The same conditions hold for checkpoints that must be present in the result signal, so using the *computeComparableSignals* method we can process our input data such that no inaccuracies will exist in the result signal.

The implementation of the operators, then, is essentially identical to the comparison operators (but with max/min functions instead of comparison operator lambdas in quantitative semantics).

4.5 Eventually

The *Eventually* operation in STL is, essentially, a maximum. In its untimed context (that is, on an unbounded interval) it is the maximum of the signal from 'now' until the end of the signal. In the case of a bounded interval, the computation is a sliding window maximising function. This type of operation is something that has been studied and an efficient method, linear in the size of the input with no impact from the window size, is known.

4.5.1 Timed Eventually

This algorithm was re-written from scratch based on the algorithm presented in *Streaming Maximum-Minimum Filter Using No More than Three Comparisons per Element* [19]. This is the same paper that was used to originally define the efficient algorithm for the computation of this operation [7]. However, during the translation of their description of the algorithm to our implementation, we repeatedly encountered errors. It is unclear whether the fault lies with their work or if we simply made a mistake in translation — we assume the latter. The easiest way to identify the mistake, at the time, seemed to be to do the translation ourselves and compare the outcome to the method implementation we had at the time.

Translating the method, and then comparing our implementation, is more involved than simply translating the method and replacing our previous implementation entirely, though. Based on that reasoning, we have re-implemented the algorithm. This has resulted in slight differences with the algorithm we were originally adapting [7]. The algorithm we obtained is fully detailed in Algorithm 1.

The actual implementation has a few extra lines of code to handle special cases, such as an input signal with fewer than 2 checkpoints, an interval I = $[a, b]$ with $a = b$. Further, we shift the input signal such that a becomes zero. If we consider an interval $[1, 3]$, then this means that any data the `Signal` object contains in the interval $[0, 1[$ is entirely irrelevant - it has no impact on the computation. The easy way to handle this is to shift the signal by $-a$ and removing all checkpoints where the time has become negative. This allows us to simplify the interval (after shift) from $[0, b - a]$ to simply the single value $b - a$.

These special cases are not explicitly handled in Algorithm 1 (so that it may fit on a single page), we simply specify the requirements for the section of code we discuss. Our implementation of the algorithm in the tool does consider these cases (and handles them appropriately). For the lines references in the rest of this section, they always refer to the line numbers in Algorithm 1.

The idea behind the algorithm is identical to the idea presented by Lemire [19]. We maintain a set *maximumCandidates* (sorted in descending order, by value) that contains the elements of the signal that may become the maximum at some future point.

Any value that does not have the potential to be the maximum at some point in the future is discarded from the set. These are divided into two categories; values that can no longer be an outcome of the algorithm because they are no longer in the window and those that are outclassed by a better candidate.

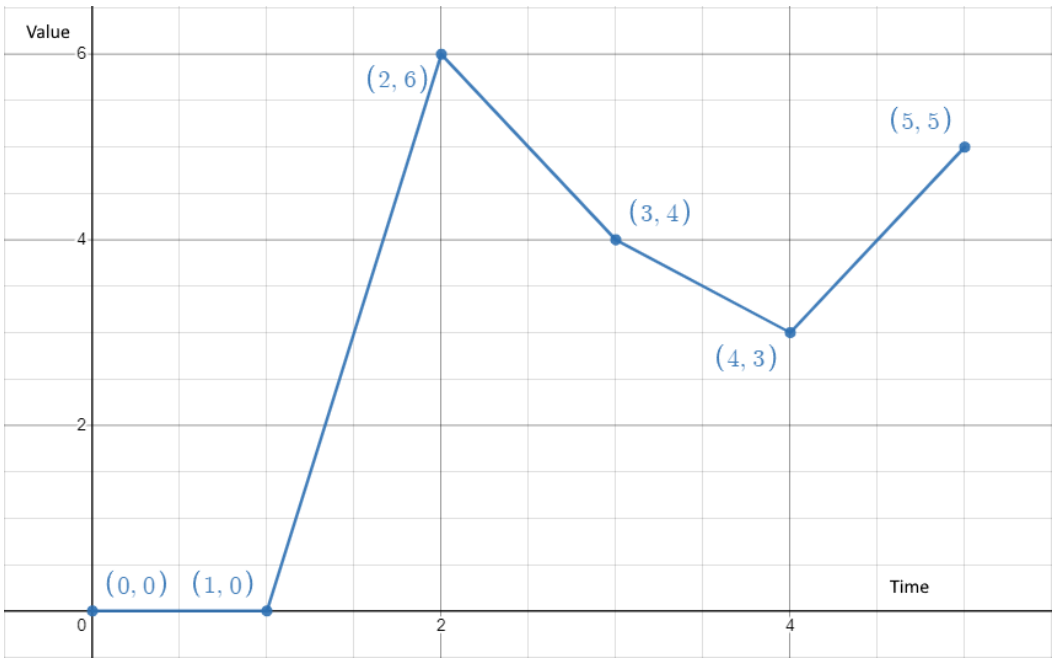

Figure 4.6: Example signal to illustrate the sliding window maximizing function

4.5.1.1 Example

To illustrate the way this algorithm functions, we work through an example in this section. For the computation, we assume we are computing the timed eventually operator for the interval $[0, 2]$. The input signals is shown in Figure 4.6. Denoting the signal in Figure 4.6 as s, we are considering the operation $\Diamond_{[0,2]}(s)$.

The algorithm iterates over the signal in ascending time order and considers line segments. We start at the first line segment in the signal, defined by the sample points $(0, 0)$ and $(1, 0)$.

For this case, the choice of sample point is rather irrelevant; both have the same value. The slightly more efficient choice would be the point $(1, 0)$, since it stays in the window longer (though the result is identical in this case, the next candidate will cause a value-based removal). The chosen candidate is added to the set of maximum candidates between Line 15 and Line 22. Since, at this time, we have not yet explored a full window in the input signal (Line 26), no checkpoint is added to the output signal.

Next, we consider the segment $(1, 0)$ to $(2, 6)$. The data in maximumCandidates, currently, is still part of the window, so no pop occurs (Line 12 to Line 14). Trivially, the maximum candidate on this segment is $(2, 6)$ because its value is larger. We append it to the set of maximum candidates (Line 15 to Line 22). Then, we check if any of the previous candidates can now be removed; which they can. The sample point $(1, 0)$ will, at no point in the future, have an impact on the maximum. The current window contains the sample point $(2, 6)$ which sets a lower limit on the value of the maximum until it leaves the window. Since this lower limit is higher than the value of $(1, 0)$, that checkpoint won't be useful anymore and can be safely removed. This removal occurs between Line 23 and Line 25. We visualize the current state of the algorithm in Figure 4.7.

At this point, the first value in the result signal is appended in Line 27; we add the sample point $(0, 6)$ to the result signal.

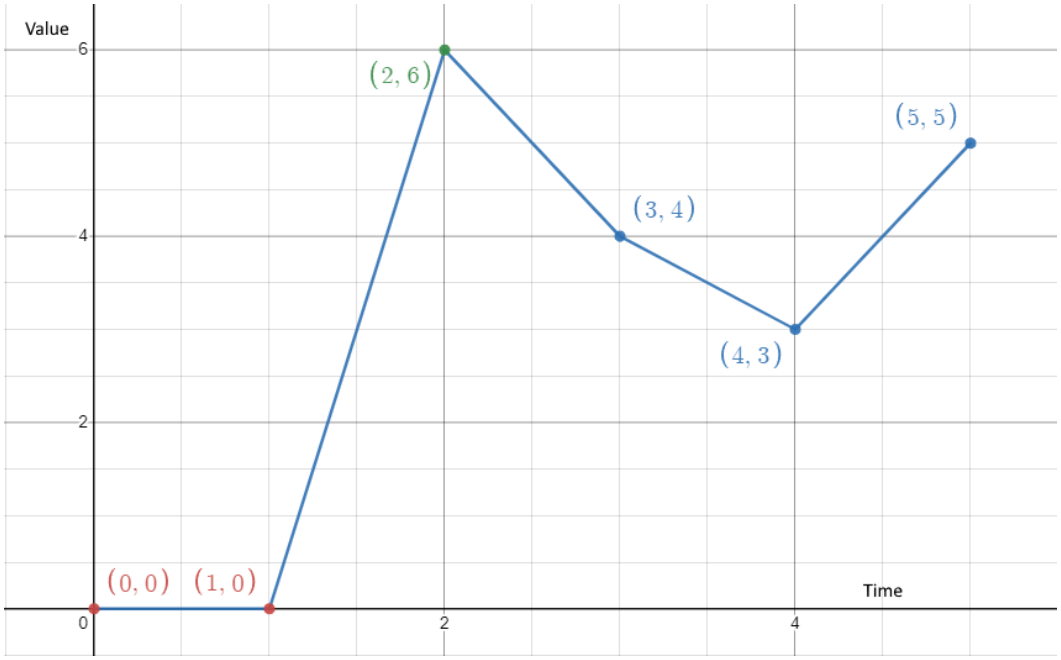

Figure 4.7: The first full interval (width = 2) has been explored. Red sample points were not added to maximum candidates, or have been removed. Green indicates that the point is currently in maximum candidates. Blue points have not yet been fully considered.

The next step explores the segment $(2, 6)$ to $(3, 4)$. On this segment, $(2, 6)$ is the maximal candidate. Since it is currently the latest entry in maximal candidates, it will not be added to the set. Very little changes in this step. Another result sample point is added: $(1, 6)$.

Next, consider the segment defined by $(3, 4)$ and $(4, 3)$. Here, $(3, 4)$ is the candidate maximum. $(3, 4)$ has a value lower than the element currently in the maximum candidates set, but it will be added because its time stamp is higher. This means that, at some point in the future, when $(2, 6)$ is no longer in the sliding window, $(3, 4)$ may become the maximum (depending on future values). A third value is added to the output: $(2, 6)$. The current state is shown in Figure 4.8.

In the following iteration, we consider the segment defined by $(4, 3)$ and $(5, 5)$. This is the final step of the algorithm.

First, we note that the current upper bound of the window (always the largest time in the current segment, 5 at the moment) minus the width of the window (2) is larger than the time for one of the sample points in maximum candidates. Between Line 12 and Line 14, the sample point $(2, 6)$ is removed from the set because it is no longer in the sliding window.

Then, we append the point $(5, 5)$ to the set of maximal candidates since its value is the largest on the current segment. Following that addition, the other entry in maximal candidates will be removed since the value $(5, 5)$ will be a better candidate for the remainder of the time that $(3, 4)$ is in the sliding window. As a result, the set of maximum candidates will now only contain the point $(5, 5)$. This point is the value appended to the result signal, creating the last point $(3, 5)$.

The final result, then, is shown in Figure 4.9.

To handle the elimination of the values in the first category, we start at the first element of maximumCandidates and check its timestamp, discard if it is lower than the current lower bound. Repeat until you find some element in maximumCandidates with a time stamp larger

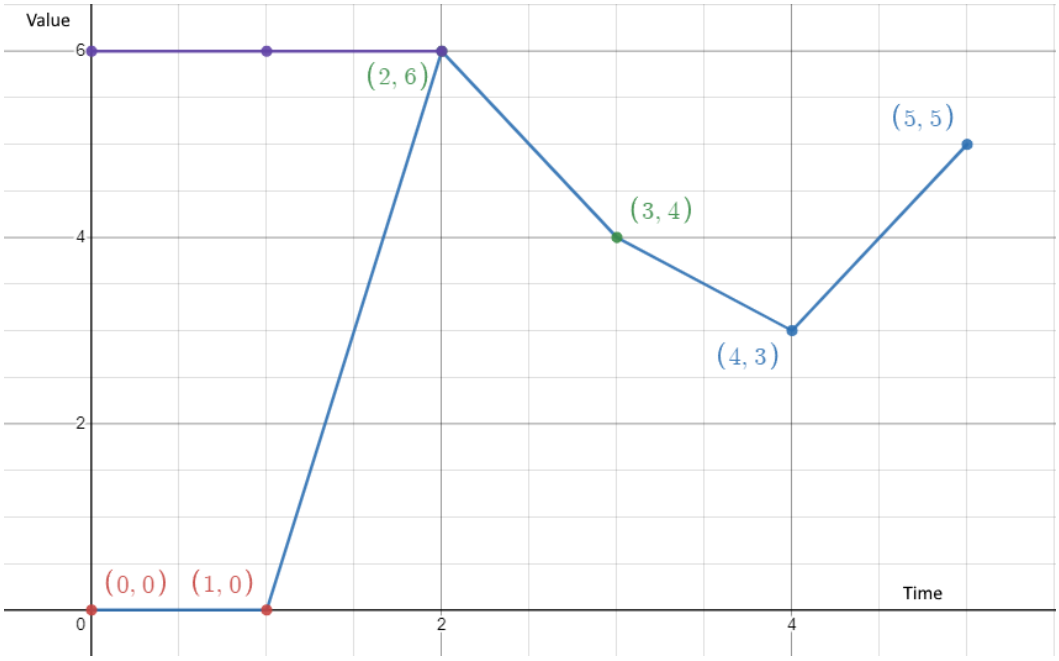

Figure 4.8: Segments up until time = 4 have been explored. Red sample points were not added to maximum candidates, or have been removed. Green indicates that the point is currently in maximum candidates. Blue points have not yet been fully considered. Purple visualizes the currently constructed part of the result signal.

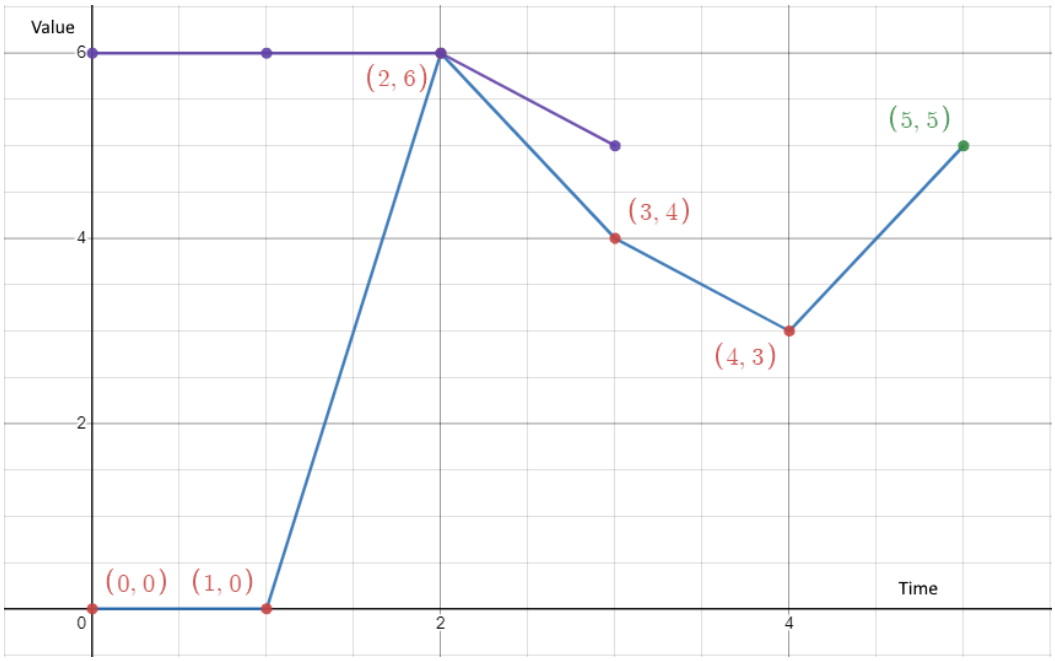

Figure 4.9: The timed eventually computation has completed. This is the final state of the algorithm. Red sample points were not added to maximum candidates, or have been removed. Green indicates that the point is currently in maximum candidates. Blue points have not yet been fully considered. Purple visualizes the currently constructed part of the result signal.

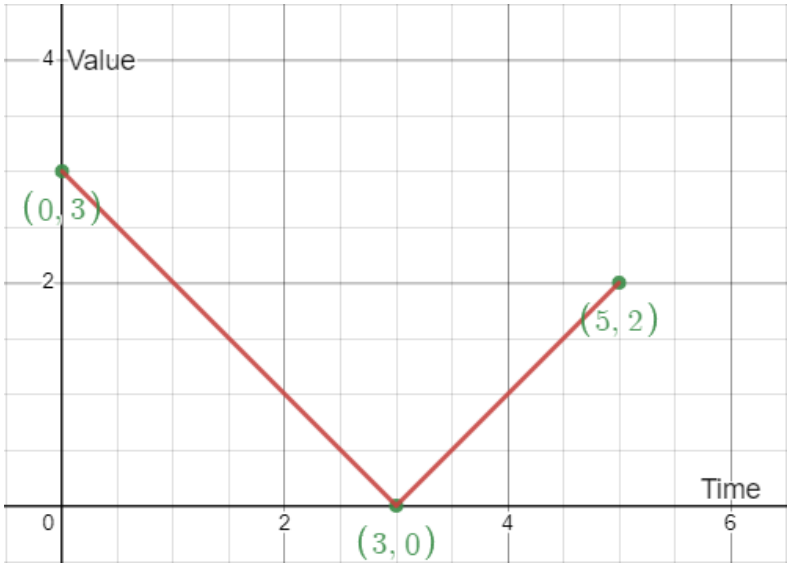

Figure 4.10: An example signal defined by three points (noted as {time, value} pairs)

than the lower bound or until maximumCandidates is empty. We check for these cases between Line 12 and Line 14.

For the second category, we exploit the fact that the set is sorted in descending order. The sorted order is maintained by checking that, after we insert a new element, that the preceding element is larger than the new one. Should this not be the case, we remove that preceding element and repeat either until the new element is the only element or until the preceding element is larger than the new one. This happens between Line 23 and Line 25.

Adding values to the output happens every iteration (Line 27), once at least an entire window width has been explored. The timestamp of the point in the result signal is the lower bound of the window, the value is the current maximum candidate (first element in the set maximumCandidates) which is the maximum on the interval [current window lower bound, current window lower bound + window width].

4.5.2 Untimed Eventually

The implementation for this operation was missing entirely in the STL tool before our work [18]. This operation is part of the definition of the efficient until algorithm [7]. It is, in essence, the same operation as the timed variant but with an interval I = [0, $+\infty$).

For any signal s, and $\varphi = \Diamond s$, the set of timestamps for φ will be the same as the set of timestamps for s plus, potentially, a few extras. The operation is fairly simple, in principle: for any `SignalValue` sv in s, compute the maximum of the values of all `SignalValues` in s with timestamp greater than or equal to the timestamp of sv.

Since all timestamps later in the signal must be considered, the natural approach is to start at the back of the signal. This is exactly the approach taken in the implemented algorithm (Algorithm 2).

There is one slight nuance to consider. Consider the signal as in Figure 4.10, defined by the three `SignalValues`. If we simply consider the maximum function for every timestamp in the

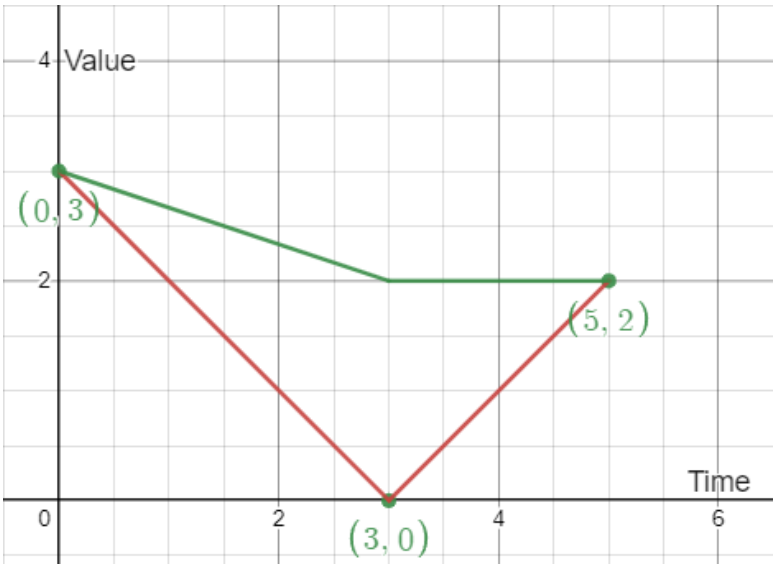

Figure 4.11: An incorrect result for the untimed eventually computation for the signal in Figure 4.10 when using only the existing signal timestamps

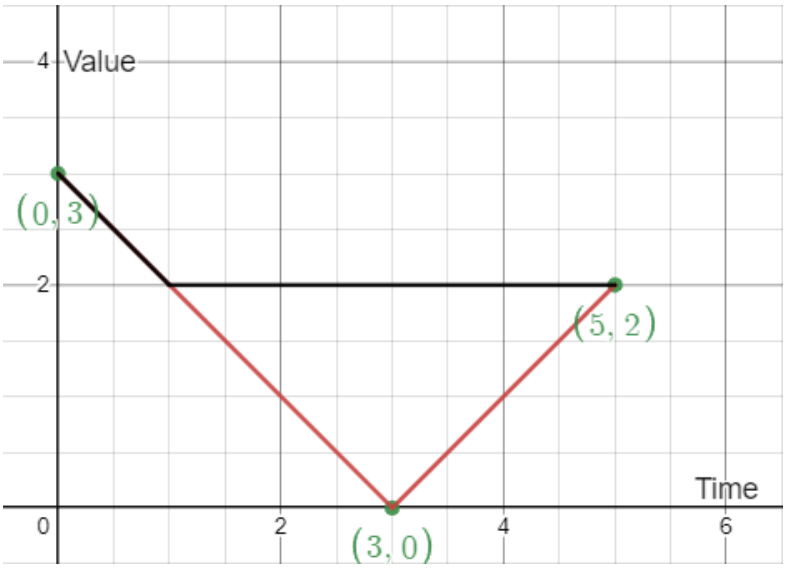

Figure 4.12: The correct result for the untimed eventually computation for the signal in Figure 4.10, with the extra introduced timestamp

signal, the result would be Figure 4.11. However, the correct result is Figure 4.12. The point where the transition to the new value must be made can be computed by finding the intersection point between the line defined by the first two points with the line defined by the last point (with a zero derivative). At that point, an extra checkpoint must be added to the signal in order to represent the correct result.

In general, this case occurs when considering a line segment defined by the `SignalValue`s $\{t_i, v_i, d_i\}$ and $\{t_{i+1}, v_{i+1}, d_{i+1}\}$. During the previous steps of the algorithm, a current highest value v_m will have been found.

Specifically, whenever $v_i > v_m > v_{i+1}$, this case pops up. At some point between t_i and t_{i+1}, the current segment must intersect $\overline{\mathbf{v}}_m$ if this condition is met. It is specifically that point of intersection that must be added to the result signal; the current segment, beyond that point, drops below a previously seen maximal value and is no longer relevant. Before that point, the segment defines exactly the maximal values. Both the point $\{t_i, v_i, d_i\}$ and the point of intersection must be added to the output signal to fully define the result of this computation.

We describe our implementation in detail in Algorithm 2, the special case is implemented between Line 12 and Line 20.

4.6 Until operation

Previous work implemented two versions of the timed until computation [18]. One based entirely on the STL syntax, which worked (mostly) correctly but is computationally inefficient. The other is a more efficient approach, based on a sliding window maximizing algorithm, adapted to work using variable time steps (and thus a non-constant number of elements in the window) [7].

For the former algorithm, we found minor errors during our work. However, none of them were caused by the algorithm's implementation itself - rather, the errors were present in some helper methods (e.g. what we discussed in Section 4.1).

The latter was rather extensively erroneous. The idea was to implement the efficient algorithm introduced in *Efficient Robust Monitoring for STL* [7]. Unfortunately, large segments of the algorithm they present were entirely missing from the implementation and some of the parts that did make it were not entirely accurate.

In this section, we address the issues we found in the implementation of the efficient version of the *Until* operation. Any errors found in other operators (or helper methods) that are used as subroutines in this operation are discussed in their own sections (Section 4.5, Section 4.1).

4.6.1 Untimed Until

This operation was partially implemented previously [18]. Unfortunately, the implementation failed to make a distinction between the untimed and the timed versions of any operator. Instead, only the timed versions were implemented (in this case, using the definition of the untimed operator — predictably creating in incorrect results). Further, the definition of the algorithm in the work our pre-existing implementation was based on contained slight errors. Referring to the implementation of the Untimed Until algorithm in Efficient Robust Monitoring For STL (Algorithm 2, in that paper) [7], we note an issue.

– If $dy(s) \leq 0$ then $\forall \tau \in [s,t)$, $\inf_{[s,\tau]} y = y(\tau)$. Thus $z_t(s) = \sup_{\tau \in [s,t)} \min\{y'(\tau), y(\tau)\}$, and $z(s) = \max\left\{z_t(s), \min\{y(t), z(t)\}\right\}$

– Otherwise $\forall \tau \in [s,t)$, $\inf_{[s,\tau]} y = y(s)$. Therefore $z_t(s) = \sup_{\tau \in [s,t)} \min\{y'(\tau), y(s)\} = \min\{y(s), \sup_{[s,t)} y'\}$, and $z(s) = \max\left\{z_t(s), \min\{y(s), z(t)\}\right\}$.

Figure 4.13: Efficient Robust Monitoring For STL discussion of the underlying mathematics of the untimed until operation [7]

$z_0 := \overline{\bot}$
$i := n_y - 1$
while $i \geq 0$ **do**
  if $dy(t_i) \leq 0$ **then**
    $z_1 := \text{Compute}(\Diamond, y'_{\restriction [t_i, t_{i+1})})$
    $z_2 := \text{Compute}(\wedge, z_1, y_{\restriction [t_i, t_{i+1})})$
    $z_3 := \text{Compute}(\wedge, \overline{y(t_{i+1})}, z_0)$
    $z_{\restriction [t_i, t_{i+1})} := \text{Compute}(\vee, z_2, z_3)$
  else
    $z_1 := \text{Compute}(\wedge, y'_{\restriction [t_i, t_{i+1})}, y_{\restriction [t_i, t_{i+1})})$
    $z_2 := \text{Compute}(\Diamond, z_1)$
    $z_3 := \text{Compute}(\wedge, y_{\restriction [t_i, t_{i+1})}, z_0)$
    $z_{\restriction [t_i, t_{i+1})} := \text{Compute}(\vee, z_2, z_3)$
  end if
  $i := i - 1$
  $z_0 := \overline{z(t_{i+1})}$
end while
return z

Figure 4.14: Efficient Robust Monitoring For STL algorithm for untimed until [7]

The condition in that algorithm is reversed compared to their discussion of the underlying mathematics (Efficient Robustness Monitoring For STL, Section 4.Operator U - relevant section included in Figure 4.13). The robustness computation defined mathematically there does not match what they describe in their Algorithm 2 (included in Figure 4.14). Essentially, swapping the body of the if-condition with the body of the else in the algorithm is required.

We present our implementation of the untimed until algorithm (which handles this swap) in Algorithm 3.

4.6.2 Timed Until

This part of the implementation was, previously, entirely missing since the timed operator was implemented using the operation definition of the untimed operator [18]. We introduce the separation between them and implement the timed until operation as required [7]. We directly apply the decomposition of the timed until operation into untimed until and timed eventually [41] [7]. Based on the previously defined operations (Section 4.5.1 and Section 4.6.1), we have implemented the operator and detail the implementation in Algorithm 4.

4.7 Remaining issues

4.7.1 Numerical Precision

Our implementation, for performance reasons, does not use exact maths. We simply rely on Python's floating point operations and consider that to be sufficiently precise.

At various points during robustness computations, signals must be interpolated. This involves a division by a difference in time; when the two times are close together, this can easily become incredibly unstable (and result in large errors). Even if the initial error is small, these inexact results may propagate through other operators and influence their results significantly.

While this behavior is not desired, the performance cost of preventing this type of error entirely is prohibitive. Small errors resulting from non-exact arithmetic is expected behavior and not an issue. However, the significant problems that occur when two timestamps that are almost equal are used in a derivative computation are undesired. To mitigate this in a way that doesn't impact performance significantly, we enforce a limit on the amount of decimals for a timestamp in the system.

In our implementation, this precision is set to $1e-5$. Times are limited to this resolution and, in that way, we can avoid the issue of a tiny error in a timestamp computation resulting in incredibly high derivatives.

Algorithm 1 Pseudo-code for the implementation of the Timed Eventually operation in the STL tool

Require: *signal* is an instance of `Signal`
Require: $signal.getCheckpointCount() >= 2$
Require: $signal.getTime(0) = 0$
Require: *interval* is a non-empty instance of `Interval`
Require: $interval.getLower() = 0$
Ensure: *result* is the robustness Signal of $\Diamond_{interval} signal$

1: **function** COMPUTETIMEDEVENTUALLY(signal: `Signal`, interval: `Interval`)
2: *result* ← `Signal`() ▷ Empty signal
▷ Since *interval*.getLower() == 0, the width is equal to the upper bound
3: *windowWidth* ← *interval*.getUpper()
4: *maximumCandidates* ← {}
▷ if *windowWidth* < size of first segment, we must handle the first point separately
5: **if** *windowWidth* < *signal*.getTime(1) - *signal*.getTime(0) **then**
6: *initial* ← max(*signal*.getValue(0), *signal*.computeInterpolatedValue(*windowWidth*))
7: *result*.emplaceCheckpoint(*signal*.getTime(0), *initial*)
8: **end if**
9: **for** *index* in range(*signal*.getCheckpointCount() - 1) **do**
▷ Since a signal is f.p.l.c., a set of two sequential checkpoints defines a line segment
10: *segment* ← *signal*.getCheckpoint(*index*), *signal*.getCheckpoint(*index* + 1)
11: *cwlb* ← *segment*[1].getTime() - *windowWidth* ▷ Current Window Lower Bound
12: **while** *maximumCandidates* and *maximumCandidates*[0].getTime() < *cwlb* **do**
13: *maximumCandidates*.pop(0)
14: **end while**
▷ Find which value in the current window is the candidate
15: **if** *segment*[0].getValue() >= *segment*[1].getValue() **then**
16: *candidate* ← *segment*[0]
17: **else**
18: *candidate* ← *segment*[1]
19: **end if**
▷ Insert the candidate if it is not there yet (could otherwise be inserted as upper and lower)
▷ Can be omitted if desired - duplicate entry in set does not break anything
20: **if** *maximumCandidates*.isEmpty() or *maximumCandidates*[-1] != *candidate* **then**
21: *maximumCandidates*.insert(*candidate*)
22: **end if**
▷ Filter maximumCandidates that will not be relevant anymore because newer one is bigger
23: **while** *maximumCandidates*.size() >= 2 and *maximumCandidates*[-2].getValue() < *maximumCandidates*[-1].getValue() **do**
24: *maximumCandidates*.pop(-2)
25: **end while**
26: **if** *segment*[1].getTime() >= *windowWidth* **then**
27: *result*.emplaceCheckpoint(*cwlb*, *maximumCandidates*[0].getValue())
28: **end if**
29: **end for**
▷ Ensure the result signal has the derivatives set correctly
30: *result*.recomputeDerivatives()
31: return *result*
32: **end function**

Algorithm 2 Pseudo-code for the implementation of the Untimed Eventually operation in the STL tool

Require: *signal* is a non-empty instance of `Signal`
Ensure: *result* is the signal representing $\Diamond signal$

```
1: function computeUntimedEventually(signal: Signal)
2:     result ← Signal()
3:     previousValue ← −∞
   ▷ Iterate over the signal in reverse order, because later values must always be considered
4:     for timeIndex in reversed(range(signal.getCheckpointCount() - 1)) do
5:         if signal.getValue(timeIndex) <= signal.getValue(timeIndex + 1) then
   ▷ if the current segment is ascending, use max of previous and second point
6:             value ← max(signal.getValue(timeIndex + 1), previousValue)
7:             result.emplaceCheckpoint(signal.getTime(timeIndex), value)
8:         else if
   previousValue >= signal.getValue(timeIndex) > signal.getValue(timeIndex + 1) then
   ▷ if both points smaller than previous, use previous
9:             result.emplaceCheckpoint(signal.getTime(timeIndex), previousValue)
10:        else if
   signal.getValue(timeIndex) > signal.getValue(timeIndex + 1) >= previousValue then
   ▷ if larger than previous value and ascending, use first point
11:            result.emplaceCheckpoint(signal.getCheckpoint(timeIndex))
12:        else
   ▷ Descending and crosses previous value; first add the new maximum point from current
   segment
13:            result.addCheckpoint(signal.getCheckpoint(timeIndex)
   ▷ Then compute the line intersection and add intersection point
14:            previousLine ← LineSegment((−∞, previousValue), (+∞, previousValue))
15:            currentStart ← (signal.getTime(timeIndex), signal.getValue(timeIndex))
16:            currentEnd ← (signal.getTime(timeIndex + 1),
   signal.getValue(timeIndex + 1))
17:            currentLine ← LineSegment(currentStart, currentEnd)
18:            intersection ← intersect(currentLine, previousLine)
19:            result.emplaceCheckpoint(intersection.x, intersection.y)
20:        end if
21:    end for
22:    result.recomputeDerivatives()
23:    return result
24: end function
```

Algorithm 3 Pseudo-code for the implementation of the Untimed Until operation in the STL tool

Require: *lhsSignal* is a non-empty instance of
Require: *rhsSignal* is a non-empty instance of `Signal`
Require: *lhsSignal* and *rhsSignal* have identical timestamps (i.e. `Signal`.computeComparableSignals() has been called)
Ensure: *result* is the `Signal` object representing the result of *lhsSignal* **U** *rhsSignal*

```
1: function computeUntimedUntil(lhsSignal: Signal, rhsSignal: Signal)
2:     previous ← ∞
3:     result ← Signal()
4:     index ← lhsSignal.getCheckpointCount() - 2
5:     while index >= 0 do
   ▷ Compute the interval we currently work on; we iterate from back of the signal to the front
   ▷ Using lhsSignal - can be done equivalently using rhsSignal
6:         currentInterval ← [lhsSignal.getTime(index), lhsSignal.getTime(index+1)]
7:         lhsInterval ← lhsSignal.computeInterval(currentInterval)
8:         rhsInterval ← rhsSignal.computeInterval(currentInterval)
   ▷ Behaviour depends on the slope of the current segment
9:         if lhsSignal.getDerivative(index) <= 0 then
10:            lhsConst ← Signal.constant(lhsSignal.getValue(index + 1))
11:            minConstPrev ← computeAnd(lhsConst, previous)
12:            lAndR ← computeAnd(lhsInterval, rhsInterval)
13:            EvtLAndR ← computeUntimedEventually(lAndR)
14:            currentOut ← computeOr(EvtLAndR, minConstPrev)
15:        else
16:            lhsConst ← Signal.constant(lhsSignal.getValue(index)
17:            evtR ← computeUntimedEventually(rhsInterval)
18:            LAndEvtR ← computeAnd(lhsInterval, evtR)
19:            LAndPrev ← computeAnd(lhsInterval, previous)
20:            currentOut ← computeOr(LAndPrev, LAndEvtR)
21:        end if
   ▷ Add the segments to the output (open interval)
22:        for cp in currentOut.getCheckpoints() do
23:            if currentInterval.contains(cp.getTime(), closed=False) then
24:                result.addCheckpoint(cp)
25:            end if
26:        end for
27:        previous ← Signal.constant(
             result.getValue(result.computeIndexForTime(lhsSignal.getTime(index)))
           )
28:        index = index - 1
29:    end while
30:    result.recomputeDerivatives()
31:    return result
32: end function
```

Algorithm 4 Pseudo-code for the implementation of the Timed Until operation in the STL tool

Require: *lhs* is a non-empty instance of Signal
Require: *rhs* is a non-empty instance of Signal
Require: *rhs* and *lhs* have the same timestamps (i.e. `Signal`.computeComparableSignals() has been called)
Ensure: *result* is the signal representing the robustness of $lhs\mathbf{U}_I rhs$

```
1: function computeTimedUntil(lhsSignal: Signal, rhsSignal: Signal, interval: Interval)
   ▷ We distinguish between the bounded and unbounded versions
2:     if isFinite(interval.getUpper()) then
3:         evtRhs ← computeTimedEventually(rhs, interval)
4:         unboundedInt ← Interval(interval.getLower(), ∞)
5:         unboundedUntil ← computeTimedUntil(lhs, rhs, unboundedInt)
6:         result ← computeAnd(evtRhs, unboundedUntil)
7:     else
8:         untimedUntil ← computeUntimedUntil(lhs, rhs)
9:         prefixInterval ← Interval(0, interval.getLower())
10:        result ← computeTimedAlways(untimedUntil, prefixInterval)
11:    end if
12:    return result
13: end function
```

5 Data Generation

For a machine learning algorithm to reliably perform well, a lot of data is required. In order to get the required data for our application, we use OpenAI gym environments to generate it [30] [31]. We train a reinforcement learning agent, created using PyTorch, to solve these environments [42]. We store the states encountered by the agent during its training, compute the robustness value for each of them (using the algorithms discussed previously, in Section 4), and use these results as training data for our robustness estimator (Section 6).

The formulas we present in this section are relatively simple; they are not intended to fully describe the mechanics of the environment. Rather, they simply describe the terminating conditions for the environments such that we can compute a robustness value for that simplified case. Creating an STL formula that fully captures the environmental mechanics is possible, but decidedly non-trivial. Our work focuses on the creation of a good robustness estimator for a given STL formula; whether that formula entirely and correctly models the use-case is not a significant factor for the quality of the estimation and thus considered out of scope for this work. It simply creates a target function that does not exactly represent the exact use-case.

The implementation of the agents solving the Open AI environments can be found in our (second) GitHub repository [38]. Since the agents are used as a tool to generate requisite data and are not directly related to the STL tool, we felt that splitting them up in this way made sense.

5.1 Environments

5.1.1 Cartpole

CartPole-v1 is an environment where the agent controls a cart, on which a pole is balanced. The goal of the environment is to move the cart in such a way that the pole remains upright, without moving the cart too much to either side. Upright is defined as within approximately 12° (0.2095 rad) of vertical, the cart is allowed to move 2.4 units in either direction. When either of these restrictions is violated, the episode immediately fails. An episode is considered successful if the agent maintains the upright position of the pole for 500 steps.

5.1.1.1 Agent

We use a simple feed-forward, fully-connected neural network to learn the environment. We use PyTorch [42] to define the agent. We construct the model from two PyTorch `Linear` modules; the first takes 4 input features and outputs 32 features, the second takes 32 input features and has 2 output features. Between them is a Rectified Linear Unit (ReLU) activation function.

ReLU is an activation function that can be defined piecewise, as in Definition 5.1.1. It is a simple, easy to compute function that is used to introduce non-linearity into the neural network.

Without such an activation function, no non-linear elements can be present in the result; this would mean the neural network is equivalent to regression.

Definition 5.1.1 (Rectified Linear Unit)

$$\text{ReLU}(x) = \begin{cases} x \leq 0 : 0 \\ x > 0 : x \end{cases}$$

We randomly initialize the model parameters and begin the reinforcement learning process. We train the agent over the course of 1000 episodes. During this process, we use the Adam optimization method to adapt the model parameters; it is configured with a learning rate set to $5 * 10^{-3}$. After training, the agent (in a 10-episode test) obtains perfect scores, though that should not be a requirement (or even a factor) for the success of the rest of the work.

5.1.1.2 Formula

Using c to denote the cart position signal and p to denote the pole angle signal, we can define Formula 5.1 describing the environment's failure conditions. When the robustness computed using this formula is negative, the episode has failed.

$$\rho = (\square(2.4 - c) \wedge \square(2.4 + c)) \wedge (\square(0.2095 - p) \wedge \square(0.2095 + p)) \tag{5.1}$$

Formula 5.1 has the property that, no matter how long the signal is, every single value must be considered. This makes the robustness hard to estimate. This type of operation runs into one of the main problems encountered by online monitoring methods, since as long as the entire signal has not yet been generated, insufficient information is available.

This issue is simple to address, in principle. If the robustness data set is labeled using an online robustness computation algorithm rather than an offline version, the labels will correctly account for the unavailable information (the way in which that happens is implementation defined, as previously discussed). As long as the estimator and the online robustness computation algorithm receive the same input data, the results we obtain should generalize to this case as well.

We have decided against this approach for performance reasons, since offline robustness computation can be performed much more quickly (as previously discussed). Instead, we avoid the problem by using timed versions of the operators (with an arbitrarily chosen interval). Following this modification, we obtain Formula 5.2.

With Formula 5.2, the input interval is always the same. While the amount of sample points within that interval can vary significantly, we would overall expect a roughly consistent measuring frequency for any particular application. Small deviations are expected, but the amount of sample points is unlikely to be, for example, an order of magnitude apart. This simplification should allow improved results for the estimator.

$$\rho = (\square_{[0,50]}(2.4 - c) \wedge \square_{[0,50]}(2.4 + c)) \wedge (\square_{[0,50]}(0.2095 - p) \wedge \square_{[0,50]}(0.2095 + p)) \tag{5.2}$$

5.1.1.3 Normalization

A further improvement we can make is to normalize the signal values. From the formula, it is clear that the range over which the cart position value can vary is much larger than the range for the pole angle. While this is not an issue for an ANN learning to solve the environment, it does have some consequences for the robustness computation.

Since the intervals for the two variables are different, the robustness computation is impacted differently by them. Assuming the pole is perfectly centered and the cart is 90% of the allowed interval over to the right (position 2.16), the computed robustness will be 0.24. If we consider the reverse case, we might expect the robustness value to be the same. However, using the computation defined by Formula 5.2, that is not the case. With the pole at an angle of 90% of the allowed interval from vertical (angle 0.18855), the computed robustness value would be 0.02095.

There is a significant difference in magnitude between these. In order to address this, we normalize the environment outputs so that the non-failing range for both signals is the interval $[-1, 1]$ (by dividing by 2.4 and 0.2095 respectively).

We specifically choose the interval $[-1, 1]$ because it is symmetric around a mean 0 and limited in magnitude. Training for neural networks is generally believed to be easier with data that is normalized to such an interval.

We do not expect this change to make a meaningful difference to the training performance of our robustness estimator. It makes intuitive sense, however, to have the impact of both variables be identical; this change ensures the computed robustness value is more representative of how far from the boundary condition the current state of the variables is. Another option to consider is to normalize the value ranges based on how volatile the values are, but that is not further explored in this work.

Formula 5.3 is obtained as a result of this normalization.

$$\rho = (\square_{[0,50]}(1 - c) \wedge \square_{[0,50]}(1 + c)) \wedge (\square_{[0,50]}(1 - p) \wedge \square_{[0,50]}(1 + p)) \tag{5.3}$$

5.1.1.4 Margin

We further introduce a margin in the formula to improve training performance. Without this modification, the values are almost always positive. A negative value is only encountered, at most, once per episode (exactly when the episode terminates because of a failure).

To facilitate the supervised learning of an estimator, more variation in the values is desired. The main goal is to ensure that the estimation is able to accurately state the sign of the robustness - an error in magnitude is less problematic. Because of this, we want more changes of sign in the signal we consider — by having a more even spread between the two categories in the training data, better performance can be achieved.

We introduce something equivalent to a safety margin in the robustness computation. Rather than considering the entire interval of values allowed by the environment as satisfying the formula, we impose a stricter requirement (50% of each interval). In doing so, we ensure that there will be far more changes of sign in the computed robustness. The presence of more such data should mean that the estimation tool will be far better at estimating the sign of the robustness value.

$$\rho = (\square_{[0,50]}(0.5 - c) \wedge \square_{[0,50]}(0.5 + c)) \wedge (\square_{[0,50]}(0.5 - p) \wedge \square_{[0,50]}(0.5 + p)) \tag{5.4}$$

Formula 5.4 is the final formula we will use to compute the Cartpole robustness.

5.1.2 Mountain car

In *MountainCarContinuous-v0*, the agent controls a car, initially located in a valley, where the goal is to ascend the valley on one side. However, simply driving in one direction, the car is unable to drive all the way up the hill.

Instead, the car must drive back and forth between the two slopes (on either side of the valley) to gain more momentum and complete its final ascent. The environment is completed successfully when the car's position is larger than or equal to 0.45. An episode fails if the car has not reached its goal at time-step 1000.

5.1.2.1 Agent

Due to the more complex nature of the continuous environment, and the fact that these agents are simply used for data generation (and so not the main focus of the work), we decided against creating our own agent for this environment. Instead, we used a pre-existing agent that solves the environment and modified it to store the training data [43]. This implementation offers many different agents; we have chosen their Soft Actor-Critic (SAC) agent to use in our work.

Our GitHub repository [38] contains a fork of their implementation, with the minor modifications necessary for recording the training data we require.

Similar to the cartpole environment, we train the agent over 1000 episodes. All other parameters for the training process are exactly as in the original implementation [43].

5.1.2.2 Formula

In Section 5.1.1, the formula defines the failure boundary and a negative robustness means the environment has failed. In this environment, the situation is reversed; we define the success boundary and expect a negative robustness until exactly the time-step that the environment is successfully completed.

We add a time limit, shorter than the one imposed by the environment, in the hope that we will see an increasing trend (if we ignore the oscillation) in robustness as the agent approaches the goal. Using p as the car position signal, we get Formula 5.5.

$$\rho = \Diamond_{[0,50]}(p - 0.45) \tag{5.5}$$

This, similarly to the cartpole case, is an incredibly simple way to represent the problem. Since the car is at the bottom of a valley, the actual robustness of the environment should be influenced by more than simply the car's position; a car at the top of the hill on the wrong side of the valley is closer to the solution than an unmoving car, at the bottom of the valley.

Similarly, a car with high velocity is closer than one with low velocity. The fact that our formula does not accurately reflect the entire environment means that the computed robustness also does not correspond exactly with the robustness one might expect.

The current formula does not account for those factors at all. However, we're not aiming to exactly capture the environment mechanics; we simply use the formula to create a labeled data set for our robustness estimation experiments. In that sense, this formula will suffice.

5.1.2.3 Normalization

In the cartpole case (Section 5.1.1), we had a formula featuring two signals and added a normalization step to make both variables have an identical impact on the robustness. Subsequently, we modify the values for both input signals such that they fall on the interval $[-1, 1]$.

In cartpole, this additionally ensures that both signals had equal impact on the computed robustness. Since this case only has one input signal, that is not a factor. The expected improvement in estimator training remains, though, so we still perform a normalization step.

By default, the environment limits the car's position to the interval $[-1.2, 0.6]$. We normalize this to the interval $[-1, 1]$ by adding 0.3 to every value and then dividing by 0.9.

To accommodate these changes, we must modify our STL formula. When applying this computation to the constant 0.45, the result is $0.8\overline{3}$. We round this value to three decimal places in Formula 5.6.

$$\rho = \Diamond_{[0,50]}(p - 0.833) \tag{5.6}$$

5.1.2.4 Margin

In the cartpole environment, the environment continues until either a time limit expires, the pole falls over or the cart moves away from the center. Episode termination in mountain car is similar (though the outcome in case of timeout is failure, rather than success). There's a time limit, wherein the car must reach the goal position. If the goal is not reached in time, the episode ends in failure. The environment ends in success whenever the goal position is reached.

This means that the data will be negative for a vast majority of the time. Since an oscillating function is desirable for the training process of our estimator, we introduce a more relaxed boundary for a positive robustness value. We will set the boundary such that the robustness becomes positive once the car has reached the halfway point.

The initial value for the car's position in the mountain car environment is randomly set to a value in the interval $[-0.6, -0.4]$. We will compute using the middle of that interval, -0.5. Since we modified the range of the input values, we must similarly scale this value in order to match it to our current input.

Using the scaling method discussed in Section 5.1.2.3, the starting value -0.5 maps to $-0.\overline{2}$. We can then compute the distance from the starting point to the goal position as $\frac{0.8\overline{3}+0.\overline{2}}{2} = \frac{1.0\overline{5}}{2}$. The midpoint between the start and the goal position, then, is $-0.\overline{2} + \frac{1.0\overline{5}}{2} = 0.30\overline{5}$, which we round to 0.306.

$$\rho = \Diamond_{[0,50]}(p - 0.306) \tag{5.7}$$

The final formula we use for the computation of the robustness for the mountain car environment then becomes what we see in Formula 5.7. Using this formula to compute the robustness for an agent trained on the mountain car environment should result in a suitably oscillating function, which will enable training the estimator to achieve reasonable accuracy.

5.2 Generation

For both environments previously discussed, we train an agent over the course of 1000 episodes. From this training process, we have a stored training log which contains the observations of every step along the way.

First, we pre-process the data set into something our STL robustness tool can process. This occurs in the file *examples/convertTrainLogToRobustnessTimeSeries.py*, located in our GitHub repository [37]. We take the set of outputs generated by the environments during training and convert all observations for a specific episode into `Signal` instances.

We limit the observations to only the signals actually used in the formula, to simplify the problem. The formulas used to compute the robustnesses for each set of signals are the formulas we have constructed in this chapter. This means that we construct two `Signal`s per episode for cartpole and one per episode for mountain car.

For these input signals, we compute the associated robustness signal (using the formulas discussed above) using the improved STL robustness computation tool. This computation uses offline semantics; it computes the entire robustness signal (one per episode) in one go. We later split these up into the correct intervals for each point in the robustness signal to mimic an online context for our estimator.

From the 1000 episodes, then, we create a data set that is close to 200000 samples in size for both environments (in case of cartpole it's around 175000, a little over 200000 for mountain car). We will then use this labeled data set to train our estimation tool.

6 Learning a Robustness Estimator

6.1 Introduction

As discussed in previous chapters, algorithms exist to compute the robustness for any given STL formula. These algorithms work well in offline contexts. In online contexts, when a full trace is not yet available, the algorithms are less efficient which leads to performance problems in during online STL robustness monitoring. In this chapter, we introduce an estimator for the STL robustness value suitable for online monitoring using an ANN.

We consider a window of size w large enough to encompass the data required to perform the computation. That is, w is equal to the size of the combined time interval of the STL formula we are attempting to estimate the robustness for.

Since signals do not necessarily have consistently sized increases in time between sample points, a window of a particular size w does not necessarily contain a constant amount of checkpoints. Because of this, we cannot easily define our estimation tool using a simple fully-connected feed-forward neural network (at least, not without significantly padding the input to make it consistently sized). In that kind of architecture, the size of the input must be constant. It is part of the definition of the neural network.

Instead, we consider an alternative architecture that does not come with this restriction: a Convolutional Neural Network (CNN or ConvNet). This is an architecture that uses kernel convolution operations for, essentially, a sliding window on the input to obtain its results. The exact mechanism is detailed below.

This allows it to easily process input of varying sizes. Operations can reduce the size of the input, but that reduction is always relative. We cannot simply force a convolutional layer to return an exact amount of values. Luckily, pooling functionality exists that does exactly that.

On the output end of our network, though, we do want a static amount of outputs. A fully convolutional network does not really make any such guarantees; when given a larger input, the application of a convolutional layer will produce a larger output. Directly linking a fully connected layer with a convolutional layer, unfortunately, runs into the same problem we discussed previously: for the fully connected layer, we must know the size of the input.

To achieve the consistency we need, then, we employ the adaptive pooling We will be combining a CNN with a fully-connected feed-forward neural network through the use of an adaptive pooling layer, as defined in PyTorch [42]. These adaptive pooling layers create an output of specified dimension, which allows us to construct the fully-connected feed-forward layers appropriately.

6.2 Kernel Convolutions

The convolution operation used by CNNs is based on kernel convolutions as used in image processing. Image filters can be implemented using them. We will use an example to illustrate the operation.

Consider the 4-by-4 'image' (represented as a matrix) I in Equation 6.1. Further, consider the 2-by-2 convolution matrix (kernel) M in Equation 6.2. Applying this matrix to an image causes a blurring effect. In this case, specifically, it's a box blur.

$$I = \begin{bmatrix} 1 & 2 & 2 & 1 \\ 2 & 4 & 4 & 2 \\ 2 & 4 & 4 & 2 \\ 1 & 2 & 2 & 1 \end{bmatrix} \tag{6.1}$$

$$M = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \tag{6.2}$$

Applying the convolution operation specified by M over the image I then involves *sliding* the matrix M over the input image I and performing a computation at each step.

Take the central element of matrix M as the *origin*. For all elements in M, find their relative offsets from the origin. Assume, without loss of generality, that we start iterating over I at location $I_{1,1}$. Using the relative offsets we have computed, we can match each element of M to an element of I.

Between each of these element pairs, we do a multiplication (in this case: $M_{i,j} * I_{i,j}$ with $i, j \in \{0, 1, 2\}^2$). All of these results are then summed together and the total is divided by the sum of elements in M. This, in essence, sets any value to be the average of itself and its surroundings (when using a matrix M that is entirely ones — different effects can be achieved by modifying the contents of M).

Using a matrix M, with odd dimensions, where only the central element is 1 and all others are 0, implements an identity function for the image I. Many other image filter operations can be implemented in this way, too.

Fully performing the computation, the result matrix is given by R in Equation 6.3. Noteworthy is the reduction in size. By limiting the sliding to the known values, we cannot compute this operation for every value in I. A solution to this problem would be to introduce *padding*.

$$R = \begin{bmatrix} 2.\overline{7} & 2.\overline{7} \\ 2.\overline{7} & 2.\overline{7} \end{bmatrix} \tag{6.3}$$

A zero-padded version of the convolution operation, using convolution matrix M and image I would result in R_2 as in Equation 6.4.

$$R_2 = \begin{bmatrix} 1 & 1.\overline{6} & 1.\overline{6} & 1 \\ 1.\overline{6} & 2.\overline{7} & 2.\overline{7} & 1.\overline{6} \\ 1.\overline{6} & 2.\overline{7} & 2.\overline{7} & 1.\overline{6} \\ 1 & 1.\overline{6} & 1.\overline{6} & 1 \end{bmatrix} \tag{6.4}$$

Other changes that might be made to modify the obtained output is to slide the matrix M around differently (e.g. moving over 2 elements at a time, rather than one). The way the kernel slides around is known as the *stride*.

In the case of convolutional neural networks, typically the padding, kernel size and stride are configurable hyperparameters. The trainable parameters for the network are the weights stored in the kernel M. Through the re-use of parameters for all parts of the input, CNNs dramatically reduce the amount of parameters in the model.

6.3 Constructing the estimator

We construct an architecture that performs well in both of our test cases. The architecture is almost identical between them (only a minimal change for input compatibility is required).

While the architecture suffices for our test cases, we do not claim that this architecture will generalize to any STL formula. We expect that the structure (i.e. convolutional layers, followed by pooling, followed by dense layers) can be adapted to apply to any STL formula, but the architectures we present will be insufficient for complex applications.

More complex neural network architectures should generalize to more complex situations, though. A number of factors are likely to significantly influence the required complexity of the estimator model. We enumerate a few of them.

An STL formula can contain timed operators, which feature some interval I that specifies the interval over which this operator is relevant. When the interval I is relatively large, the operation interacts with more data. Larger intervals can thus result in significantly more complex operations. This is a factor that must be considered in estimator design.

Next, a similar increase in computation complexity is expected when for some constant interval I, the sampling frequency is increased. When the input samples are generated more quickly, more data points fall within I. Doubling the sampling frequency (with a constant interval) is expected to result in roughly the same increase in complexity as a doubling of the size of the interval (with a constant sample rate).

The amount of signals used in the STL formula is also likely to be a significant factor. Additional signals means there is additional data to be considered, which complicates the input processing for the model. Since there is more input to consider, more complexity in the model is likely necessary to produce similar results.

Lastly, the amount of operators in the formula can have a large effect on the necessary model complexity. The operators in an STL formula define the target function for the estimator. When more operators are introduced to an STL formula, the way the computation combines the various input signals is likely to be more complex. Through this increased complexity, it is likely that the operation count will have a significant impact on the required complexity of the estimator model.

The shape of the robustness function is likely also a factor. If the robustness function is, for example, a function of one input signal and some linear transformation of that signal, approximating it should be near trivial. However, a function that shows a lot of oscillation over its domain (and other non-linear properties) will likely be harder to approximate for the network, likely resulting in the need for modifications to the architecture for any sufficiently complex formula. This is not entirely separate from the previous point (since fewer operators typically define a simpler function), but not entirely.

As an example, compare the formula ψ to $\psi' = -2 \vee (\psi \wedge 2)$. ψ' contains two extra operators. These operators, though, limit the output range of the robustness function to the interval $[-2, 2]$. If the robustness function defined by ψ oscillated heavily outside this range, the resulting robustness function would likely be simpler to approximate (and thus a good approximation would likely be achievable using a simpler model).

The factors mentioned *can* result in an increase of the required model complexity, but they may also not (or perhaps even simplify the required complexity, as in the last example). However, they are the factors that must be considered as influencing the necessary complexity of the estimator model. We cannot make any statements about the required complexity of the model in general — it is implementation dependent.

6.3.1 Convolutional part

CNNs are typically built up of layers consisting of three components: the convolution operation (as in Section 6.2), a pooling function and an activation function. For the activation function, we use the ReLU function (previously defined, Definition 5.1.1).

The other two components differ between the two convolutional layers we use.

The first convolutional layer contains the only difference in architecture between the cartpole and the mountain car estimators: since the cartpole formula requires data from two signals, its input has two input channels. The mountain car formula, on the other hand, only contains one signal (and thus, one input channel).

For both, the first convolution operation has an output of two channels. We use a kernel size equal to 10, a stride 1 and no padding for the first convolution operation. Following that operation, we apply a max-pool with kernel size 5 and stride 2, followed by the activation function.

Explicitly, the only difference between the two architectures is that the first convolution operation is defined as *Conv1d(2, 2, 10)* for the cartpole environment, whereas it is *Conv1d(1, 2, 10)* for the mountain car environment. From there on out, the architectures are identical.

For the second layer, we use a convolutional layer with 2 in- and output channels, kernel size 5, stride 1 and no padding, followed by an adaptive max pool (outputting 25 values, on 2 channels - so 50 in total) and the activation function again.

Then, we pass the two channels through a flattening operation. This turns the 2 channels of 25 values each into a single channel of 50 values, which is where the fully connected part of our architecture begins.

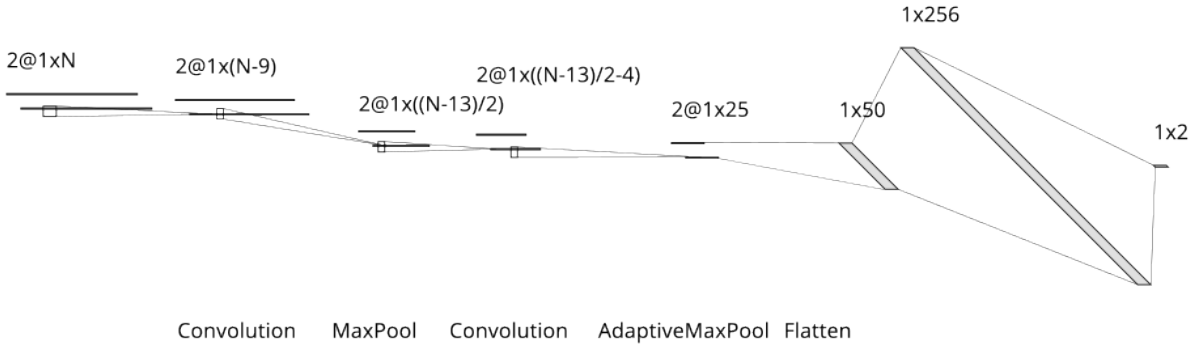

Figure 6.1: Neural network architecture for the Robustness Estimation tool. Sizes shown with an example input of 100 sample points in the window (sizes before the AdaptiveMaxPool operation are variable).

6.3.2 Fully-Connected part

Similarly to the convolutional segment, there also exists a set of components that is commonly used to define a fully-connected feed-forward layer in a neural network. In this case, that set has two components: the actual operation (which in essence is just a matrix of weights and an accompanying matrix of biases) and an activation function (in this case also the ReLU function).

We define two fully connected layers. The first has 50 input features (as required to match the amount of values forwarded from the convolutional part) and 256 output features. After computing the activation function, these values are forwarded to the next layer.

The final layer of our architecture has 256 input features (matching the output features of the previous layer) and 2 output features. These 2 values are put through a separate output activation function. We show the full network architecture (omitting activation functions and without considering mini-batches) in Figure 6.1.

6.3.2.1 Output activation function

The output activation function we use is directly influenced by the way we interpret the network's output; the two output values of the neural network are interpreted as parameters to a normal distribution (with the first output value being the mean and the second the variance).

The loss function we use requires a variance that is strictly greater than 0, so we implement an activation function to achieve this (in a way that preserves the internal graph, such that back-propagation is not disrupted) [42]. Any output the network gives is passed through this activation function so that the network output always satisfies that requirement.

We leave the first output value unmodified, there are no restrictions on the value of the mean of a normal distribution (other than that it must be a real number, in this case).

For the second output value, though, we impose a restriction. The loss function we use (discussed later, Section 6.4) requires a strictly positive variance. Without any activation function, though, this second output value might be any real number.

Through PyTorch's implementation of the *square* function, we can square the second output value quite easily (without disrupting gradient propagation). Trivially, this guarantees a non-negative result. However, strictly positive is required. To achieve this, we add a small value *eps* to the result of the square function: $eps = 1 * 10^{-6}$.

6.4 Training the estimator

As discussed in Section 5.2, we have two large labeled data sets to train our robustness estimator on (one per environment). The labeled data set was stored as `Signal`s. Per episode, we compute (for every robustness value) which subset of the input signals was relevant to its computation. We split the signals, combining exactly those subsets of the input signals with each robustness value. Then, we convert the `Signal` instances into PyTorch *tensors*.

The estimation tool does not explicitly consider the time stamps that are associated with sample points. One requirement, though, is that the times in both input signals is identical (achieved through the computation of comparable signals previously discussed). Rather, the input to the estimation tool should simply be the window of values over which the computation must occur. Given that input, it will then produce an output distribution.

The training process between both environments is very similar. For both environments, we consider a set of slightly over 4000 samples as the control (testing) data set. The other samples are used in the training process.

We consider every sample (in the training set) multiple times, each iteration over the entire set is called an epoch. The order in which the samples are considered is randomized every epoch. During training, we use mini-batches of size 256 (the final mini-batch may contain fewer samples).

The training process is repeated for 200 epochs. Throughout the training process, we use the Adam optimizer to optimize our model parameters [44], using Negative Log-Likelihood loss and a learning rate initialized to $1 * 10^{-5}$. The Negative Log-Likelihood loss function is implemented in PyTorch, as `GaussianNLLLoss`. `GaussianNLLLoss` requires specifying an *eps* value to avoid division by zero; we use the same value to set this parameter as we do in our output activation function (to guarantee that the output distribution variance is strictly larger than 0), which is $eps = 1 * 10^{-6}$.

The implementation of the training process for both estimators is available in our GitHub repository [37] as an IPython notebook. Specifically, they are the files *trainMountainCarRobustnessEstimator.ipynb* and *trainCartpoleRobustnessEstimator.ipynb*.

For the training process, to avoid entirely random behavior, we seed the torch random number generator manually. There are still some other non-deterministic behaviors in the framework [42], which we do not control for. Their effect should be smaller than that of the weight initialization, though.

The models that have resulted from our training are available in the repository as pickled objects [37]. In each of the notebook files, sections are present to load the model from these files and to allow easy verification of our results. Due to size limits in GitHub, the processed training data files are not stored in the repository; the processing scripts to convert them into the required format are from the Open AI gym outputs (and those outputs themselves) are, though. Those files can be found under *examples/cartpole* and *examples/mountaincar* respectively.

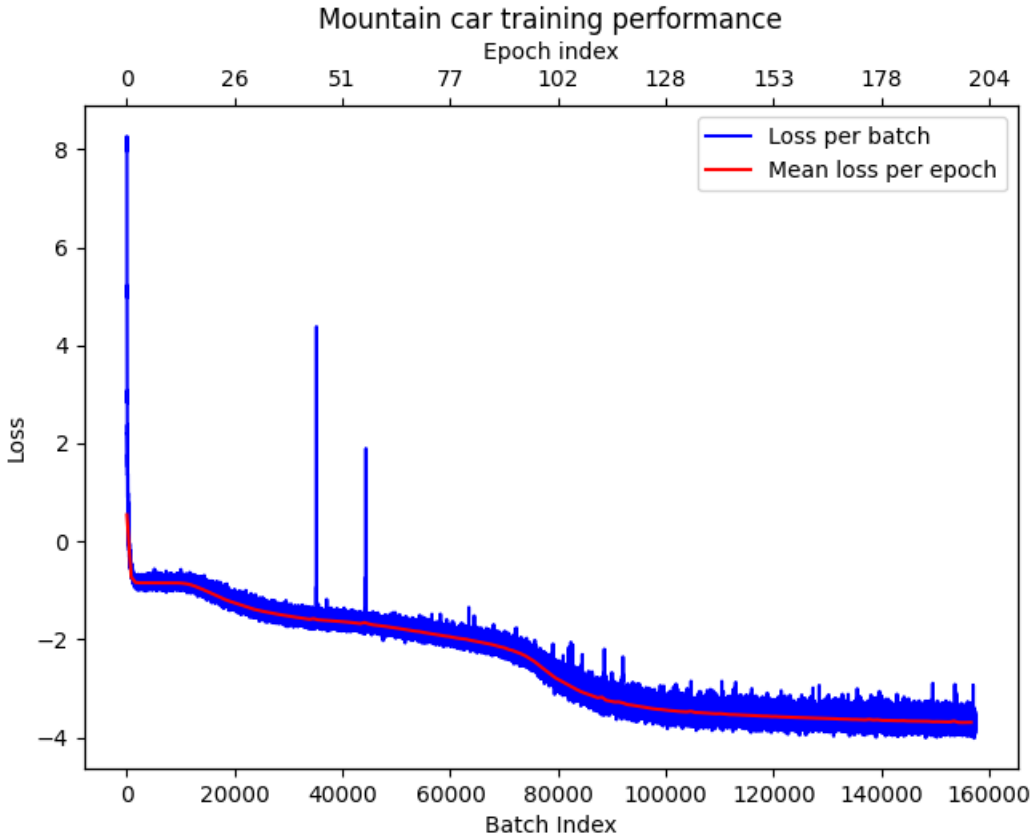

Figure 6.2: Estimator's loss during training for the mountain car environment

6.4.1 Mountain Car

We first consider the mountain car environment. The formula for this environment results in a robustness function that, intuitively, should be easier to approximate than the robustness function defined by the cartpole formula. We expect training for this data set to either converge faster than in the cartpole case or for the approximation to perform better, or both.

The loss obtained by the estimation tool through the training process is visualized in Figure 6.2.

During training, loss is quickly reduced from the initial (large) value obtained through the random weight initialization. After the initial, dramatic drop-off, progress is much slower. We run 200 epochs in this environment, over a little more than 200000 data samples, which results in 157400 batches of size 256 (of which up to 200 batches with smaller sizes), as we can see in Figure 6.2.

Based purely on this graph, we can see that the estimator fairly quickly learns an initial approximation that dramatically reduces the loss function. Later improvements, though, are less dramatic.

Intuitively, this makes sense: when the approximation of the target function is bad, it is easy to make large corrections. As the approximation improves, it becomes harder and harder to make adjustments that quickly improve performance.

As a bit of context to these values: the absolute minimum value the loss function can obtain

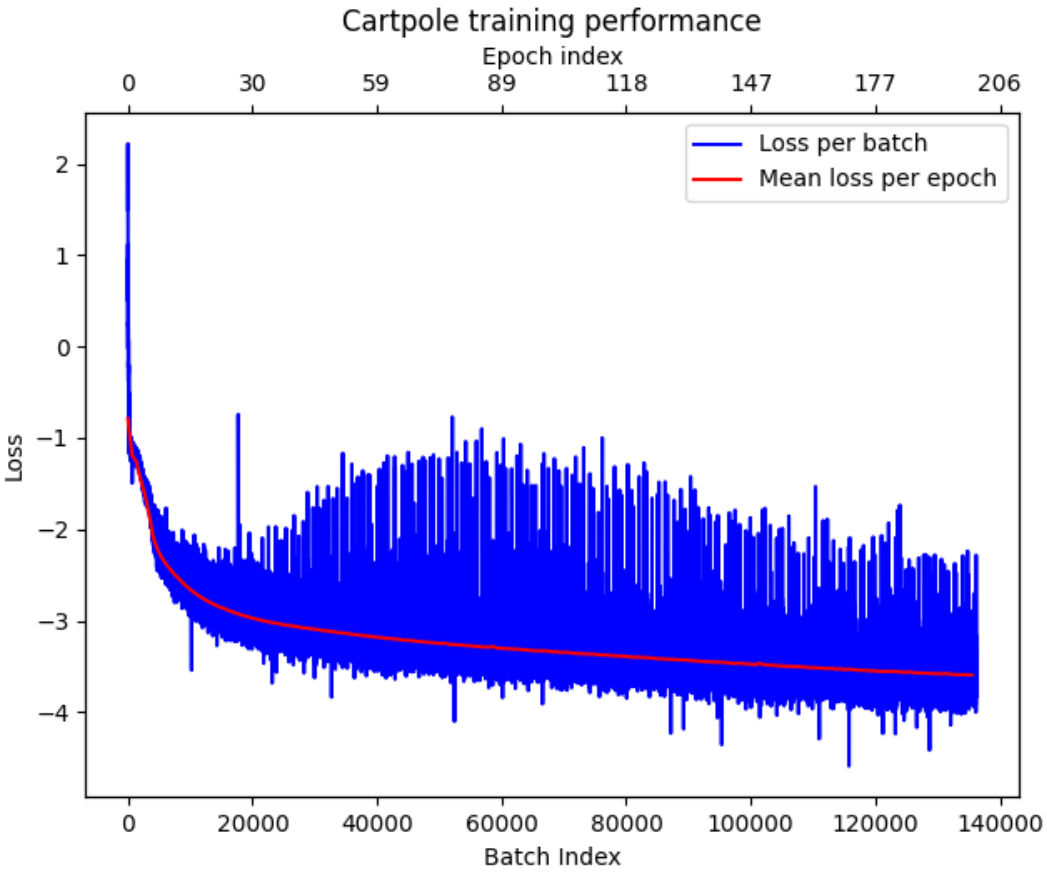

Figure 6.3: Estimator's loss during training for the cartpole environment

given the configuration with $eps = 1 * 10^{-6}$ is -6.9078. This value is obtained when the mean is exactly equal to the expected robustness value and the variance is eps. Our loss, after training, seems to settle around a value a little higher than -4.

We expect a reasonably good, though noticeably imperfect, approximation based on that loss value, but it is hard to make any definitive statements based on this information alone.

We detail the approximation quality on the test set in Section 6.5.1.

6.4.2 Cartpole

Moving on to the cart pole training, then. Considering the increased complexity in the STL formula underlying the robustness computation, we expect the function in this case to be harder to approximate. We expect that to translate as slower training convergence or worse overall performance (or some combination of the two).

We show the loss throughout the training process for the cartpole robustness estimator in Figure 6.3.

In this environment, similarly to mountain car, we let the estimator train for 200 epochs. Since we have approximately 170000 sample points in the training data set and are using a batch size of 256 samples per batch, this translates to the evaluation of, overall, a little over 132000

Offset from mean	Count
1σ	3858
2σ	523
3σ	13

Table 6.1: Overview of how far (measured in multiples of the standard deviation) the expected robustness is from the predicted mean value in the validation data set for the mountain car estimator

batches (up to 200 of which may be smaller than 256 sample points in size).

Similarly to before, we see a fast initial improvement that dramatically slows as training progresses, which is expected behavior. The minimum possible loss value remains the same as in the mountain car section, at -6.9078, while the estimator's loss near the end of its training run seems to have settled around a value a little under -3.

This value is slightly higher than the value we saw in mountain car, so later when we are comparing the results, we may expect slightly worse performance in the cartpole environment than in the mountain car environment, as predicted.

6.5 Results

6.5.1 Mountain Car

We show the results of our estimation tool over the data we kept separate as a validation set in Figure 6.4. By the graph, we see that the estimation always follows the rough shape of the function.

The true robustness, for the most part, appears to be within one standard deviation of the estimated value. The worst cases appear to be within two standard deviations, visually. We verify this through an exhaustive comparison. The results are shown in Table 6.1.

This mostly matches our visual observations of the graph — though there are apparently 13 sample points where the actual robustness value is more than 2 standard deviations away from the estimated mean.

Another interesting metric to consider is the sign of the mean compared to the sign of the expected robustness. Since the sign of the robustness tells us if a certain set of inputs satisfies a formula or not, it is an important aspect of the result.

Differences in magnitude between the estimated and the actual (expected) robustnesses often are not a significant issue; a difference in sign, though, may lead to problems (since then a system may think the specification has been violated when it has not, or vice versa).

When we compare the sign of every predicted mean value for the robustness distributions to the sign of the expected robustness for that output, we find that there are 50 cases where the signs do not match. 50 mismatches, out of a validation data set containing 4394 samples, is a classification accuracy of 98.9%.

For the cases where the sign was incorrectly predicted, the mean magnitude of the predicted mean for the robustness distribution was around $2 * 10^{-3}$. The largest magnitude where the

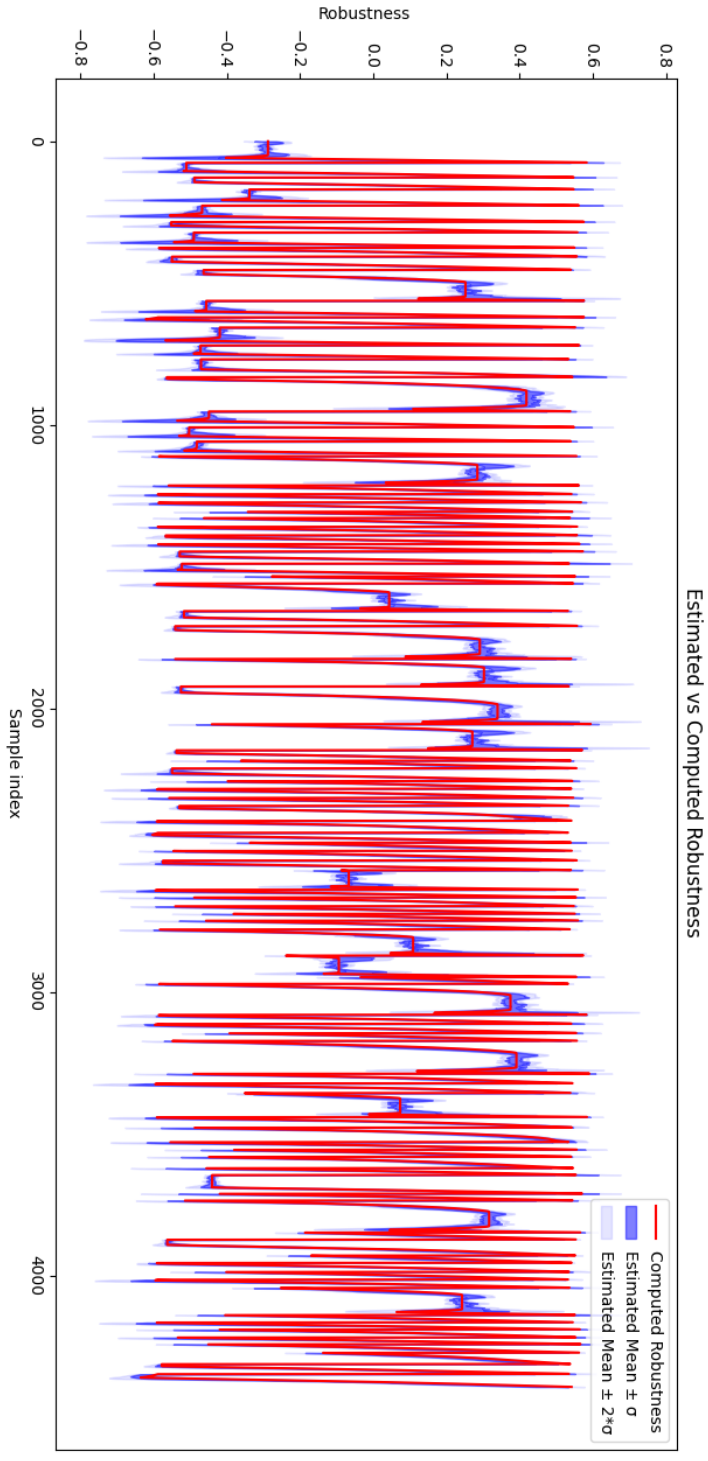

Figure 6.4: Estimator performance during testing for the mountain car environment

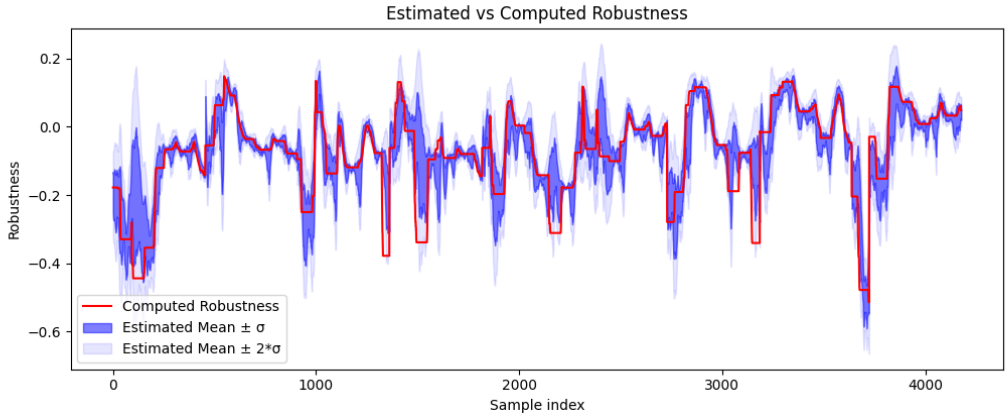

Figure 6.5: Estimator performance during testing for the cartpole environment

sign was incorrectly encountered was a predicted mean of $7 * 10^{-2}$.

Since these are relatively small values, we assert that using the magnitude of the predictions as indicators of the necessity of algorithmic robustness computation should result in reasonable certainty that the STL formula is not violated (at least in this case — results may differ in other situations).

6.5.2 Cartpole

For cartpole, then, we show the results in Figure 6.5.

Here, too, we see that the estimation roughly follows the shape of the function, which is a good sign. However, there are some notable exceptions. Right near the front, for example, there is quite a spike in the estimated robustness that does not exist in the actual computed robustness.

In that case, the expected mean does not actually cross the 0-boundary, so it does not necessarily indicate a problem. However, when the accuracy is lower, as it appears to be in this case, we do expect that such cases will be more frequent.

In Table 6.2, we take a look at the offsets from the mean for the expected robustness values in our validation set. The overall performance is notably worse than we saw in the mountain car environment.

There is a significant increase in amount of sample points that are multiple standard deviations removed from the mean, which clearly means that the quality of the estimated distributions has deteriorated.

Using the amount of standard deviations a robustness value is away from the predicted mean, as in Table 6.2, is an imperfect metric, since a prediction involving a very large standard deviation would almost always fall within the first category — even if the estimated robustness is significantly different from the estimated mean.

Luckily, the magnitude of the robustness is not vital to the functionality of our tool. By taking a closer look at how well the sign of the estimated means matches the sign of the expected robustness, we show that the accuracy our estimator achieves in classifying formula satisfaction

Offset from mean	Count
1σ	3066
2σ	920
3σ	137
4σ	42
5σ	8
6σ	2
7σ	3

Table 6.2: Overview of how far (measured in multiples of the standard deviation) the expected robustness is from the predicted mean value in the validation data set for the cartpole estimator

has similarly deteriorated.

In this example, out of the 4178 sample points, 314 have a predicted mean with a sign that does not match the computed robustness value. This amounts to an accuracy of approximately 92.5%. The highest magnitude of estimated mean when the sign is incorrect was $1.3 * 10^{-1}$, while the mean for those cases was $4 * 10^{-2}$.

While this accuracy is still sufficient for the estimation tool to be useful, these figures are a significant deterioration from the same numbers seen in the mountain car environment. Clearly, the cartpole estimator performs significantly worse. In the next section, we consider an expanded architecture to address our hypothesis that more complex models may more easily model more complex robustness functions.

6.6 Upgraded Cartpole Estimator

We repeat the previous training process for the cartpole environment, with a more complex estimator.

Previously, relative to the size of the input the model has to process, the model for the mountain car environment was more complex; it is constructed almost identically, but receives only half the input (a single signal, instead of 2). We hypothesize at this point that a similar increase in complexity in the cartpole estimator model would improve its performance.

We modify the architecture of the estimator as follows: we increase the amount of channels in the convolutional layers from 2 to 4 (other than the initial input channel count). The other aspects of the convolutional layers remain unmodified.

To handle the amount of data resulting from the increased channel count, we must similarly increase the size of the first dense layer. It will now expect 100 input features, where previously it expected 50.

Other than the modifications described, the network rest of the network architecture remains unchanged: the first dense layer still has 256 output features, the final layer still has 2 output features.

In essence, the overall network structure is unchanged compared to what we showed in Figure 6.1, we have simply increased the amount of channels in the convolutional section to 4 and propagated the doubled size throughout the model. The first dense layer is enlarged to handle

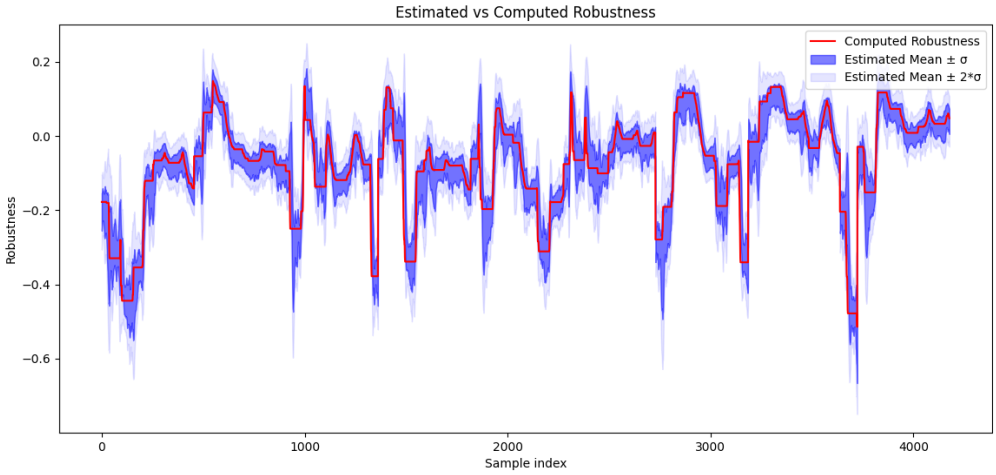

Figure 6.6: Estimator performance during testing for the cartpole environment, using the upgraded architecture

Offset from mean	Count
1σ	3510
2σ	634
3σ	33
4σ	1

Table 6.3: Overview of how far (measured in multiples of the standard deviation) the expected robustness is from the predicted mean value in the validation data set for the improved cartpole estimator

100 input features to accommodate the extra data being fed into the dense section after the flattening operation. From that point onward, the dense section of the neural network is identical to the previous version.

The performance of this modified architecture is an improvement that brings its performance more in line with what we see in the mountain car environment.

Recording training loss for this estimator results in a figure that is very similar to the previous two, so we do not include it. It seems to finally settle in at a loss, at the end of training, that is a slight improvement over what we previously saw in the cartpole environment, right around -4. Since this value is very close to what we saw in mountain car, we are hopeful about its results.

In Figure 6.6, we show the results the improved architecture obtains when estimating the robustness for the cartpole environment.

Compared to Figure 6.5, there are notably fewer cases where there are significant discrepancies between the computed, expected robustness and the result of our estimator.

To further illustrate this, we consider the same metrics we did previously. Table 6.3 contains the summary of the differences between the estimated mean and the expected robustness in terms of the variance.

The results in this case are closer together again. The highest offset is now limited to 4σ, rather

than 7σ which is a dramatic improvement. The distribution of the values over the categories also seems to lean more closely to what we saw in the mountain car environment (which we assume is more desirable, since it was performing better).

From the output results, we further compute how many of the estimated means do not have the same sign as the expected robustness value. In this case, that is 96 sample points; this corresponds to a classification accuracy of 97.7%. While that is not quite the accuracy we achieved in the mountain car environment, we believe the improvement sufficient to show that minor increases in estimator complexity can offer significant benefits in obtained function approximation.

In the cases where the sign of the estimated mean of the robustness distribution does not match the expected robustness value, the mean magnitude of the predicted distribution means was a marked improvement over the previous cartpole estimator, but not quite as good as what we saw in the mountain car case, at $5 * 10^{-3}$.

The maximum, similarly, was better than the previous cartpole case but not as good as what we saw in the mountain car environment. We find a maximum magnitude of incorrectly signed mean equal to $9 * 10^{-2}$.

6.7 Performance

We introduced the estimation tool as a method of getting an indicative robustness value for a given input trace and a given STL formula. Since a fully accurate method of computing this value exists, there is no benefit in this tool unless it can perform this computation significantly faster.

The main downside of the existing algorithm (that can guarantee correctness of the computed value) is that it can have a major impact on the runtime performance of the system being monitored. The estimation tool we present can mitigate that performance hit. By using the estimation tool to compute a 'quick & dirty' value, and only performing the full computation in case the estimated value indicates a potential problem (by, for example, predicting a significantly lower mean or higher variance), online STL monitors can improve their performance by omitting unnecessary robustness computations.

This workflow relies on the fact that our estimation can be computed much quicker than the robustness computation itself. Since we have already introduced significant improvements in the performance of the robustness computation (Section 3.4), it is necessary to show that this model offers further improvement.

All performance statistics were gathered on a Ubuntu 20.04 LTS system, using an AMD Ryzen 5900x CPU and an Nvidia RTX 3080 GPU, both running at stock configuration. Throughout testing, lightweight monitoring software was used to ensure no unexpected CPU or GPU down-clocking occurred. No applications likely to result in a measurable impact on performance were running on the system during these measurements. All machine learning applications were executed on the GPU, other applications used the CPU.

6.7.1 Performance Comparison Setup

We compare the performance both for the mountain car and cartpole environments. The point of comparison for both will be the efficient algorithm we have previously introduced (Chapter 4). Since we have presented two architectures for cartpole estimation, we will take the more complex one for this performance comparison (since it is likely to be slightly slower).

Because we have a large data set that we have trained the estimator on, pre-processed using the efficient algorithm, we re-use that data set to compare the performances. We expect that these performance statistics generalize well to other cases.

For the estimator tool performance statistics, we provide a cell in the notebooks that will run the performance analysis. The files in question are *trainCartpoleRobustnessEstimator.ipynb* and *trainMountainCarRobustnessEstimator.ipynb* (located in our GitHub repository [37]).

For the efficient algorithm, enabling the performance analysis requires a little more manual action. We have previously used the file *examples/convertTrainLogToRobustnessTimeSeries.py* to turn a raw log of Open AI gym outputs in to Signals with an associated computed robustness. A few lines of code in this file are commented, with an accompanying comment stating they are relevant for the performance computation.

To reproduce the results presented in this section, these lines must be uncommented (and, to reduce unnecessary wasted time, a few others should be commented). Specific instructions for which lines to comment and uncomment for various analyses are present in the file.

6.7.2 Online Robustness

To simulate an online environment, we evaluate each method through the computation of one sample point of the robustness signal at a time. This is analogous to an online environment because when a new input sample point is received, a single new output point can be computed.

This is what happens naturally for the estimation tool we have presented; every data point is computed individually using input tensors containing exactly the window of relevant data. In order to meaningfully compare the performance of this tool, we must ensure that anything we compare the estimator to is working under similar restrictions.

We achieve this through a pre-processing of the data sets. During the generation of our data set, in order to generate all the labels, we used the offline version of the algorithm. We passed the entire output of an episode into the algorithm, which computed a robustness signal that contained multiple sample points.

From that robustness result and the input signals associated with it, we then generated a set of input signal subsets associated with a single output robustness value. When working this way, the performance of the efficient algorithms we have discussed in Chapter 4 is quite acceptable.

In order to emulate an online context, we instead split the input signals into the respective subsets before passing them into the algorithm. In this way, each call of the algorithm results in a single output robustness value. This change means that a lot of the optimisations in the algorithms that make them ideal for offline contexts are no longer as effective, so the performance drops quite dramatically (this can be seen by comparing the results in Table 6.4 and Table 6.5).

In Table 6.4, we compare the performances (for both environments) between the estimators (the

	Cartpole	Mountain car
Efficient Algorithm	879.98	214.38
Estimator	64.36	72.26

Table 6.4: Time taken (in seconds, rounded to nearest hundredth) to compute the robustness for the entire data set (178280 samples for cartpole, 205731 for mountain car) when computing as if running in an online context (i.e. one result sample point at a time)

improved/complex one, in the case of cartpole) and the algorithms (efficient algorithm, in both cases - based on previous analyses, we know that the syntax algorithm would simply perform worse and add no value to this analysis).

The performance difference between the estimators is minimal. If we normalize for the size of the sample set, the difference is smaller than $1 * 10^{-5}$ seconds per sample. Since their architectures are very similar, this is not unexpected. The one major factor influencing their performance would be the difference in architecture. These should, intuitively, not result in significant performance degradation since the architectural changes made are relatively limited.

A number of extra parameters that must be evaluated are introduced in the more complex cart pole architecture, however the parallelization allowed through GPU evaluation should be more than sufficient to prevent a significant decrease in performance.

For the algorithm cases, the difference in performance is very much *not* minimal. The computation for the mountain car data set is roughly 4 times faster than the computation for the cartpole data set.

The cartpole formula contains 4 *always* operators, and 3 *and* operations. In this context, each of the *always* operators returns a single value so the complexity of the *and* operations is rather trivial. There are no intersections between signals to consider and the length of the signals to be operated on is minimal. By this logic, the *always* operators are likely to dominate the complexity of the computation. Based on that observation, it makes a lot of sense that the computation for cartpole takes around 4 times as long as the computation for mountain car.

Interesting, of course, is the comparison between the estimator and the algorithm. In both cases, we see a significantly better performance in the case of the estimator. Since the estimator does not fully consider the complexity of the performed operations (and instead performs a computationally simpler approximation), this behavior is expected.

It is also this behavior that means that an estimator is useful. Since the complexity of the STL formula has far less of an impact on the performance of the estimator than on the performance of a traditional robustness computation, we argue that an estimator like we have presented has use in situations where a complex behavior (specified through STL formula) must be monitored on a system with limited runtime resources.

6.7.3 Offline Robustness

The algorithms perform significantly faster if they operate on the entire input signal at a time, so we would like to include performance statistics for this case as well even though that is not the foreseen use-case. Even in these cases, it is not unreasonable to say that the estimator finds its results quicker.

To illustrate, we consider the same data sets as before. This time, though, we will use the data

	Cartpole	Mountain car
Efficient Algorithm	37.38	6.32
Estimator	2.15	2.43

Table 6.5: Time taken (in seconds, rounded to nearest hundredth) to compute the robustness for the entire data set (178280 samples for cartpole, 205731 for mountain car) when computing as if running in an offline context (one episode at a time for the algorithms, batch size 256 for the estimators)

set without the pre-processing step mentioned in Section 6.7.2, instead simply computing the robustness signal one episode at a time, as would be expected in an offline context.

The estimator supports the use of batches. We have used this extensively throughout the training process and this can be used to dramatically speed up the evaluation. When considering an offline use-case, an entire input signal is available. If this signal is appropriately divided into sub-intervals, it can be passed into the estimator in a batch. The estimator, then, will return a 1D tensor that is essentially a vector containing every output robustness distribution. In this way, we can consider the evaluation of batched input as the application of our estimator to an offline context.

The performance improvement resulting from batching holds up to some unknown batch size that we do not further specify, since it heavily depends on the environment in which the estimator is used (for example, a batch size that exceeds GPU memory is bound to cause issues).

In Table 6.5, we show the performance results in an offline context. As before, the performance difference between the two estimators is very small. In absolute terms, the cartpole estimator has performed more quickly again. That is a direct result of the size of the data set, though; the mountain car estimator performs slightly faster per sample (as expected, since it is architecturally slightly simpler).

In case of the algorithms in this case, we see a larger relative difference between the two environments. We assume this is caused by the fact that in this case the AND operations are non-trivial. In the online context, these were operating on single values. Now, they are operating on multi-valued signals. This means that intersections between these signals must be considered, which results in the other operators being less dominant in the overall complexity.

If we assume the *and* operations to be of similar complexity as the *always* operations, there are a total of 7 operations in the formula. Compared to the single operator in the mountain car formula, we would thus expect roughly a 7-fold slowdown which is closer to the almost 6 we actually found. We assume this final difference is due to the fact that *and* is easier to compute than *always* or due to the fact that the *always* operations have slightly shortened the signals.

Comparing the algorithms with the estimators, we see that there again is a rather significant performance benefit for the estimators. The estimator for the mountain car environment fully processes the data set in under half the time the efficient algorithm requires.

For cartpole, the improvement is even more significant. We see a factor 17 improvement going from algorithm to estimator.

	Cartpole	Mountain car
Algorithm	879.98	214.38
Algorithm + Approximator	324.34	112.31

Table 6.6: Time taken (in seconds, rounded to nearest hundredth) to compute the robustness for the entire data set (178280 samples for cartpole, 205731 for mountain car) when computing as if running in an online context as a full STL monitor

6.7.4 Full Monitor Performance

The performance differences discussed in the previous sections, while impressive, do not show the full picture. As we've mentioned, the envisioned use for the approximator is not to entirely replace the traditional computation. Rather, it is to be used as a way to optimize an STL monitor's performance by eliminating unnecessary computations.

In this section, we will construct an example online STL monitor for each of the environments. We will consider a robustness that is within 2σ of zero to be sufficient cause to require the full algorithmic computation. The operation of the monitor is, thus, the computation of the estimate, followed by a check on the mean and variance's magnitude. In case the absolute value of the estimated mean is lower than twice the estimated standard deviation, we compute the robustness using the efficient algorithm.

The exact criteria used to make the decision if the full robustness computation is necessary will significantly impact the performance improvement one can see using the estimator. Since these criteria are implementation dependent, we cannot make conclusive statements about the real-world performance of the system in this work. While we provide indicative results from our examples, and we hypothesize that similar results are possible for other use-cases, we cannot guarantee that.

The statistics in the top row of Table 6.6 are identical to those previously seen in Table 6.4. We repeat them for clarity.

Notable in the results shown in Table 6.6 is that in this case the performance difference between the two estimator variants is significant. In previous results, the estimators were always very close together.

Since the estimators have not changed since the previous online context result, that leaves two factors that may influence this difference. One is the amount of times the robustness computation is necessary, the other is how long each of these computations takes.

It is a combination of these factors that causes the difference in results we see in the bottom row of Table 6.6. Trivially, since the computation is much harder for the cartpole environment (as is clear from the algorithm results in the top row), the impact on the cartpole monitor is also larger per computation.

Additionally, in the mountain car environment around 10000 robustness computations occur over the entire data set. In the cartpole case, we see around 30000 robustness computations. Since the metrics for our upgraded cartpole estimator were still worse than the metrics for the mountain car estimator, this is not unexpected. We believe that if further improvements to the model (or the training process) are made, the results we present can be easily improved upon.

For both scenario's, though, a significant overall performance gain is noted between the simple online STL monitor and the monitor-estimator combination in Table 6.6. In a simple environment,

such as mountain car, the addition of a robustness estimator allows a factor two improvement in overall time spent computing the robustness. The more complex environment, even with an estimator that works less precisely, achieves an even better result. The runtime for the robustness monitor, augmented with an approximator, is over 2.5 times faster than its counterpart.

7 Conclusion

Runtime verification remains a great tool to ensure systems are behaving in a way that follows their specification. STL, as a formalism, is well-suited to modeling the behavior of reactive systems. Previous work has enabled efficient monitoring of a system modeled using STL in offline contexts, but in online contexts further improvement is desired.

The robustness estimator we have presented is a step toward this goal. Eliminating a lot of the computations through an easy-to-compute estimate of the robustness offers significant potential performance improvements. While fulfilling the prerequisites for the implementation of this estimator, we have further contributed significant code quality & functional improvements to the STL robustness computation tool as well as presented a modification of a fixed-size sliding window maximization algorithm to support variably-sized steps between sequential sample points (and thus a non-constant number of samples in the window).

7.1 STL Robustness Tool

We presented significant performance and general code quality improvements to an existing STL robustness computation tool [18]. Beyond these qualitative changes, we have also made functional improvements; we solve the problems identified in the original paper and any issues further identified through the introduction of a comprehensive test suite.

One of the main algorithmic improvements required to address the issues described in the original work, was a full re-implementation of the timed eventually operation using an efficient sliding-window maximization algorithm [19]. We modified this previously known implementation in a few ways. We remove the minimization functionality that is also included (since it offers no benefit for the timed eventually operation). Additionally, the original implementation works on a fixed-size window, whenever a new sample enters the window, one must leave. If that assumption is not fulfilled, the algorithm fails. We extend the implementation to support a variable amount of checkpoints in the window.

The code quality improvements involved an extensive refactor of the entire code-base. We have described the changes in detail and shown that the performance of the tool has significantly increased following these changes (by comparing the syntax-derived algorithm in the previous implementation to the same algorithm in the current version).

The implementation of previously presented efficient algorithms [7] allowed an additional massive performance increase. In testing, using a formula with a large time interval, we have shown a computation time that is almost 350 times shorter than the computation time of the syntax algorithm for the same formula in the old implementation. The benefit almost entirely disappears in simpler formulas, but since the entire computation time is very low in those cases anyway generally the improvement is less necessary and would be less impactful.

Through the improvements in the robustness tool, we have massively increased its efficiency for

robustness computation in offline contexts. The improved performance in offline contexts has enabled the generation of large robustness-labeled data sets. We create such a data set using the training output of an agent in two different OpenAI environments. This data set, then, is used to show that the robustness function (at least in the cases defined by our STL formulas, though this likely generalizes) can be approximated using an ANN.

7.2 Robustness Estimator

We presented a robustness estimator, implemented as an ANN built up of multiple convolutional layers, followed by multiple fully-connected feed-forward layers.

Using the improved STL robustness computation tool, we compute a large labeled data set of sample inputs. This data set enables us to train this estimator.

Using commonly seen techniques in neural network training, we arrive at an estimator model that achieves impressive accuracy when classification of STL formula satisfaction is used as a metric. The magnitude of the target robustness is closely followed, though deviations from the magnitude do occur. These deviations were expected in our estimator, though, and are not a problem — the pertinent metric is if the STL formula is satisfied.

We have shown that with a reasonably sized data set, a relatively simple estimator model can be trained to achieve a good approximation of the target robustness function. While this approximation is very close in our examples, whether this translates to more complex systems is likely dependent on specific properties of the problem in question.

Training an approximator to be useful for a system that often operates near the boundary condition (where the magnitude of the robustness is low) is much harder than training for a use-case where the robustness is mostly of high magnitude (since a greater degree of precision is required). The amount of time and data required to train an approximator to achieve satisfactory results in the former case may be prohibitive. Furthermore, it is likely that a more complex model will be required which may (slightly) impact the resulting performance.

We have further shown that the computation of this robustness estimate is much faster than the full robustness computation. Additionally, from our results, it appears that while models must be expanded to approximate more complex STL formulas, the increase in runtime required to evaluate the expanded model is less than the increase in runtime required for evaluation of the more complex STL formula.

Of course, the approximator cannot entirely replace the exact computation. The exact computation is able to give guarantees about formula satisfaction, which an approximator cannot. These guarantees are often only required when a system is near the boundary condition of the STL formula, though. Based on the magnitude and variance of the estimated robustness distribution, an online STL monitor augmented with an estimator may decide whether or not the full computation is necessary.

By omitting the computation entirely in cases where magnitude is high and/or variance is low, significant performance gains can be achieved. We have identified two criteria that are likely to influence the amount of performance that can be gained through the implementation of an estimator in the online STL monitoring system. These are the complexity of the system behavior (the STL formula) and the typical magnitude of the robustness during system operation (assuming our criterion for the omission decision is used). More generally, the second factor is

how often the system violates the criterion that allows the robustness computation to be omitted. The possible performance gains will be larger for a system that violates such a criterion less often.

Systems with a complex STL formula should see a more significant performance increase between the computation and the estimator than their less complex counterparts. While we have noted that approximator models may need to be adjusted based on the complexity of the STL formula defining the target robustness function, we have shown that the increased computation time required to evaluate such a model is far less than the increase in computation time required to fully compute an exact robustness value. In this way, the approximator is able to offer more performance gain per evaluation for more complex systems. Since these are likely the systems suffering from performance issues, this is a very desirable property.

Important to note for any system that relies on the performance of the STL robustness monitor, though, is that these performance improvements cannot be consistent (in the current implementation, we note a possible solution later). Due to the nature of this implementation, there will be moments where the traditional computation is necessary to provide guarantees about the computed robustness value. When that is deemed necessary, the computation for that data point is slower than the traditional computation. While faster overall, the time taken is likely to oscillate far more erratically.

7.3 Future Work

We have shown for our test cases that in a relatively simple environment, a doubling of overall performance can be achieved. Further, a more complex test case achieves more performance improvement. We hypothesize that this pattern continues. In more complex environments, the improvement can be higher because the cost of the algorithmic computation is much higher while the estimator's evaluation likely doesn't need as much of a complexity increase.

Further work is required to verify that. For much more complex environments, it is likely that significant changes to the model's architecture must be made (to preserve the quality of the approximation), which may further slow down its performance. While we expect this performance degradation to be significantly less than the performance degradation in the algorithmic solution for an equivalent increase in complexity, we cannot state this definitively at this time.

Extra work regarding the criteria used to determine the necessity of the algorithmic computation is also needed. In our work, we've used a very simple heuristic based on the absolute value of the estimated mean for the robustness distribution and its standard deviation. It is possible that a more sophisticated decision process can further optimize the results we've found. Additionally, we believe it likely that the heuristic we've chosen is insufficient for significantly more complex systems than the examples we've considered.

Lastly, to address the performance degradation that may be encountered on a single checkpoint basis (when the estimation indicates that a full computation is necessary), it is interesting to consider parallelizing these operations. An implementation of an online STL monitor that always starts the algorithmic computation simultaneously with the computation of the estimate would not see this degradation. However, this means that the monitor would require additional computational resources, so there is still a larger performance impact. Further study is required to determine if parallelization of these operations is feasible in the implementation of online STL monitors. Such work should consider the trade-off between the increased load on the computation resources (since starting every robustness computation results in more computations) and the

amount of time that this method saves. The amount of time that can be saved through this method is likely to vary heavily between various use-cases.

Bibliography

[1] A. Dokhanchi, B. Hoxha, and G. Fainekos, "On-line monitoring for temporal logic robustness," in *International Conference on Runtime Verification*, Springer, 2014, pp. 231–246.

[2] A. Cerone, P. A. Lindsay, and S. Connelly, "Formal analysis of human-computer interaction using model-checking," in *Third IEEE International Conference on Software Engineering and Formal Methods (SEFM'05)*, IEEE, 2005, pp. 352–361.

[3] D. Bushnell, D. Giannakopoulou, P. Mehlitz, R. Paielli, and C. Pasareanu, "Verification and validation of air traffic systems: Tactical separation assurance," in *2009 IEEE Aerospace conference*, 2009, pp. 1–10. DOI: 10.1109/AERO.2009.4839621.

[4] X. Leroy, "Formal verification of a realistic compiler," *Commun. ACM*, vol. 52, no. 7, pp. 107–115, Jul. 2009, ISSN: 0001-0782. DOI: 10.1145/1538788.1538814. [Online]. Available: https://doi.org/10.1145/1538788.1538814.

[5] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "Sel4: Formal verification of an os kernel," in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, ser. SOSP '09, Big Sky, Montana, USA: Association for Computing Machinery, 2009, pp. 207–220, ISBN: 9781605587523. DOI: 10.1145/1629575.1629596. [Online]. Available: https://doi.org/10.1145/1629575.1629596.

[6] E. Bartocci and Y. Falcone, *Lectures on Runtime Verification*. Springer, 2018.

[7] A. Donzé, T. Ferrère, and O. Maler, "Efficient robust monitoring for stl," in *Computer Aided Verification*, Springer Berlin Heidelberg, 2013, pp. 264–279, ISBN: 978-3-642-39799-8.

[8] J. V. Deshmukh, A. Donzé, S. Ghosh, X. Jin, G. Juniwal, and S. A. Seshia, "Robust online monitoring of signal temporal logic," *CoRR*, vol. abs/1506.08234, 2015. arXiv: 1506.08234. [Online]. Available: http://arxiv.org/abs/1506.08234.

[9] A. Perotti, A. d'Avila Garcez, and G. Boella, "Neural networks for runtime verification," in *2014 International Joint Conference on Neural Networks (IJCNN)*, 2014, pp. 2637–2644. DOI: 10.1109/IJCNN.2014.6889961.

[10] A. Pnueli, "The temporal logic of programs," in *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, 1977, pp. 46–57. DOI: 10.1109/SFCS.1977.32.

[11] A. Donzé, "On signal temporal logic," in *International Conference on Runtime Verification*, Springer, 2013, pp. 382–383.

[12] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications," *ACM Trans. Program. Lang. Syst.*, vol. 8, no. 2, pp. 244–263, Apr. 1986, ISSN: 0164-0925. DOI: 10.1145/5397.5399. [Online]. Available: https://doi.org/10.1145/5397.5399.

[13] A. Donzé and O. Maler, "Robust satisfaction of temporal logic over real-valued signals," in *Formal Modeling and Analysis of Timed Systems*, K. Chatterjee and T. A. Henzinger, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 92–106, ISBN: 978-3-642-15297-9.

[14] D. Nikovi and T. Yamaguchi, "Rtamt: Online robustness monitors from stl," in *Automated Technology for Verification and Analysis*, D. V. Hung and O. Sokolsky, Eds., Cham: Springer International Publishing, 2020, pp. 564–571, ISBN: 978-3-030-59152-6.

[15] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Ng, "Ros: An open-source robot operating system," in *Proc. of the IEEE Intl. Conf. on Robotics and Automation (ICRA) Workshop on Open Source Robotics*, Kobe, Japan, May 2009.

[16] F. Lorenz and H. Schlingloff, "Online-monitoring autonomous transport robots with an r-valued temporal logic," in *2018 IEEE 14th International Conference on Automation Science and Engineering (CASE)*, 2018, pp. 1093–1098. DOI: 10.1109/COASE.2018.8560421.

[17] X. Qin and J. V. Deshmukh, "Predictive monitoring for signal temporal logic with probabilistic guarantees: Poster abstract," in *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control*, ser. HSCC '19, Montreal, Quebec, Canada: Association for Computing Machinery, 2019, pp. 266–267, ISBN: 9781450362825. DOI: 10.1145/3302504.3313353. [Online]. Available: https://doi.org/10.1145/3302504.3313353.

[18] L. Van Damme, "Signal temporal logic monitoring and online robustness approximation," University of Antwerp, Antwerp, Belgium, 2021.

[19] D. Lemire, "Streaming maximum-minimum filter using no more than three comparisons per element," *CoRR*, vol. abs/cs/0610046, 2006. arXiv: cs/0610046. [Online]. Available: http://arxiv.org/abs/cs/0610046.

[20] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT press, 2016.

[21] Y. Li, "Deep reinforcement learning," *CoRR*, vol. abs/1810.06339, 2018. arXiv: 1810.06339. [Online]. Available: http://arxiv.org/abs/1810.06339.

[22] K. Hornik, M. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators," *Neural networks*, vol. 2, no. 5, pp. 359–366, 1989.

[23] B. C. Csáji *et al.*, "Approximation with artificial neural networks," *Faculty of Sciences, Etvs Lornd Unive rsity, Hungary*, vol. 24, no. 48, p. 7, 2001.

[24] T. K. Ho, "Random decision forests," in *Proceedings of 3rd international conference on document analysis and recognition*, IEEE, vol. 1, 1995, pp. 278–282.

[25] J. H. Friedman, "Greedy function approximation: A gradient boosting machine," *Annals of statistics*, pp. 1189–1232, 2001.

[26] S. Suthaharan, "Support vector machine," in *Machine Learning Models and Algorithms for Big Data Classification: Thinking with Examples for Effective Learning*. Boston, MA: Springer US, 2016, pp. 207–235, ISBN: 978-1-4899-7641-3. DOI: 10.1007/978-1-4899-7641-3_9. [Online]. Available: https://doi.org/10.1007/978-1-4899-7641-3_9.

[27] R. E. Schapire, "The boosting approach to machine learning: An overview," *Nonlinear estimation and classification*, pp. 149–171, 2003.

[28] D. H. Wolpert, "The lack of a priori distinctions between learning algorithms," *Neural computation*, vol. 8, no. 7, pp. 1341–1390, 1996.

[29] M. Fernández-Delgado, E. Cernadas, S. Barro, and D. Amorim, "Do we need hundreds of classifiers to solve real world classification problems?" *The journal of machine learning research*, vol. 15, no. 1, pp. 3133–3181, 2014.

[30] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, *Openai gym*, 2016. DOI: 10.48550/ARXIV.1606.01540. [Online]. Available: https://arxiv.org/abs/1606.01540.

[31] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, *Open ai*, `https://gym.openai.com`, Online; Accessed: 2022/03/28.

[32] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[33] K. Cho, B. van Merrienboer, D. Bahdanau, and Y. Bengio, "On the properties of neural machine translation: Encoder-decoder approaches," *CoRR*, vol. abs/1409.1259, 2014. arXiv: `1409.1259`. [Online]. Available: `http://arxiv.org/abs/1409.1259`.

[34] K. O'Shea and R. Nash, "An introduction to convolutional neural networks," *CoRR*, vol. abs/1511.08458, 2015. arXiv: `1511.08458`. [Online]. Available: `http://arxiv.org/abs/1511.08458`.

[35] W. Wang and J. Gang, "Application of convolutional neural network in natural language processing," in *2018 International Conference on Information Systems and Computer Aided Education (ICISCAE)*, 2018, pp. 64–70. DOI: `10.1109/ICISCAE.2018.8666928`.

[36] A. Borovykh, S. Bohte, and C. W. Oosterlee, *Conditional time series forecasting with convolutional neural networks*, 2017. DOI: `10.48550/ARXIV.1703.04691`. [Online]. Available: `https://arxiv.org/abs/1703.04691`.

[37] P. Hendriks and L. Van Damme, *Tool for stl monitoring*, `https://github.com/pieter-hendriks/STL-monitoring`, Online.

[38] P. Hendriks, *Master's thesis repository*, `https://github.com/pieter-hendriks/thesis`, Online.

[39] T. Parr, *Another tool for language recognition*, `https://www.antlr.org/`, Online; Accessed: 2022/05/28.

[40] L. Van Damme, *Tool for stl monitoring*, `https://github.com/LaurensVanDamme/STL-monitoring`, Online.

[41] D. D'Souza and N. Tabareau, "On timed automata with input-determined guards," *CoRR*, vol. abs/cs/0601096, 2006. arXiv: `cs/0601096`. [Online]. Available: `http://arxiv.org/abs/cs/0601096`.

[42] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: `http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf`.

[43] P. Christodoulou, *Deep reinforcement learning algorithms with pytorch*, `https://github.com/p-christ/Deep-Reinforcement-Learning-Algorithms-with-PyTorch`, Online; Accessed: 2022/04/07.

[44] D. P. Kingma and J. Ba, *Adam: A method for stochastic optimization*, 2014. DOI: `10.48550/ARXIV.1412.6980`. [Online]. Available: `https://arxiv.org/abs/1412.6980`.