# Robust Monitoring of Medical Cyber-Physical Systems

Saarland University

Center of Bioinformatics

Bachelor Bioinformatics

Bachelor's Thesis

*submitted by*

Jessica Aline Schmidt

Saarbrücken, February 2021

| | |
|---|---|
| Supervisor: | Prof. Bernd Finkbeiner, Ph.D. |
| Advisor: | Maximilian Schwenger, M.Sc. |
| Reviewers: | Prof. Bernd Finkbeiner, Ph.D.<br>Prof. Dr. Andreas Keller |
| Submission: | 09 February, 2021 |

## Abstract

Medical cyber-physical systems (MCPS) have become indispensable in modern medicine. They monitor patients' endogenous data, warn them if abnormalities are found and might even decide on a treatment. An essential requirement of such health-critical systems is robustness, a metric that small changes in the input only cause slight changes in the output. In particular, measurement errors should influence the output only slightly and not lead to wrong decisions of the monitor.

Using RTLola as a monitoring language, the thesis present an algorithm to analyze the robustness of an RTLola specification, which is used to generate the monitor. The algorithms are based on symbolic execution and constraint programming. They calculate the maximum output difference when the input is slightly mutated. Furthermore, we will interpret the outputs of the algorithms and evaluate them with respect to their practical application.

## Zusammenfassung

Medizinische cyber-physische Systeme (MCPS) sind unverzichtbar in der modernen Medizin. Sie überwachen die körpereigenen Daten der Patienten, warnen, wenn Anomalien festgestellt werden, und können sogar über Behandlungen entscheiden. Eine wesentliche Anforderung an solche gesundheitsrelevanten Systeme ist Robustheit, eine Metrik, die besagt, dass kleine Änderungen in der Eingabe nur geringe Änderungen in der Ausgabe verursachen. Insbesondere sollten Messfehler die Ausgabe nur geringfügig beeinflussen und nicht zu Fehlentscheidungen des Monitors führen.

Unter Verwendung von RTLola als Monitoringsprache wird in dieser Arbeit ein Algorithmus zur Analyse der Robustheit einer RTLola Spezifikation vorgestellt, die zur Generierung des Monitors verwendet wird. Die Algorithmen basieren auf symbolischer Ausführung und Constraintprogrammierung. Sie berechnen die maximale Ausgabedifferenz, wenn die Eingabe leicht verändert wird. Darüber hinaus werden wir die Ausgaben der Algorithmen interpretieren und im Hinblick auf ihre praktische Anwendung bewerten.

## Acknowledgements

I want to thank Prof. Bernd Finkbeiner and Prof. Andreas Keller for supporting my work and still giving me the freedom to integrate my own ideas. It is particularly important to me to say thank you for making it possible to write a thesis in this research area. Moreover, I thank Prof. Felix Mahfoud, Christian Ukena, and Jan Wintrich. Together with Prof. Andreas Keller, who have expanded my medical knowledge and enabled me to undertake application-oriented studies. Furthermore, I am grateful for all the help of Maximilian Schwenger, who always had an open ear for my questions and motivated me to go my way and to develop my own ideas. As importantly, I thank my family and friends for supporting me throughout the process of this thesis — especially for enduring and improving all my moods along the way. Here, I would like to say a special thank you to Stefan Oswald and Kallistos Weis, who have taken their time to proofread this thesis despite their own work.

**Erklärung**

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig verfasst und alle verwendeten Quellen angegeben habe.

**Statement**

I hereby confirm that I have written this thesis on my own and that I have not used any other materials than the ones referred to in this thesis.

**Einverständniserklärung**

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

**Declaration of Consent**

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

---

Saarbrücken, 09 February, 2021

# Contents

Chapter 1

# Introduction

*Medical cyber-physical systems* (MCPS, medical CPS) such as electronic implants, play an increasingly prominent role in modern medicine and personalized treatment. Automated insulin pumps, *Artificial Pancreata* (AP) [1], used to treat diabetes, are among the most frequently implanted devices. They collect and analyze endogenous data of patients to warn and protect them from health-critical situations. For that, they monitor medical properties and decide whether a treatment is required based on a given specification. Such monitoring is essential to process data quickly, as CPS often need to react in real-time to protect the system or patient. Since the monitor decides on the treatment and directly influences the patients' health status, it is a health-critical part of this architecture.

Such health-critical monitors may not make incorrect decisions that might harm the patient. Therefore, trustworthiness plays a significant role in the development of medical software. This trustworthiness includes explainability, traceability [2], verification [3], and robustness. Here, robustness describes the ability of a system to handle erroneous inputs and executions. It also considers how small changes in the input affect the output. A system is considered robust if small input changes only cause small output changes. Consequently, robustness plays a vital role since safe decisions and safe executions have the highest priority for MCPS.

Looking at these problems, the following can be concluded: we need a stable, explainable — and hence comprehensible — system for monitoring and evaluating the specifications. Such a framework is offered by RTLola [4, 5, 6]. RTLola develops around a stream-based, real-time specification language of the same name. It supports complex calculations on sliding windows over real-time streams and was already applied in other CPS like unmanned aircraft systems [7]. An RTLola specification uses real-time data of sensors as input and analyzes whether these values exceed a defined threshold. Therefore, a monitor based on such a specification can check whether the patient is in a health-critical situation. If a violation of the specification is found, the monitor warns the patient.

Additionally, RTLola already guarantees many desirable properties for MCPS. First of all, it offers an explainable and traceable execution of the specifications. It also ensures the exclusion of undefined behavior and the termination of all calculations defined within a specification. Moreover, it can statically analyze memory usage, such that the memory consumption can be adapted to the low available memory in MCPS. Consequently, RTLola offers essential guarantees for the application in MCPS, which generally have very few resources available and have to fulfill high safety, correctness, and robustness requirements.

Nevertheless, there is currently no analysis for the robustness of RTLola specifications. It is yet necessary to guarantee robustness to make RTLola applicable in the health-critical area and thus be able to use all the advantages of the language and its framework. However, it is impossible to specify one general robustness metric [8] for all systems, as they accept varying degrees of variation. In consequence, systems define robustness differently depending on their application area.

Hence, we develop a robustness analysis for RTLola specifications, which is kept general, i.e., it does not define a metric but calculates the output variation as a function of the input variation. Thus, for each specification it can be examined whether the output variation is still acceptable, and therefore the resulting monitor is considered robust.

Chapter 2

# Related Work

The vast amount of data has been a significant problem in medicine for a long time. Nowadays, machine learning offers many possibilities to process this data, but at the same time, it also brings many disadvantages — especially in safety-critical and health-critical contexts [9, 10]. In health care, machine learning models can, for example, improve the diagnosis and therefore help to find a personalized treatment. Machine learning has already found application in numerous fields such as tumor detection through image classification [11] and electronic health records through implants [12]. Nevertheless, neural networks are hard to train as medical data is often incomplete and noisy. Moreover, the neural networks' decisions must be understandable to provide a safe treatment for the patient.

Especially for systems using machine learning and other safety- or health-critical systems, an important property must be fulfilled: trustworthiness. This trustworthiness includes explainability, traceability [2], certifiability [13], and verification [3]. Here, robustness plays a crucial role since safe decisions have the highest priority in health-critical areas. While researchers already tried to verify neural networks in safety-critical areas such as autonomous driving [14], computer vision [15], and cyber-physical systems [16, 17], robustness needs more consideration. Especially analyses of implants are lacking, so that often no statements about their robustness can be found.

To achieve the primary goal of having a robust monitor, we need an already stable framework that guarantees a safe and explainable execution. Such a framework is offered by RTLola [4, 18]. RTLola evolves around a stream-based specification language of the same name [5, 6]. As a real-time monitoring language, RTLola handles real-time streams, supports complex calculations including sliding windows, and offers many guarantees on, e.g., the runtime and termination of calculations declared in an RTLola specification. It is also possible to statically analyze the specification's memory consumption, which solves the problem of strict resource limitations in implants. Moreover, RTLola is realizable in hardware [19] and was already applied to aircraft systems [7, 20] and network communication [21]. Besides all these benefits, a

robustness analysis for RTLola specifications does not exist yet. Since it is a known problem that there is no general metric for robustness [8], an algorithm must allow the possibility of defining a new metric for every system or interpreting outputs differently to make it applicable to several application areas. This point is addressed in this work, which lays the necessary foundations to include machine learning directives into RTLola and make it applicable to MCPS.

More formally, robustness requires a comparison of two inputs to analyze the output behavior. Such properties that check more than one execution trace are called hyperproperties [22, 23] — contrarily to, e.g., LTL [24] or STL [25], which can only express individual trace properties. Within hyperproperties, we can express the similarity of inputs and outputs, or several executions, based on their traces.

To show that a monitor is not robust, an input pair must be found that violates the property. Techniques for robustness analysis often include fuzzing [27] and symbolic execution [28]. Additionally, *reliability*, a subtopic of robustness, applies metrics such as *mean time between failure* (MTBF) and *mean time to failure* (MTTF) [29]. Although MTBF and MTTF are common in the verification process of cyber-physical systems (CPS), they are not applicable in monitoring contexts since the monitor is running continuously and is not allowed to fail. Additionally, it is a known problem that fuzzing does not provide any formal guarantees and can oversee branches that are barely executed. Consequently, a fuzzer's output is not the robustness metric we can rely on in the health-critical context.

Symbolic execution, however, does not have these problems and can investigate robustness in a very general way. Nevertheless, it cannot analyze unbounded loops and unbounded recursion [30]. Additionally, this approach has to face runtime and memory issues when analyzing larger programs or specifications [30] because the search space for finding a solution out of the symbolic traces is too complex. Thus, the idea of symbolic execution can be applied to robustness, but it is not sufficient on its own.

All in one, robustness analysis tries to find the maximum output difference while bounding the input difference. Here, the optimization is intricate with discrete inputs and outputs due to various problems [26]. These problems include that local extrema are considered as the optimum during optimization, whereas the global extremum is searched. This is explained by the fact that many optimizations follow the gradient: if the sign of the gradient changes, it is assumed that an extremum is present, but it cannot be decided whether it is only a local or the global extremum.

Chapter 3

# Preliminaries

## 3.1. Properties for System Executions

When arguing about the correctness of a system, it is inevitable to examine the system's *execution*. For this, we model a system as a non-empty set of infinite execution traces, where a *trace* is defined as a sequence of states. Hereby, every finite trace — which represents the termination of an execution — can be extended to an infinite trace by looping the final state infinitely often.

Clarkson et al. introduced the following notation in their paper *Hyperproperties* [22], which we adopt in this work:

**Definition 1** (Set of Traces)

Let $\Sigma$ be the set of states. The set of traces $\Psi$ is

$$\Psi = \Psi_{fin} \cup \Psi_{inf}$$

$$\Psi_{fin} = \Sigma^*$$

$$\Psi_{inf} = \Sigma^\omega$$

**Definition 2** (Trace Access)

For a trace $t = s_0 s_1 ...$ and an index $i \in \mathbb{N}$, we define the following trace access:

$$t[i] = s_i$$

$$t[..i] = s_0 s_1 ... s_i$$

$$t[i..] = s_i s_{i+1} ...$$

### 3.1.1. Trace Properties

A *trace property* describes that a defined property holds for a specific execution trace. For this, a trace property is defined as a set of execution traces.

**Definition 3** (Set of Trace Properties)
The set of all trace properties *Prop* is defined as

$$Prop = \mathcal{P}(\Psi_{inf})$$

where $\mathcal{P}$ denotes the powerset.

**Definition 4** (Satisfying Trace Property)
A trace property $P$ is satisfied by a set of traces $T$, if and only if $T \subseteq P$. Then we write $T \vDash P$.

$$T \vDash P \iff T \subseteq P$$

Nevertheless, trace properties cannot compare different execution traces to each other and are only applicable to individual traces.

### 3.1.2. Hyperproperties

*Hyperproperties* can relate different traces and build sets of trace properties, i.e., sets of sets of execution traces.

**Definition 5** (Set of Hyperproperties)
The set of all hyperproperties *HP* is defined as

$$\begin{aligned} HP &= \mathcal{P}(\mathcal{P}(\Psi_{inf})) \\ &= \mathcal{P}(Prop) \end{aligned}$$

where $\mathcal{P}$ denotes the powerset.

**Definition 6** (Satisfying Hyperproperty)
A hyperproperty $H$ is satisfied by a set of traces $T$, if and only if $T \in H$. Then we write $T \vDash H$.

$$T \vDash H \iff T \in H$$

As hyperproperties only generalize trace properties, every trace property can be defined as a hyperproperty — but not vice versa. Moreover, trace properties cannot compare traces with each other as they only define properties for single traces. Because hyperproperties are defined over sets of trace properties, different traces — and thus different executions — can be compared. This comparability is necessary to define robustness.

## 3.2. RTLola

RTLola [5, 6] is a stream-based specification language aiming for expressiveness and simplicity. Although these two goals seem to be contradictory at first sight, RTLola supports complex real-time computations while at the same time pursuing the purpose of an easily understandable syntax.

In the following, we see an overview of the RTLola syntax and semantics. Note that we only present the supported subset of the RTLola grammar. For details, see Schwenger's *Let's not Trust Experience Blindly: Formal Monitoring of Humans and other CPS* [6].

### 3.2.1. Basics

| Syntax | Semantics |
|---|---|
| `input i: T` | defines an input stream `i` of type `T` |
| `output o: T := e` | defines an output stream `o` of type `T` which is described by the expression `e` |
| `trigger cond "msg"` | warns with message `msg` if the condition `cond` evaluates to false (`cond` has to be of type `Bool`) |

Table 3.1.: RTLola syntax for stream definitions

Possible types `T` are: `Bool`, `Int64`, `UInt64`, `Float64`

### 3.2.2. Stream Access

| Syntax | Semantics |
|---|---|
| `s` | accesses current value of stream `s` |
| `s.offset(by: n)` | accesses value with offset `n` of stream `s` (only negative offsets are possible, i.e. $n < 0$) |
| `s.defaults(to: val)` | defines the default value `val` which is used if the stream `s` is accessed out of bounds |
| `if cond then` $s_1$ `else` $s_2$ | execute $s_1$ if `cond` (has to be of type `Bool`) evaluates to true, otherwise execute $s_2$ |

Table 3.2.: RTLola syntax for stream accesses

Additionally, we only consider event-based streams and leave periodic streams out of consideration. Consequently, we only support discrete past offsets and do not handle real-time streams so that only discrete offsets, i.e., untimed offsets, can be used.

### 3.2.3. Monitoring

RTLola converts a given specification into a monitor. For this simplified subset and synchronous monitoring, we can imagine that this monitor iterates — possibly infinitely often — over the defined streams and updates their values according to their new calculations. Intuitively, synchronous here means that all input streams receive inputs simultaneously. In contrast, asynchronous allows an input stream to receive, for example, two inputs while another input stream did not have any input.

We call the number of iterations currently analyzed the *evaluation depth* of the specification. For example, if the evaluation depth is two, the monitor iterates twice over every stream — hence, every stream is evaluated two times. Accordingly, each iteration is called *evaluation step*.

The monitoring process becomes more complicated when including periodic streams and asynchronicity. Nevertheless, we only consider the reduced grammar shown above so that this simplified idea is adequate. This conception is sufficient for this work's comprehensibility, and we will not formalize this any further.

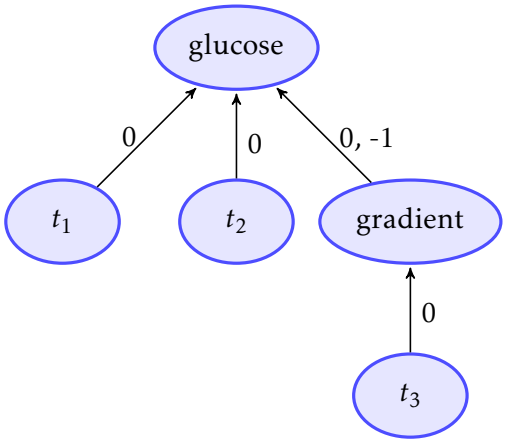
### 3.2.4. Dependency Graph

```
1 input glucose: Int64 // glucose values from glucose sensor
2 trigger glucose > 120 "Hyperglycemia detected!"
3 trigger glucose < 60 "Hypoglycemia detected!"
4 output gradient: Int64 := glucose - glucose.offset(by: -1).defaults(to: glucose)
5 trigger gradient > 50 ∨ gradient < -20 "Glucose value varys a lot!"
```

Figure 3.3.: Example of an RTLola specification for monitoring the glucose levels in blood

RTLola statically analyzes the memory consumption of a specification by building a *dependency graph*. Each stream is a node in the graph. The edges represent the stream accesses, with the weights corresponding to the offsets being accessed. Consequently, this graph represents the dependencies between the streams: if and only if $s_1$ accesses $s_2$, there is an edge from $s_1$ to $s_2$ in the dependency graph. From this construction, it follows that if and only if $s_1$ *depends* on $s_2$, i.e., $s_1$ accesses $s_2$ directly or indirectly by accessing another stream which then accesses $s_2$, there is a path from $s_1$ to $s_2$ in the dependency graph.

**Example 3.2.1.** Consider the specification in Figure 3.3. This specification has the following dependency graph:




Note that $t_1$, $t_2$ and $t_3$ denote the first, second, and third trigger, respectively. △

### 3.2.5. Glucose Monitoring

Consider an automated insulin pump that analyzes the glucose level in a patient's blood and administers insulin to counteract hyperglycemia, a health-critical situation in which the glucose level rises too high. Typically, we speak of hyperglycemia at values above 120 $mg \cdot dl^{-1}$. In contrast, hypoglycemia describes a period of too low blood glucose values, i.e., less than 60 $mg \cdot dl^{-1}$, and can be threatened by eating or drinking something sugary. These situations can be predicted by considering the gradient of the glucose curve, as high or low gradients might indicate too high or too low glucose levels, respectively.

Both hypo- and hyperglycemia are health-critical. Therefore, a patient needs to be warned — and in consequence, prevented from even more severe health status. For this, we want to implement an RTLola specification that considers all these critical points and informs the patient if hypo- or hyperglycemia is detected.

**Example 3.2.2.** Consider the specification in Figure 3.3.

In the first line, we define an input stream `glucose` of type `Int64` with the values measured by the glucose sensor. The next two lines warn the patient if the glucose level is either too high, `glucose > 120`, or too low, `glucose < 60`. We can use the keyword **`trigger`** and use the string after the condition as the warning message. A warning is raised if its condition evaluates to true — we also say the trigger is *taken*.

Line 4 defines a new output stream `gradient` to calculate the gradient. For simplicity, we only consider the difference between two successive glucose values. As RTLola is a stream-based language, every input and output holds its history of values, e.g., the stream `glucose` stores every measured glucose value. When we want to access the current glucose value, we write `glucose`. To get the predecessor of glucose, we can use the `offset`-function on a stream-value, i.e., `glucose.offset(by: -1)` gives us the previous glucose value. As the first value of `glucose` does not have a predecessor, we use the function `defaults` to define a value that is chosen if the stream is accessed out of bounds. Here we

select the value of `glucose` itself because we do not have any additional information and consider the glucose levels to be constant.

Lastly, we can use the last trigger to indicate that the gradient is either rising or falling too fast, respectively, `gradient > 50` or `gradient < -20`. △

Chapter 4

# Robustness

Robustness describes the capability of a program or system to cope with small mutations in the input. This robustness is necessary to estimate the behavior of MCPS. MCPS have the crucial task of deciding on health-critical issues. At the same time, however, wrong decisions can also harm patients. Such wrong decisions can be triggered by measurement errors. However, the system must be able to reckon with such errors and consequently cope with them to prevent harm to the patient. To guarantee this, MCPS must be robust.

Since the sensors' accuracy in different systems varies, and the systems themselves are subject to various fluctuations, there can be no general metric for robustness. Nevertheless, robustness can be concluded in the fact that small mutations in the inputs only cause slight variations in the program's output. In consequence, a robust system allows better predictability on its outputs and gives several safety guarantees. All these guarantees aim to protect the patient from health risks.

## 4.1. Definition

Before we consider systems, we first formalize robustness on functions.

**Definition 7** ($(\epsilon, \delta)$-Robustness of Functions)

A function $f : \mathbb{R}^n \to \mathbb{R}^n, n \in \mathbb{N}$, is called $(\epsilon, \delta)$-robust, if a difference of at most $\epsilon \in \mathbb{R}$ of the inputs can cause a difference of at most $\delta \in \mathbb{R}$ of the outputs. Let $d : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$ be a distance function and $i_1, i_2 \in \mathbb{R}^n$.

$$d(i_1, i_2) \leq \epsilon \implies d(f(i_1), f(i_2)) \leq \delta$$

In consequence, $\delta$ is a quantity for the robustness of a function. Therefore, maximizing $\delta$ for a fixed $\epsilon$ allows us to find an upper bound for the the outputs' variation

depending on $\epsilon$. Moreover, $\epsilon$ and $\delta$ can be imaged as the allowed ranges for the inputs and outputs. The shape of these ranges depends on the chosen distance functions.

### 4.1.1. Distance Functions

There are several conceivable possibilities for distance norms to compare the inputs and outputs. In this thesis, we will focus on the following two norms: first of all, the *Manhattan norm*, also known as $L_1$ *norm*, to calculate the *Manhattan distance* — and secondly, the *Chebyshev norm*, also known as $L_\infty$ *norm*, for the *Chebyshev distance*.

**Definition 8** (Manhattan norm, Manhattan distance)
Let $x, y \in \mathbb{R}^n$, $n \in \mathbb{N}$, be:

$$x = (x_1, ..., x_n)^T, \quad y = (y_1, ..., y_n)^T$$

Then the *Manhattan norm* of $x$ is defined as the sum of all values in the vector $x$:

$$||x||_1 = \sum_{i=1}^{n} x_i$$

The *Manhattan distance* of $x$ and $y$ is defined as the sum of their differences:

$$d_1(x, y) = ||x - y||_1 = \sum_{i=1}^{n} |x_i - y_i|$$

**Definition 9** (Chebyshev norm, Chebyshev distance)
Let $x, y \in \mathbb{R}^n$, $n \in \mathbb{N}$, be:

$$x = (x_1, ..., x_n)^T, \quad y = (y_1, ..., y_n)^T$$

Then the *Chebyshev norm* of $x$ is defined as the maximum absolute value within the vector $x$:

$$||x||_\infty = max\{|x_1|, ..., |x_n|\}$$

The *Chebyshev distance* of $x$ and $y$ is defined as the maximum of their absolute differences:

$$d_\infty(x, y) = ||x - y||_\infty = max\{|x_1 - y_1|, ..., |x_n - y_n|\}$$

Consequently, the Manhattan distance considers all vector's differences, while the Chebyshev distance only considers the "worst", i.e., the greatest, difference. Accordingly, the Manhattan distance penalizes every error, whereas the Chebyshev distance

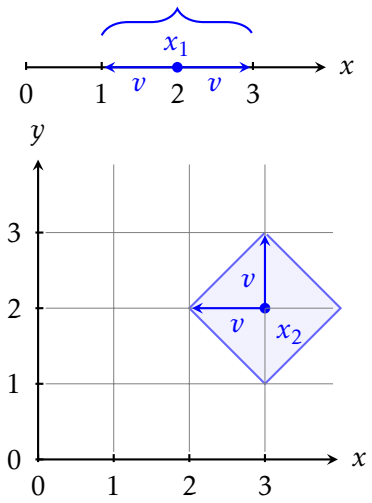
penalizes only the most severe one. We can imagine the Manhattan distance like a distribution system, which distributes a maximum amount of $\epsilon$ over the inputs, whereas the Chebyshev distance assigns at most $\epsilon$ to each input.

Therefore, the Manhattan norm is also stricter when considering differences as errors since it requires a tighter bound than the Chebyshev distance. This can be seen in the following examples.

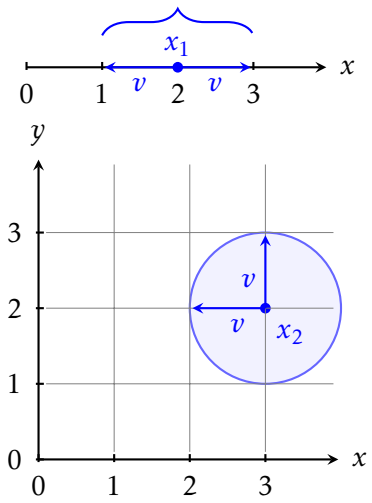
**Example 4.1.1.** Define two vectors $x_1 \in \mathbb{R}^1$, $x_2 \in \mathbb{R}^2$:

$$x_1 = (2), \quad x_2 = (3, 2)^T$$

Then, the Manhattan and the Chebyshev norm build the following shapes if the variance is $v \in \mathbb{R}$, $v = 1$. Here, $x$ can be either an input or output vector, so that $v$ is either $\epsilon$ or $\delta$, respectively.




Manhattan norm




Chebyshev norm

The line drawn through the variation in the one-dimensional, or the spanned area in the two-dimensional, contain all values that x can take due to the variation $v$. In the one-dimensional, both norms still behave in the same way. Looking at the two-dimensional areas, it is noticeable that the area drawn by the Manhattan norm is smaller than that of the Chebyshev norm. Since this area contains all allowed values, it is a metric for the number of accepted values. Consequently, the Manhattan norm is stricter compared to the Chebyshev norm, since it allows fewer values. △

**Example 4.1.2.** Define two vectors $x, y \in \mathbb{R}^3$:

$$x = (1,2,7)^T, \quad y = (1,3,5)^T$$

Then it is:

$$\text{Manhattan distance: } d_1(x,y) = \|x-y\|_1 = \quad |1-1| + |2-3| + |7-5| = 3$$

$$\text{Chebyshev distance: } d_\infty(x,y) = \|x-y\|_\infty = max\{|1-1|, |2-3|, |7-5|\} = 2$$

If $x$ and $y$ were two output vectors, and $\delta = 2$, the robustness definition would be satisfied when using the Chebyshev distance, but unsatisfied with the Manhattan distance:

$$\text{Manhattan distance: } d_1(x,y) \leq \delta \iff 3 \leq 2 \iff \bot$$

$$\text{Chebyshev distance: } d_\infty(x,y) \leq \delta \iff 2 \leq 2 \iff \top$$

Therefore, the Manhattan distance is again stricter because it already rejects this example, whereas the robustness definition is still fulfilled when using the Chebyshev distance. △

To apply these distances in connection with the robustness definition also to systems — which do not directly represent functions that are comparable — additional definitions are necessary.

### 4.1.2. Robustness as a Hyperproperty

To adapt the $(\epsilon, \delta)$-Robustness shown in Definition 7 on systems, it is inevitable to examine the system's execution. To model this, we refer to the system as a set of execution traces, where a trace is a sequence of states. More precisely, we need to compare every trace with every other trace, which inputs differ by at most $\epsilon$. If the comparison of all pairs of traces shows that the outputs differ by at most $\delta$, we call the system robust.

→ Sec. 3.1, Page 5

These comparisons can be expressed with *Hyperproperties*, which are presented in Section 3.1. It needs to be mentioned that *Trace Properties* are not sufficient for expressing robustness since they are not able to compare multiple traces with each other.

**Definition 10** ($(\epsilon, \delta)$-Robustness of Systems)

Let $\epsilon$ and $\delta$ be upper bounds for the maximum variations of the inputs and outputs, respectively. Let $in(t)$ be a function returning the inputs to a trace $t$, and $out(t)$ a function returning the outputs. Then we define the robustness $R$ as:

$$R = \{T \in Prop \mid \forall t_1, t_2 \in T : d(in(t_1), in(t_2)) \leq \epsilon \implies d(out(t_1), out(t_2)) \leq \delta\}$$

Moreover, hyperproperties allow us, besides comparing the inputs and outputs of traces, to compare the reached states. Hence, robustness can also be defined as the fact

that, e.g., the prefixes of every pair of traces are equal. This therefore allows to show that not the inputs are equal, but a part of the execution. Nevertheless, we will focus on the $(\epsilon, \delta)$-Robustness definitions, which we refer to when merely writing *robustness*.

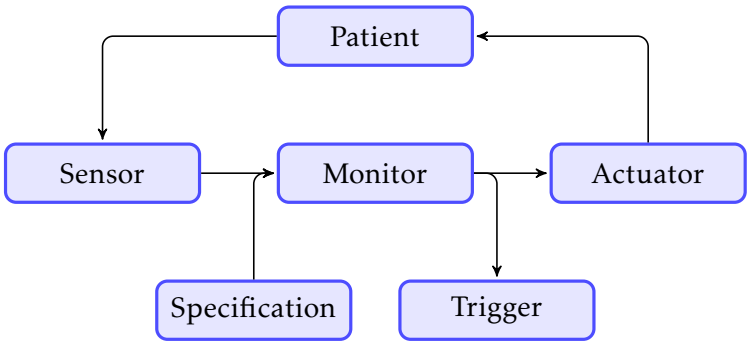
With the help of this robustness definition, it is now possible to draw various conclusions regarding systems' safety, which provides significant advantages for its medical applicability.

### 4.1.3. Safety Guarantees

Robustness allows for representing the output variation of a system depending on the input variation. Consequently, we find an upper bound on how far the inputs affect the outputs. This upper bound makes it possible to estimate the system's behavior better and make a prediction about its behavior. In particular, this applies to expected input variations as well as erroneous inputs. These inherently unexpected errors can also be tested only slightly. Nevertheless, with robustness, we can set a frame in which the output will be contained.

In practice, this implies that a patient can be protected from possible mistreatment, which can be the result of measurement errors in particular. Moreover, we can predict that the system will behave similarly with similar inputs. Thus, if we know whether a system administers the correct dose of medication in a health-critical condition, we can assume that the system will do so in similar health-threatening situations. To further illustrate the importance of robustness, consider the following example.

**Example 4.1.3.** Consider an AP, which monitors a patient's glucose levels in blood. This system consists of the following components:




The sensor measures the glucose levels of the patient. These values are passed to a monitor, which then decides — based on a specification — whether a health-critical situation is currently present. This specification includes values, e.g., glucose levels indicating hypo- or hyperglycemia. If the monitor detects a health risk, it can use a trigger to warn the patient. Moreover, it can decide whether a treatment is required, and the actuator should apply a medication. In our case, this medication is insulin, which can be applied at hyperglycemia stadiums to counteract the high glucose levels. Nevertheless, if insulin is applied if the glucose level is too low, we might cause hypoglycemia, which can be manifested by various, partly severe, symptoms.

Now consider the following scenario: the glucose sensor measures a value of 180 $mg \cdot dl^{-1}$. This value implies hyperglycemia. Consequently, the monitor decides to administer insulin to normalize this value. If the patient, however, would have an actual glucose value of 100 $mg \cdot dl^{-1}$, which is still within the normal range, and the 180 $mg \cdot dl^{-1}$ was actually a measurement error, we would likely induce hypoglycemia by incorrectly treating with insulin. This could, for example, cause the patient to faint. Hence, by our decision to administer insulin, we would have harmed a patient. This case must never occur. However, if the monitor was robust, it could recognize that the 180 $mg \cdot dl^{-1}$ was a measurement error, for example because the previous values were significantly lower, and decide against insulin administration. In this way, we would have protected a patient from mistreatment and the associated health risks. △

## 4.2. Concrete Robustness Analysis

In this chapter, we introduce a concrete analysis for the robustness of an RTLola specification. We present an algorithm for finding an input pair for a given specification that maximizes the output variation. Each step of the algorithm includes an example, which is in the end summarized by an execution protocol of the algorithm. Lastly, we discuss the method's limitations, which leads us to a second, more generalized approach.

### 4.2.1. Algorithm Overview

```
1 input i: Int64
2 output a: Int64 := a.offset(by: -1).defaults(to: 5) + 20
3 output b: Int64 := a + i.offset(by: -1).defaults(to: 0)
4 trigger b > 10 "Violation"
```

Figure 4.2.: Example of a simple RTLola specification

→ Sec. 4.2, Page 17

Algorithm 1 shows the main parts of the concrete robustness analysis of an RTLola specification. Given an RTLola specification, an evaluation depth $n$, and an $\epsilon$, which defines the maximum allowed variations of the input streams, we calculate the possible variation $\delta$ of the output streams after $n$ evaluation steps of the specification. For this, we rebuild the trigger condition by recursively examining every stream access made by the current stream, starting at the trigger. We construct a single large condition, called *true condition*, by conjuncting the so built sub-conditions. By renaming every variable within the true condition and negating it, we create a constraint for when the trigger evaluates to false, called *false condition*.

Moreover, we need a third *input condition* to guarantee that the inputs vary within the specified $\epsilon$ range, i.e., that the inputs which fulfill the true condition, and the inputs which fulfill the false condition, differ by at most $\epsilon$. Building the conjunction of the three conditions thus gives us a formula that can be solved using common SMT solvers. In consequence, we get an input for which the currently analyzed trigger evaluates to

**Algorithm 1:** concrete robustness analysis of an RTLola specification

**Input:** spec, $\epsilon$, n
**Output:** $\delta$

```
for trigger ∈ spec do
    unknown := trigger[n].output_dependencies
    true_condition := rename_copy(trigger[n], ⊤)

    for u ∈ unknown do
        new_condition := analyze(u)
        true_condition := true_condition ∧ new_condition
        unknown := (unknown ∪ u.output_dependencies) / {u}

    false_condition := ¬ rename_copy(true_condition, ⊥)
    input_condition := d(true_condition.inputs, false_condition.inputs) ≤ ϵ
    formula := true_condition ∧ false_condition ∧ input_condition
    formula.maximize_outputs()

    (input_true, input_false) := solve(formula)
    output_true := spec.exec(input_true)
    output_false := spec.exec(input_false)
    δ := d(output_true, output_false)
```

true and an input which it evaluates to false while differing by at most $\epsilon$. By calculating the difference of the output streams for these two inputs, we get the output variation $\delta$. Moreover, we can even express $\delta$ as a function of $\epsilon$.

#### Notation

Let s be a stream in an RTLola specification, and $n \in \mathbb{N}$: `s[n]` denotes the $n$-th evaluation step of s.

### 4.2.2. Memory Consumption

RTLola statically analyzes the memory requirement of a specification by considering its dependency graph, which is described in Section 3.2.4. From this analysis, we can see how many values of which stream need to be stored to accomplish every calculation defined within the specification. We call the number of required values for a stream its *memory consumption*.

→ Sec. 3.2, Page 8

**Definition 11** (Memory Consumption)

Let $max_w^s$ be the maximum weight of the ingoing edges of a stream $s$ in the dependency graph. Note, that $max_w$ is equal to the maximum offset with which $s$ is accessed and $max_w \leq 0$. Then, the memory consumption of s $mem_s$ is

$$mem_s = |max_w^s| + 1$$

If $s$ has no ingoing edges, it is $mem_s = 1$.

Intuitively, this definition follows directly from the dependency graph. If a stream $s_1$ accesses $s_2$ with offset $o$, there is an edge in the dependency graph from node $s_1$ to node $s_2$ with weight $o$. Therefore, the weights $w$ of the ingoing edges of a stream $s$ are exactly the offsets with which $s$ is accessed. To guarantee that enough values are stored so that every calculation of the streams depending on $s$ is possible, we need to store at least the values from position $max_w^s$ to 0 — where 0 is the current value — which are $|max_w^s| + 1$ values.

With this information, we can find a lower bound for the number of required values of an input stream and the number of variables within the true, false, and input condition.

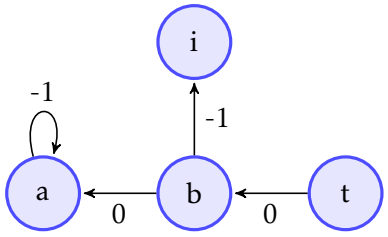
**Lemma 1.** Let *streams* be the set of all streams defined in an RTLola specification, and $mem_s$ the memory consumption of a stream $s$. We need at least

$$max_{mem} = \max_{s \in streams} \{mem_s\}$$

values of each input stream to execute every operation within the specification, i.e., execute the specification without using any default cases.

It follows that we need to evaluate at least $max_{mem}$ iterations of a specification to analyze its robustness — as otherwise, not every part of the specification would be executed, and we oversee possibly unrobust calculations.

**Example 4.2.1.** Consider the RTLola specification in Figure 4.2. This specification has the following dependency graph: →Sec. 4.2, Page 16




It follows that:

$$mem_i = 2,\ mem_a = 2,\ mem_b = 1,\ mem_t = 1$$
$$\Rightarrow max_{mem} = 2$$

Consequently, we need 2 values of the input stream $i$ to execute every calculation defined within the specification. △

### 4.2.3. Stream Access

As we focus on event-based streams, we only have to handle discrete stream offsets. From Lemma 1, we know that we need to evaluate at least $max_{mem}$ values. Hence, to simulate a stream by a vector, the vector needs at least length $n$, $n \geq max_{mem}$. Within this vector representation, every access with offset $m$ to the stream is represented by accessing the associated vector at position $n + m - 1$. It needs to be mentioned that $m \leq 0$ because RTLola only supports past offsets. So, `s` is — theoretically — equivalent to `s.offset(by: 0)` for some stream `s` — although the syntax does not allow 0 within an offset. Moreover, it follows from the definition of memory consumption, presented in Definition 11, and $n \geq max_{mem}$ that $n > m$. →Sec. 4.2, Page 18 →Sec. 4.2, Page 18

**Example 4.2.2.** Consider the RTLola specification in Figure 4.2. From Example 4.2.1, we know that $max_{mem} = 2$. When replacing the streams with vectors of length 2, we can imagine the specification to look like the following. For simplicity, we leave types out of consideration. →Sec. 4.2, Page 16

```
1 [i0,i1]
2 a1 = a0 + 20
3 b1 = a1 + i0
4 t1 = b1 > 10
```

Here, we emitted the default case when accessing $i$ at offset $-1$ because we know that $i_0$ is accessible when evaluating $i_1$. Nevertheless, we see that $a_0$ is still unknown, and we have to analyze this access again:

```
a_0 = a_-1 + 20
```

As −1 is an out of bounds access, we replace it with the default case:

```
a_0 = 5 + 20
```

This results in the following calculations:

```
1 [i_0, i_1]
2 a_0 = 5 + 20
3 a_1 = a_0 + 20
4 b_1 = a_1 + i_0
5 t_1 = b_1 > 10
```

We now have a specification where every access is known and therefore is executable. Note, that input streams are always considered to be known. △

As we now know how to represent the required input streams and output streams by vectors of length at least $max_{mem}$, we will refer to them as *input vectors* and *output vectors*, respectively. Moreover, we call this representation *vector representation*.

### 4.2.4. Types and Norms

RTLola supports four different types for input streams: `Bool` (boolean), `Int64` (signed integer), `UInt64` (unsigned integer), and `Float64` (floating-point number). For comparing input and output vectors, we either apply the Manhattan or the Chebyshev distance to define that two input vectors $x$ and $y$, both of length $n$, may vary within the defined $\epsilon$ range:

$$d_1(x-y) \leq \epsilon = \|x-y\|_1 \leq \epsilon \equiv \sum_{i=1}^{n} |x_i - y_i| \leq \epsilon$$

$$d_\infty(x-y) \leq \epsilon = \|x-y\|_\infty \leq \epsilon \equiv max\{|x_1 - y_1|, ..., |x_n - y_n|\} \leq \epsilon$$

Note, that both distances are only defined in $\mathbb{R}^n$, $n \in \mathbb{N}$. In consequence, they are directly applicable to streams of types `Int64`, `UInt64`, and `Float64`. To make them also applicable to `Bool`, we interpret *true* ($\top$) and *false* ($\bot$) as 1 and 0, respectively. Nevertheless, we do not support a robustness analysis including `Float64` in this thesis. We will discuss this problem in Section 4.4.5.

### 4.2.5. Building the Overall Constraint

With the previously examined information, we can now generate the true, false, and input condition.

→ Sec. 4.2, Page 19

**Example 4.2.3.** Consider the RTLola specification in Figure 4.2. In Example 4.2.2, we build the analyzed specification:

```
1 [i_0, i_1]
2 a_0 = 5 + 20
3 a_1 = a_0 + 20
4 b_1 = a_1 + i_0
5 t_1 = b_1 > 10
```

With this, we can now generate the constraints which need to be solved:

- true condition:

$$\left(a_0^t = 5 + 20\right) \wedge \left(a_1^t = a_0^t + 20\right) \wedge \left(b_1^t = a_1^t + i_0^t\right) \wedge \left(b_1^t > 10\right)$$

- false condition:

$$\left(a_0^f = 5 + 20\right) \wedge \left(a_1^f = a_0^f + 20\right) \wedge \left(b_1^f = a_1^f + i_0^f\right) \wedge \neg\left(b_1^f > 10\right)$$

- input condition:

$$d\left((i_1^t, i_0^t), (i_1^f, i_0^f)\right) \leq \epsilon$$

  where $d$ is either $d_1$ or $d_\infty$

To get the complete formula, we conjunct all conditions which results in the following constraint:

$$\begin{aligned} &\left(a_0^t = 5 + 20\right) \wedge \left(a_1^t = a_0^t + 20\right) \wedge \left(b_1^t = a_1^t + i_0^t\right) \wedge \left(b_1^t > 10\right) \\ \wedge\ &\left(a_0^f = 5 + 20\right) \wedge \left(a_1^f = a_0^f + 20\right) \wedge \left(b_1^f = a_1^f + i_0^f\right) \wedge \neg\left(b_1^f > 10\right) \\ \wedge\ &d\left((i_1^t, i_0^t), (i_1^f, i_0^f)\right) \leq \epsilon \end{aligned}$$

△

### 4.2.6. Maximizing Output Difference

Until now, we calculated the output variation for *one* possible set of input vectors. Nevertheless, we are interested in the *maximum* output variation. For this, we need to *maximize* the difference of the output streams for when the trigger evaluates to true and false and expand the previous constraint with this optimization constraint.

**Example 4.2.4.** Consider the RTLola specification in Figure 4.2. In Example 4.2.3, we build the output condition: As we want to maximize the output variation for each stream, we want to maximize the output stream difference:

→ Sec. 4.2, Page 16

$$max\left(d\left((a_0^t, a_1^t), (a_0^f, a_1^f)\right)\right) \wedge max\left(d\left((b_0^t, b_1^t), (b_0^f, b_1^f)\right)\right)$$

where $d$ is either $d_1$ or $d_\infty$. △

### 4.2.7. Example

→ Sec. 4.2, Page 16
→ Sec. 4.2, Page 30

Consider the RTLola specification in Figure 4.2. The execution protocol for this specification can be seen in Table 4.3. We define $n = 2$, $\epsilon = 3$ and use the Chebyshev norm.

### 4.2.8. Limitations

To summarize, the concrete robustness analysis provides an example input, which maximizes the output variation. This input can be used for further testing the specification since the input is a particular case that maximizes the output variation. Hence, fixing this example improves the specification's robustness. Nevertheless, it cannot be guaranteed that the optimal solution is found. As we use integers, the optimization results in a *discrete optimization problem*. Consequently, the solution of the optimization problem requires a lot of runtime. It may also be that a local extremum is accepted as the optimal solution, whereas the global extremum is searched.

Moreover, the specification's behavior needs to be investigated for arbitrary evaluation steps to make a confident statement about the specification's robustness. Thus, it can easily be overseen that output streams become non-robust for high evaluation depths, or only one specific step is non-robust. Consequently, we need an approach that abstracts from the concrete evaluation depth and makes a more general statement about the specification's robustness.

## 4.3. Symbolic and Concolic Robustness Analysis

The previously described approach provides input examples on how to achieve the calculated output variation $\delta$. For this, a fixed evaluation depth $n$ is required. If we want to make a statement about $\delta$ for general $n$, we need to represent $\delta$ as a function of $\epsilon$. Here, we do not fix $n$ or $\epsilon$, but rather use them as a symbolic input to our approach and calculate $\delta$ as a function of $\epsilon$. Therefore, we do a symbolic execution of the specification.

Nevertheless, we first need to define *linear* and *recursive* streams to understand the design of the symbolic execution.

### 4.3.1. Linearity and Recursion of Streams

We define linear and recursive streams with the help of the dependency graph.

**Definition 12** (Linear and Recursive Streams)

A stream is called *linear* if it does not depend on a stream which occurs within a cycle in the dependency graph.

Contrarily, a stream is called *recursive* if it does depend on a stream which occurs within a cycle in the dependency graph — we say that the specification includes *circular dependencies*. Note that self edges also count to cycles.

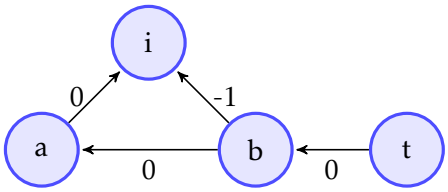
```
input i: Int64
output a: Int64 := i + 20
output b: Int64 := a + i.offset(by: -1).defaults(to: 0)
trigger b > 10 "Violation"
```

Figure 4.4.: Linear example of an RTLola specification

**Example 4.3.1.** Consider the RTLola specification in Figure 4.2 and its dependency graph shown in Example 4.2.1. As $a$ has a self-edge, $a$ is a recursive stream. Consequently, $b$ and $t$ are also recursive because both depend on $a$. △

→Sec. 4.2, Page 16

**Example 4.3.2.** Consider the RTLola specification in Figure 4.4. This specification has the following dependency graph:




As there are no circles in the dependency graph, all streams are linear. △

### 4.3.2. Algorithm Overview

We split the specification into linear and recursive streams, as it is not possible to symbolically execute recursive parts of the specification. Algorithm 2 shows the main part of the symbolic robustness analysis of an RTLola specification. From Lemma 1, we know that the behavior of a stream $s$ does not vary when analyzing more than $max_{mem}$ steps. Consequently, it is sufficient to calculate $max_{mem}$ iterations, as all subsequent steps behave equally to step $max_{mem}$.

→Sec. 4.3, Page 24
→Sec. 4.2, Page 18

To analyze the recursive streams, we take an evaluation depth $n$ as an input to perform a *concolic* analysis, i.e., a mix between concrete and symbolic analysis. This approach can be seen in Algorithm 3. Here, we analyze a concrete number of $n$ evaluation steps but treat the allowed variation $\epsilon$ as a symbolic input.

→Sec. 4.3, Page 24

As we do not analyze the specification concretely, we do not have fixed values to decide on conditions in if cases or triggers. Therefore, we leave these out of consideration.

**Algorithm 2:** symbolic robustness analysis of an RTLola specification

```
Input: spec
Output: δ
linear_streams := spec.linear_output_streams
max_mem := spec.max_mem

for stream ∈ linear_streams do
    formula := stream[max_mem]
    unknown := stream[max_mem].output_dependencies

    for u ∈ unknown do
        analyzed := analyze(u)
        formula := formula.inline(analyzed)
        unknown := (unknown ∪ u.output_dependencies) / {u}
    δ := remove_non_epsilon_dependencies(formula)
```

**Algorithm 3:** concolic robustness analysis of an RTLola specification

```
Input: spec, n
Output: δ
recursive_streams := spec.recursive_output_streams

for stream ∈ recursive_streams do
    formula := stream[n]
    unknown := stream[n].output_dependencies

    for u ∈ unknown do
        analyzed := analyze(u)
        formula := formula.inline(analyzed)
        unknown := (unknown ∪ u.output_dependencies) / {u}
    δ := remove_non_epsilon_dependencies(formula)
```

### 4.3.3. Evaluation Depth

As the concolic analysis shown in Algorithm 3 has a fixed evaluation depth, we do not have to argue which depth needs to be analyzed. Therefore, we now consider the symbolic analysis presented in Algorithm 2.

From Lemma 1, we already know that we need to analyze at least $max_{mem}$ iterations of the specification to investigate the entire specification's robustness. As $max_{mem}$ is the maximum required memory of all streams, it follows directly that a stream $s$ can access streams that have at most $max_{mem}$ memory consumption. Hence, it is sufficient to analyze $max_{mem}$ values for $s$ because, with $max_{mem}$, every calculation defined within $s$ is executed. This is sufficient to analyze the stream symbolically.

### 4.3.4. Example

Consider the specification in Figure 4.4. As shown in Example 4.3.2, this specification includes only linear streams. The execution protocol for the symbolic analysis can be seen in Table 4.5. → Sec. 4.3, Page 23 → Sec. 4.3, Page 31

We already saw in Example 4.3.1 that every stream of the specification in Figure 4.2 is recursive. Hence, we use this specification to show the concolic analysis of Table 4.6 and define $n = 2$. → Sec. 4.2, Page 16 → Sec. 4.3, Page 32

Although Figure 4.2 includes recursive streams, we can find a closed formula which represents the output variation. This is only the case because the recursive stream does not depend on input streams. More limitations are discussed in the following section.

**Example 4.3.3.** We adapt Example 4.2.2 for the specification presented in Figure 4.2. We transform the specification to use symbolic offsets. According to their memory consumption, we require 2 value to analyze the output streams $a$ and $b$, and 1 value to analyze the trigger. We need $max_{mem} = 2$ inputs $i$ so that every calculation is executed. → Sec. 4.2, Page 19

```
1 [i_{n-1}, i_n]
2 a_n = a_{n-1} + 20
3 b_n = a_n + i_{n-1}
```

Note, that we do not need to analyze $a_{n-1}$ as we know that $a_{n-1}$ behaves similarly to $a_n$ regarding $\epsilon$ for arbitrary $n$ because $a$ does not depend on input streams.

Now, we allow every input stream to vary within an $\epsilon$ range. We write $\pm\epsilon$ behind a stream which varies by at most $\epsilon$.

```
  [i_{n-1} ± ε, i_n ± ε]
```

Because the inputs vary by $\epsilon$, $b$ varies by $\epsilon$ too. As $a$ does not depend on the inputs, it does not vary.

```
1 a_n = a_{n-1} + 20
2 b_n ± ε = a_n + (i_{n-1} ± ε)
```

This results in the following functions:

$$\delta_a(\epsilon) = 0,\ \delta_b(\epsilon) = \epsilon$$

△

### 4.3.5. Limitations

When analyzing the specification symbolically or conconically, we cannot decide whether the condition of an if or a trigger evaluates to true or false. Hence, we cannot analyze specifications in which a stream depends on another stream that includes an if-statement or find out whether the variation is sufficient to make the trigger evaluate differently.

Moreover, it is a known problem that symbolic execution does not scale for unbounded iterations. In our case, this means that recursive streams cannot be analyzed via symbolic analysis. Therefore, it is only possible to express $\delta$ as a function of $\epsilon$ for output streams that do not depend on a recursive stream or are recursive themselves. Hence, we bind the evaluation depth to $n$ when analyzing recursion so that $\delta$ is only calculated for the first $n$ evaluation steps. In the following, we present examples in which it is not possible to analyze the specification without depending on the evaluation depth $n$.

**Example 4.3.4.** Consider the following specifications, which are already transformed to vector representation.

→ Sec. 4.2, Page 16

First, consider the specification in Figure 4.2:

```
1 [i_{n-1}, i_n]
2 a_n = a_{n-1} + i_n
```

Here, the input variation of $\epsilon$ depends on the evaluation depth $n$, as it stacks with every iteration: $a_0$ varies by $\epsilon$, $a_1$ by $2\epsilon$, $a_2$ by $3\epsilon$, ..., $a_n$ by $(n+1)\epsilon$. This formula can be represented with a general evaluation depth, although $a$ is a recursive stream. Nevertheless, this is more a special case, because finding closed formulas becomes harder for complex calculations:

```
1 [i_{n-3}, i_{n-2}, i_{n-1}, i_n]
2 a_n = a_{n-1} + a_{n-3} + i_n
```

In this example, we have an unknown recursion tree. Here, the recursion with $n-3$ reaches the base case of the recursion, i.e., the execution of the default case, in less steps as the one with $n-1$. Hence, we can only bound the variation of $a$ by regarding the recursion with $n-1$ twice instead of $n-1$ and $n-3$ — because the variation for $n-1$ is an upper bound for the variation for $n-3$. Nevertheless, to find the exact variation, we still require a fixed evaluation depth $n$. △

Moreover, arithmetic operations — plus and minus excluded — like multiplications, pose a particular problem in connection with recursion.

**Example 4.3.5.** Although we know that a stream, which does not depend on any input stream, has a variation of 0, we cannot refer to them — if another stream depends on them — as 0 in any case. This means that — although $\delta_a = 0$ — $a$ can actually influence the variation of other streams, although it does not vary itself.

```
1 [i_{n-1}, i_n]
2 a_n = a_{n-1} + 1
3 b_n = a + i_{n-1}
```

In this example, $a$ does not depend on input streams and therefore has a variation of 0, i.e., $\delta_a = 0$. As $i_{n-1}$ varies by $\epsilon$, and $a$ has no variation, $b$ has a variation of $\epsilon$ too, i.e., $\delta_b = \epsilon$.

Now we replace the sum of $b$ with a product and consider the following new specification:

```
1 [i_{n-1}, i_n]
2 a_n = a_{n-1} + 1
3 b_n = a * i_{n-1}
```

In this case, $b$ uses $a$ within a multiplication with an input. Therefore, the variation of $b$ depends on the exact value of $a$. In consequence, the $\delta_b$ depends on the exact value of $a$ — although $a$ itself has a variation of 0. Here, we again require a concrete evaluation depth $n$. △

## 4.4. Implementation Details

The described analyses were implemented in *Rust*[1]. The implementation is published on *GitHub*[2]. To interpret the RTLola specification, we use the RTLola parser[3], which transcripts the given specification into an internal representation (IR). From this IR, we can then gain the required constraints. To simplify and solve them, we use an SMT solver, i.e., the *z3* Rust library[4]. Consequently, the constraints need to be represented in the z3 abstract syntax tree (AST).

### 4.4.1. General Implementation

We can define whether the concrete or the symbolic analysis should be executed via input parameters. Both approaches require the specification as well as an evaluation depth $n$. The symbolic analysis requires this value too as it directly calls the concolic robustness analysis on the recursive streams of the specification after it terminates.

Furthermore, we analyze at least $max_{mem}$ iterations — even if a smaller $n$ is given — to guarantee that each calculation rule can be executed at least once, i.e., without executing the default cases.

As shown in Algorithm 1, Algorithm 2, and Algorithm 3, we presented our algorithms implemented *top-down*. This means that we started at the top iteration, i.e. at the evaluation depth $n$ for the concrete and concolic analyses, and $max_{mem}$ for the symbolic analysis, and go downwards to the lowest iteration accessed. This leads to the fact that we only analyze the specific iterations of the streams that are actually accessed. However, the Rust implementation analyzes the streams *bottom-up*, which means that we start at the first iteration and go to the highest iteration, i.e., $max_{mem}$ for the symbolic analysis or $n$ for the concrete and concolic analyses. We store all streams that have

[1] https://www.rust-lang.org/
[2] https://github.com/jessias2/RTLolaRobustness
[3] https://docs.rs/rtlola-frontend/0.3.3/rtlola_frontend/
[4] https://docs.rs/z3/0.9.0/z3/

already been analyzed so that no stream needs to be analyzed more than once. Therefore, we can guarantee that every stream access done in the current iteration goes to a stream that has already been analyzed. This has several advantages for the analyses, which we elaborate on in the following sections.

### 4.4.2. Concrete Robustness Analysis

To gain a more accurate overview of the behavior of the specification, we output the results of every stream in every evaluation step. Therefore, not only the accessed stream iterations may be analyzed. Instead, we analyze all stream iterations, so it does not matter whether bottom-up or top-down implementation is used.

Moreover, we build the true and false condition simultaneously because this is easier than copying the true condition and renaming the variables afterward to get the false condition. Nevertheless, we only analyze streams that influence a trigger, i.e., streams on which any trigger depends, as the goal is to change the outputs of the program. Since the only "real" outputs of an RTLola specification are the results of the triggers, it is redundant to consider streams on which no trigger depends. Consequently, the implementation analyzes the robustness per trigger.

With the goal of changing the output, we also add the condition to the overall constraint that evaluates the trigger condition in the last analyzed iteration of the true condition to true and the false condition to false. This limits the subsequent maximization to changing the output. That is why we maximize only the differences of the output streams of the last iteration and not all of the streams. Moreover, this leads to a significantly improved runtime.

To actually solve the constraints, we cannot represent booleans as an integer as presented in Section 4.2.4. For this, we use the `Bool` type, which is offered by the z3 library. Hence, it is not possible to maximize the output differences of streams with type `Bool`.

### 4.4.3. Symbolic and Concolic Robustness Analysis

Because of the bottom-up implementation and the evaluation order described in Section 4.4.4, we can guarantee that every stream accessed in the current iteration has already been analyzed. Therefore, we do not need to implement a recursive approach or adapt the unknown set over which we are currently iterating. Moreover, it is not possible to modify an already built z3 AST in retrospect which would be necessary in the top-down implementation for inlining the sub-conditions of lower evaluation steps. Hence, bottom-up significantly eases the implementation.

In addition, the symbolic approach analyzes the streams before the $max_{mem}$-th iteration to calculate the base cases of the delta formula. Therefore, we also have an overview of the behavior when default cases are still executed.

### 4.4.4. Evaluation Order

The symbolic and concolic analysis require that streams are sorted increasingly regarding their *evaluation layer*. This layer denotes when a stream should be evaluated, which is not necessarily the declaration order. Consequently, the layer gives us a partial order of evaluation, without which values that have not yet been evaluated may be accessed. Nevertheless, we will not further elaborate on how this layer results. The concrete analysis does not depend on this order, as it does not access the streams but writes a variable into the constraint, which is defined later.

**Example 4.4.1.** Consider the following specification:

```
output x: Int64 := y + 1
output y: Int64 := y.offset(by: -2).defaults(to: 0) + 10
```

We see that $x$ accesses $y$ before $y$ is declared. Hence, $y$ would be unknown when evaluating $x$ and we would assign the default value. What actually happens is that $y$ is evaluated before $x$, since here $y$ has the evaluation layer 1 and $x$ has 2. Consequently, the access is not undefined and the default case is not executed.

As we inline all stream accesses in the symbolic execution, we also depend on the fact that $y$ is already analyzed. Hence, we need to order the streams in increasing order via layer.

Nevertheless, the concrete analysis is independent of this sorting, although it builds the constraints in the declaration order:

```
x_0 = y_0 + 10
y_0 = 0 + 10
x_1 = y_1 + 1
y_1 = 0 + 10
```

Here, no variable is inlined, but simply defined afterward. We just know that the default case must be evaluated if the offset leads to a negative index. Note that this is only possible as we focus on synchronous monitoring. In this case, we know in iteration $n$ that every stream already has been evaluated $n-1$ times, i.e., we can access streams with offset $o$, where $o \in [-(n-1),...,0]$. Every access out of this bounds executes the default case. Otherwise it is possible that one stream holds more values than another stream, and accordingly you can access it with different maximal offsets. △

### 4.4.5. Floating-Point Numbers

Although the presented approaches can be applied on `Float` streams, our implementation does not support them. This is the case because the z3 library does not provide floating-point number types. However, it is possible to approximate them as a rational number by converting them into a fraction. This transformation is not lossless — and robustness could only be estimated. Therefore, floating-point numbers were not implemented.

| line | state |
|---|---|
| 1 | $trigger := (t_1 = b_1 > 10)$ |
| 2 | $unknown := \{b_1\}$ |
| 3 | $true\_condition := \left(t_1^t = b_1^t > 10\right)$ |
| 5 | $u := b_1$ |
| 6 | $new\_condition := (b_1 = a_1 + i_0)$ |
| 7 | $true\_condition := \left(t_1^t = b_1^t > 10\right) \wedge \left(b_1^t = a_1^t + i_0^t > 10\right)$ |
| 8 | $unknown := \{a_1\}$ |
| 5 | $u := a_1$ |
| 6 | $new\_condition := (a_1 = a_0 + 20)$ |
| 7 | $true\_condition := \left(t_1^t = b_1^t > 10\right) \wedge \left(b_1^t = a_1^t + i_0^t > 10\right) \wedge \left(a_1^t = a_0^t + 20\right)$ |
| 8 | $unknown := \{a_0\}$ |
| 5 | $u := a_0$ |
| 6 | $new\_condition := (a_0 = 5 + 20)$ |
| 7 | $true\_condition := \left(t_1^t = b_1^t > 10\right) \wedge \left(b_1^t = a_1^t + i_0^t > 10\right) \wedge \left(a_1^t = a_0^t + 20\right) \wedge \left(a_0^t = 5 + 20\right)$ |
| 8 | $unknown := \{\}$ |
| 10 | $false\_condition := \neg\left(t_1^f = b_1^f > 10\right) \wedge \left(b_1^f = a_1^f + i_0^f > 10\right) \wedge \left(a_1^f = a_0^f + 20\right) \wedge \left(a_0^f = 5 + 20\right)$ |
| 11 | $input\_condition := d\left((i_1^t, i_0^t), (i_1^f, i_0^f)\right) \leq \epsilon$ |
| 12 | $formula := true\_condition \wedge false\_condition \wedge input\_condition$ |
| 13 | $formula := formula \wedge max\left(d\left((a_1^t, a_0^t), (a_1^f, a_0^f)\right)\right) \wedge max\left(d\left((b_1^t, b_0^t), (b_1^f, b_0^f)\right)\right)$ |
| 15 | $input\_true := \left(i_1^t = 0, i_0^t = -3\right)$<br>$input\_false := \left(i_1^f = -3, i_0^f = -6\right)$ |
| 16 | $output\_true := \left(b_1^t = 17, b_0^t = 17, a_1^t = 20, a_0^t = 17, t_1^t = \top, t_0^t = \top\right)$ |
| 17 | $output\_false := \left(b_1^f = 11, b_0^f = 0, a_1^f = 17, a_0^f = 14, t_1^f = \top, t_0^f = \bot\right)$ |
| 18 | $\delta_b := 6$, $\delta_a := 5$, $\delta_t := 1$ |

Table 4.3.: Example execution protocol of the concrete robustness analysis

| line | state |
|---|---|
| 1 | $linear\_streams := \{a, b\}$ |
| 2 | $max\_mem := 2$ |
| 4 | $stream := a$ |
| 5 | $formula := (a_1 = (i_1 \pm \epsilon) + 20)$ |
| 6 | $unknown := \{\}$ |
| 12 | $\delta_a := \epsilon$ |
| 4 | $stream := b$ |
| 5 | $formula := (b_1 = a_1 + (i_0 \pm \epsilon))$ |
| 6 | $unknown := \{a_1\}$ |
| 8 | $u := a_1$ |
| 9 | $analyzed := (a_1 = (i_1 \pm \epsilon) + 20)$ |
| 10 | $formula := (b_1 = ((i_1 \pm \epsilon) + 20) + (i_0 \pm \epsilon))$ |
| 11 | $unknown := \{\}$ |
| 12 | $\delta_b := 2\epsilon$ |

Table 4.5.: Example execution protocol of the symbolic robustness analysis

| line | state |
|---|---|
| 1 | $recursive\_streams := \{a, b\}$ |
| 3 | $stream := a$ |
| 4 | $formula := (a_1 = a_0 + 20)$ |
| 5 | $unknown := \{a_0\}$ |
| 7 | $u := a_0$ |
| 8 | $analyzed := (a_0 = 5 + 20)$ |
| 9 | $formula := (a_1 = 5 + 20 + 20)$ |
| 10 | $unknown := \{\}$ |
| 12 | $\delta_a := 0$ |
| 3 | $stream := b$ |
| 4 | $formula := (b_1 = a_1 + (i_0 \pm \epsilon))$ |
| 5 | $unknown := \{a_1\}$ |
| 7 | $u := a_1$ |
| 8 | $analyzed := (a_1 = a_0 + 20)$ |
| 9 | $formula := (b_1 = (a_0 + 20) + (i_0 \pm \epsilon))$ |
| 10 | $unknown := \{a_0\}$ |
| 7 | $u := a_0$ |
| 8 | $analyzed := (a_0 = 5 + 20)$ |
| 9 | $formula := (b_1 = ((5 + 20) + 20) + (i_0 \pm \epsilon))$ |
| 10 | $unknown := \{\}$ |
| 12 | $\delta_b := \epsilon$ |

Table 4.6.: Example execution protocol of the concolic robustness analysis

Chapter 5

# Evaluation

The proposed analyses allow us to investigate whether an RTLola specification is robust. This robustness is necessary to protect patients from mistreatment triggered by erroneous inputs and provides an essential safety guarantee for MCPS. Thus, it remains to be clarified how the different approaches' outputs are to be interpreted and how they interact to obtain a detailed model of the specification's behavior. To actually make these approaches applicable in practice, it is also necessary to evaluate how efficient the results are calculated.

## 5.1. Results and Output Interpretation

We already know that the single algorithms have different limitations. However, these can be balanced across by considering the approaches' interaction. Accordingly, the methods provide different information that together gives the necessary overall picture. In the following, we will investigate which approach provides which outputs and how they can be used to infer the system's robustness.

The concrete analysis calculates an input pair for which the output is maximized. Therefore, these inputs are a special case of the specification that must be considered during development and testing. Nevertheless, it is difficult to conclude the specification's overall behavior from this one example. To further estimate the behavior, several runs of concrete analysis are necessary.

However, concrete analysis can be used to analyze specifications that cannot be analyzed symbolically or concolically, namely those containing conditions, e.g., in the form of if-then-else directives. For if-cases, it is necessary to decide whether the if-condition is satisfied or not. Therefore, a concrete value is required, while a symbolic value is not sufficient to determine this. Furthermore, the concrete analysis can immediately investigate whether the given input variation is sufficient to switch a trigger from true to false or vice versa. Such a trigger switch can cause a different warning and hence a wrong decision. While the concrete analysis directly provides this information, these

inferences must be drawn from the formulas built by symbolic and concolic analysis since the trigger condition cannot be decided by their symbolic inputs.

The concrete analysis thus provides, as the name suggests, a concrete input pair with which a concrete execution can be performed. To make more general statements about the specification, the symbolic and concolic analysis are required. The symbolic approach calculates a function for every linear stream, which defines the output variation $\delta$ depending on the input variation $\epsilon$. From this formula, it is now possible to read off directly whether the function corresponds to the $(\delta, \epsilon)$-Robustness presented in Definition 7.

→ Sec. 4.1, Page 11

→ Sec. 4.3, Page 25

However, as shown in Section 4.3.4, it is generally impossible to find such a formula for recursive streams. Hence, the symbolic approach is only applicable to linear streams. To represent the variation of recursive streams, the formula must additionally depend on the evaluation depth $n$. The concolic analysis calculates this. Although the concolic analysis can be performed on both linear and recursive streams, it does not provide more value than the symbolic analysis when executed on linear streams. In such a case, the calculated formula would only depend on an additional variable $n$, which is not necessary. Accordingly, the symbolic analysis contains the information of the concolic analysis but offers a more general statement so that the concolic analysis on linear streams is redundant.

It is important to mention that none of the approaches define a fixed metric for robustness. Even if, from a purely mathematical point of view, every non-linear function contradicts the robustness definition presented in Definition 7, the system's behavior can be predicted with the help of the calculated formulas of the symbolic or concolic analysis. This guarantee can be sufficient in certain areas to classify the system as robust and safe. Nevertheless, the user has to assess whether the calculated output variance is still acceptable in the specific application area. Hence, the analysis remains context-independent and can be applied to arbitrary systems with different requirements in measurement error tolerance.

→ Sec. 4.1, Page 11

## 5.2. Validation

The constraints created during the concrete robustness analysis enable various validations, which we will analyze in this section. To recap: the true and false condition built during the concrete analysis represent when the trigger evaluates to true and false, respectively, after $n$ steps. Different conclusions follow these two conditions:

- If and only if the true condition is not satisfiable, the trigger is never taken after $n$ steps.
- If and only if the false condition is not satisfiable, the trigger is always taken after $n$ steps.
- If and only if the true and false conditions are satisfiable, there are two inputs for which one takes the triggers, and the other does not after $n$ steps.

These conclusions can even be strengthened if we consider the overall constraint, i.e., the true, false, and input condition, which allows the inputs to vary by at most $\epsilon$. Hence, if and only if the overall constraint is satisfiable, there are two inputs that vary by at most $\epsilon$, for which one takes the trigger, and the other does not.

This validation is the fundamental basis of the robustness analysis. If we determine that a trigger always evaluates to true or false, the actual output of the specification cannot change. Consequently, the treatment decision cannot change due to a measurement error because the output always remains the same. Since the output variance is also zero, the system is robust in any case. Nevertheless, there may be a severe error that causes the patient to be treated incorrectly. In the previous Example 4.1.3 of AP, → Sec. 4.1, Page 15 this means that a patient permanently believes that he is suffering from hyperglycemia and administers insulin, which is not caused by a measurement error but by a programming error in the specification triggering this false warning.

In conclusion, robustness analysis is only useful if it is guaranteed that the trigger — as the specification's actual output — can change. Therefore, this free sideproduct is used before the robustness analysis to exclude such mistakes and to have a solid foundation for robustness analysis.

## 5.3. Runtime

To make the robustness analysis practicable, it must not have an excessively long runtime that would significantly slow down the development process. To evaluate this, we tested the runtime of the algorithms by executing the *release build* of the Rust implementations with different specifications on an *Intel Core i7-9750H*. Here, we linearly changed one parameter while fixing the others to see what runtime effects the individual parameters have. Each test case was executed 100 times. To avoid measurement inaccuracies caused by caching, we waited 2 seconds before testing the next case. Nevertheless, slight fluctuations in the runtime can in general be caused by scheduling.

We focus on the runtime of the already presented specifications from Figure 3.3 and → Sec. 3.2, Page 8 Figure 4.2. The evaluation of two additional specifications that were used for testing → Sec. 4.2, Page 16 can be seen in Appendix A. Each of the following graphs presenting the runtime is constructed as follows: the x-axis shows the parameter that is increased, and the y-axis shows the corresponding runtime in seconds. For comparing different test cases among each other, we used the average execution time of 100 runs which are represented by markers. Additionally, we added a *LOWESS*[1] trendline for every test to get an overview of the runtime development.

### Notation

Let *#stream* be the number of streams within the specification, and *#linear_streams* and *#recursive_streams* the number of linear and recursive streams, respectively.

[1] locally weighted regression

### 5.3.1. Symbolic Analysis and Concolic Analysis

The symbolic analysis is only executed on linear streams, and the concolic analysis is automatically applied on the recursive streams of the given specification afterwards. As already stated in Example 4.3.2 and Example 4.3.1, Figure 3.3 contains only linear streams, whereas Figure 4.2 contains only recursive streams. If we run both specifications, we can draw conclusions about the runtime of the symbolic analysis by Figure 3.3, and about the concolic analysis by Figure 4.2.

→ Sec. 3.2, Page 8
→ Sec. 4.2, Page 16

The symbolic analysis is independent of the evaluation depth $n$, as it always analyzes $max_{mem}$ values. Hence, the runtime remains constant when increasing $n$ which can be seen in Figure 5.1a. Theoretically, this analysis evaluates $max_{mem} \cdot (\#linear\_streams)$ streams. Accordingly, the runtime rises for a specification with a higher $max_{mem}$ or more linear streams. Here, the runtime increases linearly when increasing only one of those two parameters, and squares when both are increased.

→ Sec. 5.3, Page 38

However, this does not apply to the concolic approach shown in Figure 5.1b. Here, $n \cdot (\#recursive\_streams)$ streams are analyzed, so that the runtime also depends on the selected evaluation depth. Hence, incrementing the evaluation depth increases the runtime linearly, which is also shown in the graph. Correspondingly, there is again linear growth when one of the factors becomes larger, and quadratic growth when both are increased.

→ Sec. 5.3, Page 38

### 5.3.2. Concrete Analysis

In the concrete analysis, $n \cdot (\#streams)$ streams are evaluated. We can therefore directly assume that again the runtime increases linearly if one of these factors increases when fixing $\epsilon$ to $\epsilon = 3$. This is true for the pure building of the constraints, which is shown in Figure 5.2, but not for the solving of them. The only thing that can be observed is that the constraint with the Manhattan norm is built slightly slower. This is due to the fact that this input condition requires more partial conditions to be built. However, this difference is not significant in the overall approach. This is shown in Figure 5.3, where we see the runtime which includes solving the constraint, i.e., with executing z3. Here, we see directly that the runtime increases significantly, and more strongly than linearly. This becomes especially noticeable in direct comparison to the runtime for building the constraints from Figure 5.2. The increase in runtime is due to the increasing complexity of the formulas, and thus also to the more complex optimization task for the z3 solver. Also remarkable is the fact that the runtime of the Manhattan norm is significantly higher compared to the Chebyshev norm for the linear specification, which can be seen in Figure 5.3a — whereas Figure 5.3b indicates that for recursive specifications it is the other way around. However, to find out exactly why this is the case, the internals of z3 have to be investigated, which we will not do this at this point.

→ Sec. 5.3, Page 39
→ Sec. 5.3, Page 40

Moreover, the concrete approach depends on the additional input $\epsilon$. The influence of $\epsilon$ can be seen in Figure 5.4 for which $\epsilon$ was incremented while the evaluation depth $n$ was fixed to $n = 100$. We see that the runtime for both specification remains nearly

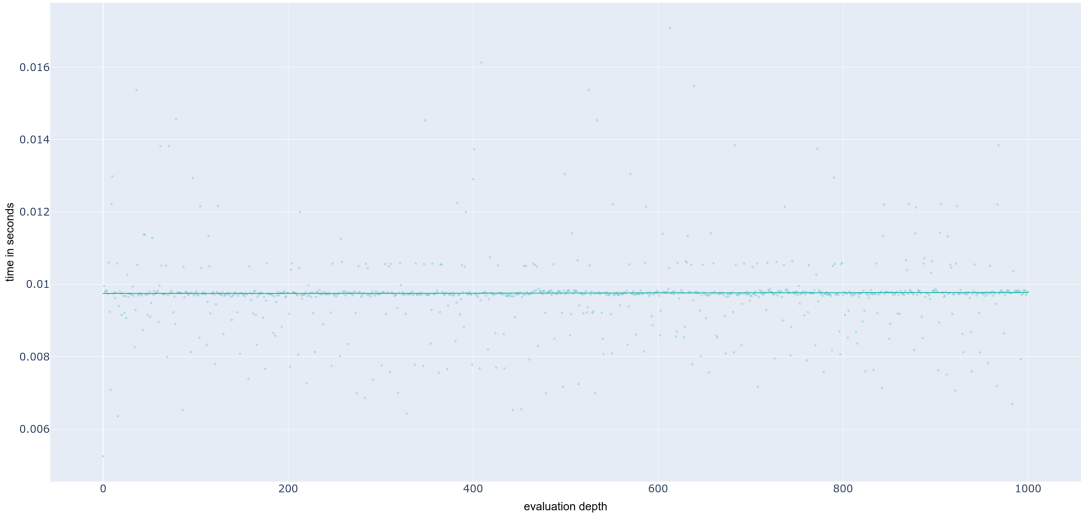
→ Sec. 5.3, Page 41

constant even for increasing iterations for the respective norm. Therefore, we conclude that the choice of $\epsilon$ has no significant influence on the runtime.

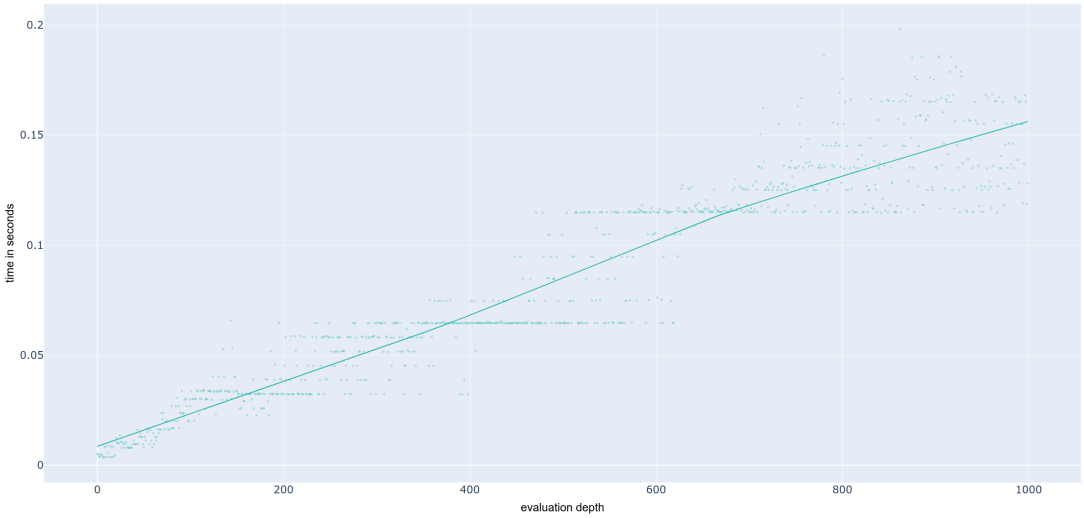
### 5.3.3. Overall Runtime

In summary, the runtime depends on a variety of factors, with $\epsilon$ shown to be the only input that does not affect runtime. Looking at the exact runtimes of the symbolic and concolic analysis, we see that the examples' runtimes, even for high iterations, is well below one second. However, the runtime increases dramatically when the built constraints are to be solved, as is the case in the concrete analysis, but still, the runtimes are in the seconds range. Additionally, we see plateaus within the runtimes. Random samples indicate that these are influenced by different caching behavior of the hardware and interrupts of the operating system. We do not have a more detailed explanation for this. nevertheless, this has no significant influence on the runtime since it is only a matter of a few seconds, usually microseconds.

Overall, the symbolic and concolic analyses scale well for large specifications and high evaluation depths. Only the runtime of the concrete analysis then increases sharply. However, this is again at the discretion of the user, who must decide whether the time required for robustness and safety guarantees is justified or not. Nevertheless, the approaches are all in one well applicable in the development process of a specification, and thus not only an important but also a practicable guarantee that MCPS must support.

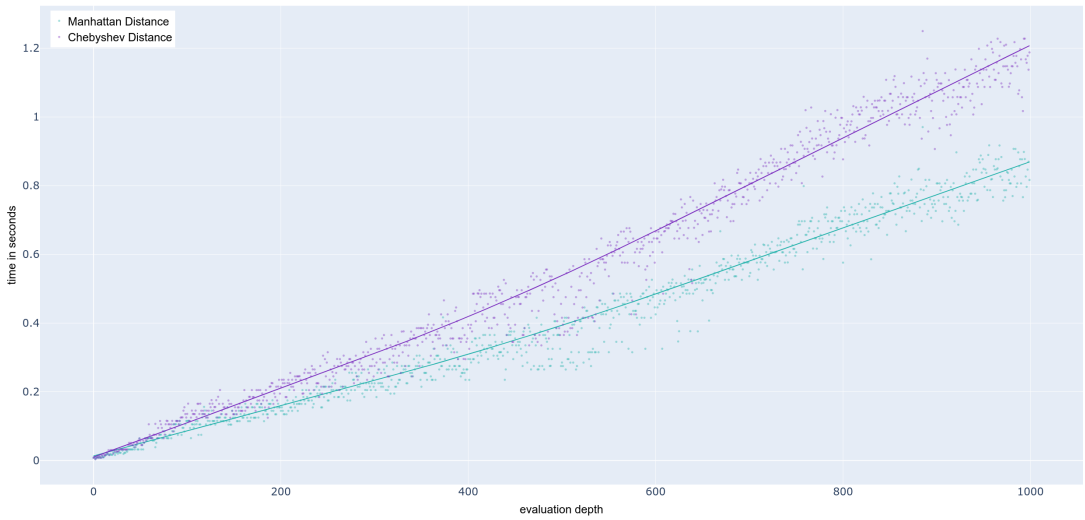



(a) Runtime of the symbolic analysis while incrementing the evaluation depth

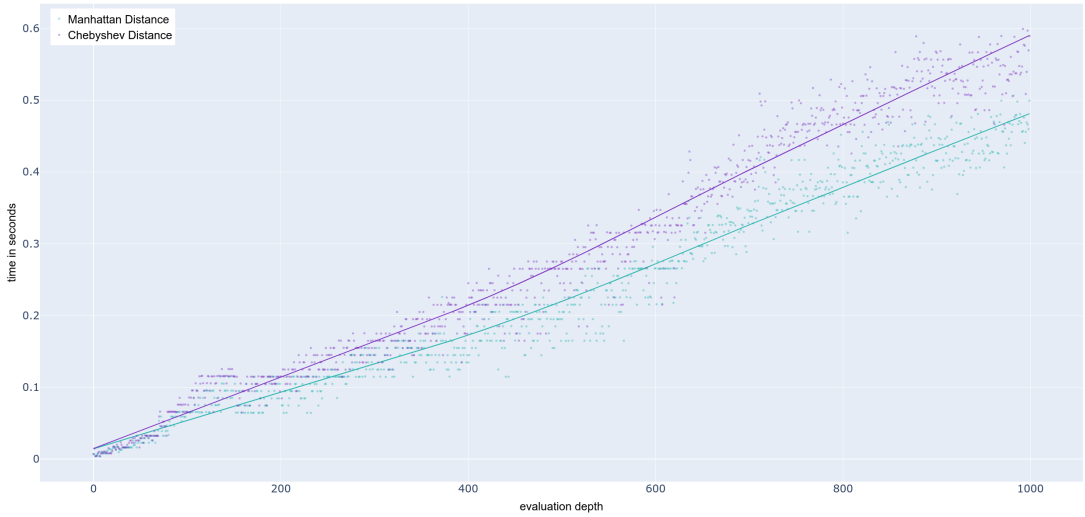



(b) Runtime of the concolic analysis while incrementing the evaluation depth

Figure 5.1.: Runtimes of the symbolic and concolic analysis

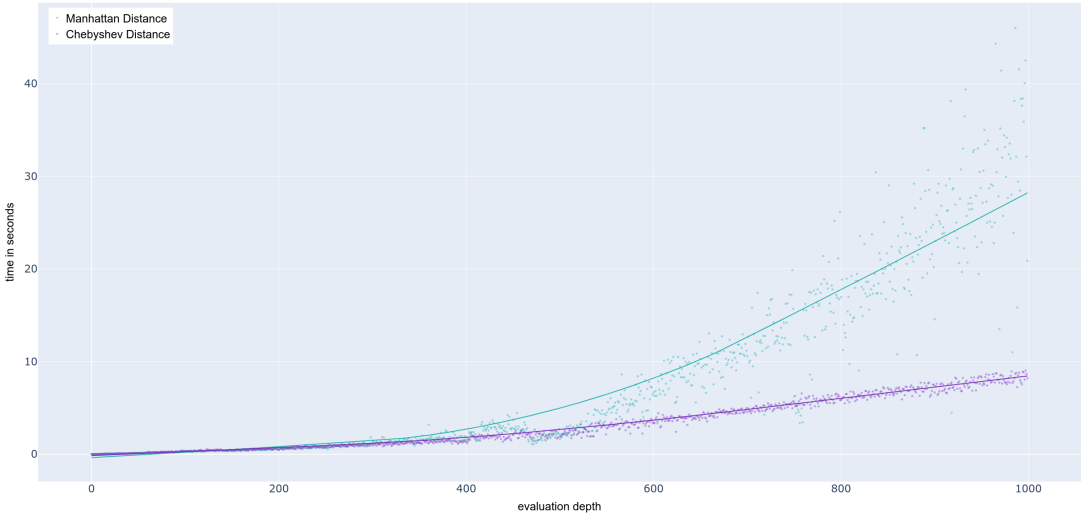



(a) Runtime of the specification including linear streams

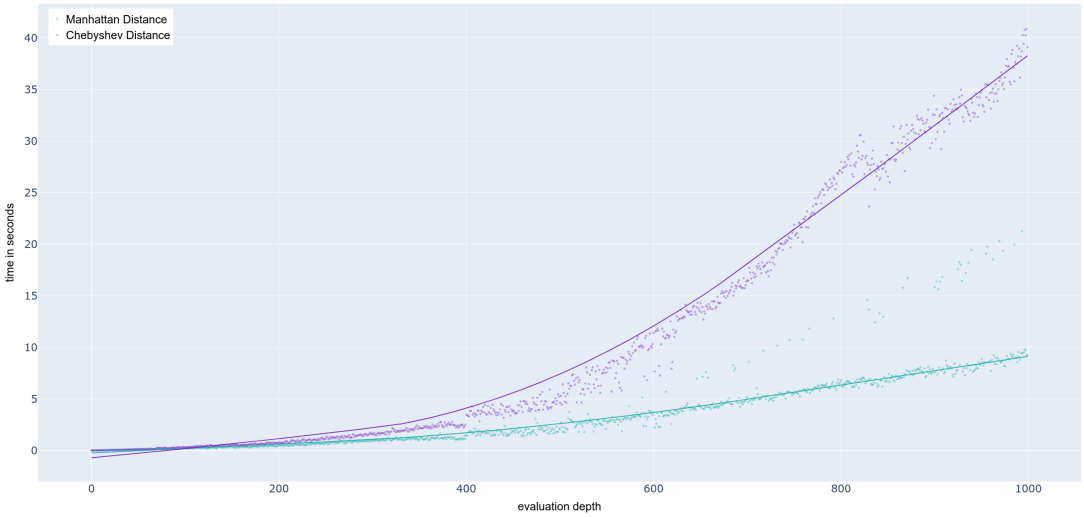



(b) Runtime of the specification including recursive streams

Figure 5.2.: Runtimes of the concrete analysis for building constraints (excl. z3 solving) while incrementing the evaluation depth

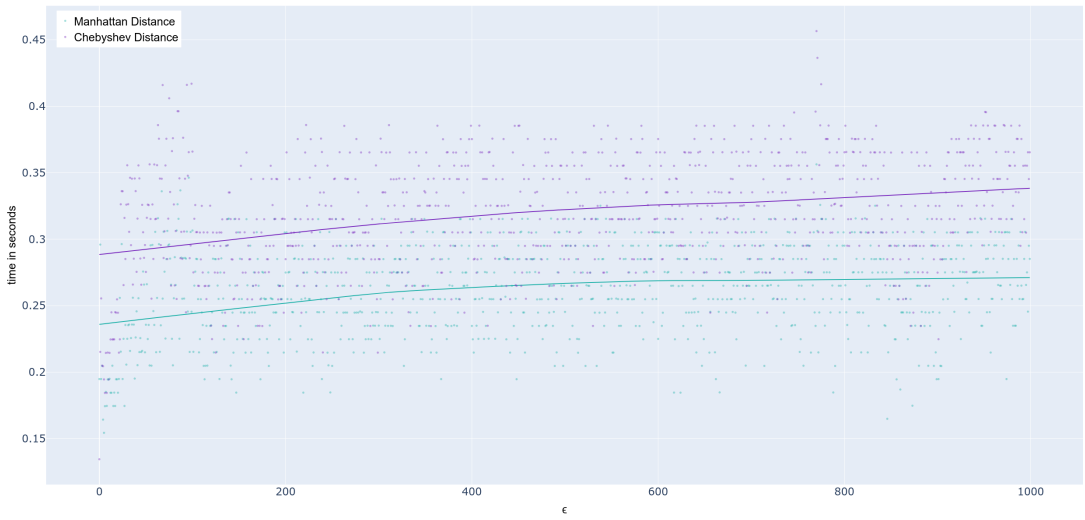



(a) Runtime of the specification including linear streams

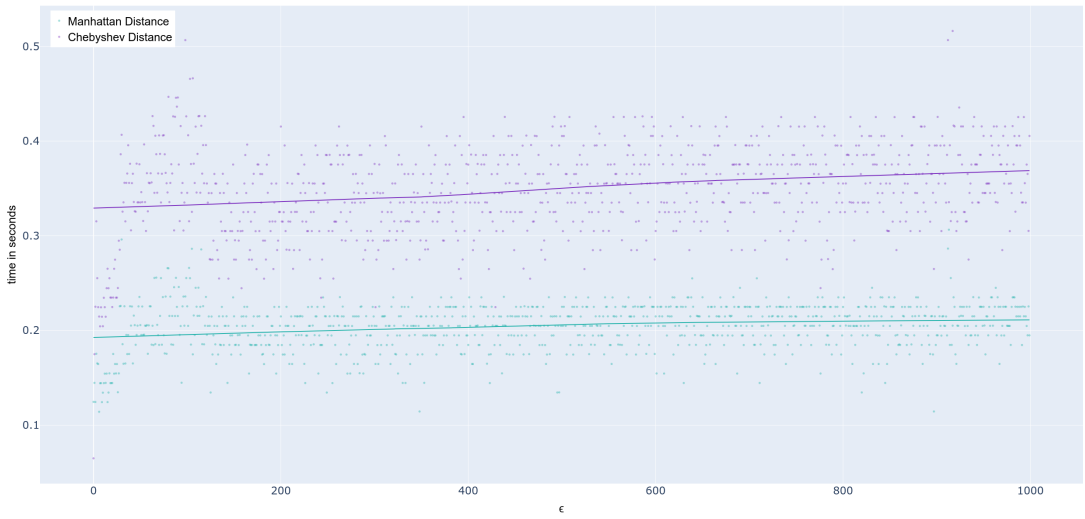



(b) Runtime of the specification including recursive streams

Figure 5.3.: Runtimes of the complete concrete analysis (incl. z3 solving) while incrementing the evaluation depth




(a) Runtime of the specification including linear streams




(b) Runtime of the specification including recursive streams

Figure 5.4.: Runtimes of the complete concrete analysis (incl. z3 solving) while incrementing $\epsilon$

Chapter 6

# Discussion

Robust MCPS are necessary to protect patients from harmful mistreatment. It therefore supports the general medical goal of ensuring the safety of a treatment and protecting the patient from health risk. However, robustness also offers crucial guarantees in other application areas.

## 6.1. Robustness Analysis for Safety-Critical CPS

Robustness can also be transferred to other safety-critical areas such as autonomous cars and unmanned aircraft systems. In the worst case scenario, people can also be injured in accidents involving these systems, which must be avoided at all costs. Even if no one is harmed, the economic damage is very high. Consequently, these systems must be as safe as possible, so that wrong decisions due to measurement errors must be avoided and so robustness plays a significant role here as well.

For illustration, consider the example of a drone, and the possible effects of measurement errors. Drones can examine their surroundings with the help of sensors and, for example, find a path between two houses which is should be flown. However, if a sensor measures incorrectly, the drone may estimate the distances to the houses wrong and crash into one of them. If this drone was based on a robust system, it would recognize a measurement error as such. Therefore, the drone's tolerance to input variations would increase, so that the drone could be expected to fly a similar route for similar measurements. This also leads to the acceptance of smaller measurement errors. Overall, the drone's flight path becomes more predictable and accidents can be avoided.

Chapter 7

# Conclusion

In this thesis, we presented a robustness analysis for RTLola specifications. A system is called robust if a variation of at most $\epsilon$ in the inputs causes a variation of at most $\delta$ in the outputs. We decide between three approaches: the concrete analysis, the symbolic analysis, and the concolic analysis.

The concrete method analyzes the specification and builds different constraints on the behavior simultaneously. Therefore, we find an input pair which maximizes the output difference $\delta$ while the inputs vary by at most $\epsilon$. We can use this example a basis for which the specification can be tested and further adapted and can investigate, whether $\epsilon$ is enough to change the trigger condition and cause a different output. Nevertheless, the concrete approach depends on a fixed $\epsilon$ and a fixed evaluation depth. A generalization is offered by the symbolic analysis, which expresses the output variation $\delta$ as a formula of $\epsilon$. Since symbolic analysis is only possible for linear streams, it is complemented by concolic analysis, whose output formulas additionally depend on a given evaluation depth. Because each system has different requirements regarding deviations — and consequently, no general robustness metric can be specified — it is the user's responsibility to decide whether the calculated output variation is still acceptable in the specific application area.

All in one, these analyses allow us to investigate how far the output of a system changes if, e.g., a measurement error of a sensor causes a deviation by $\epsilon$ in the input. Therefore, robustness is essential for MCPS, which are not allowed to make wrong decisions even in case of measurement errors and thus harm a patient. With the calculated formulas, we can thus assess whether a measurement error can trigger an inappropriate treatment. In consequence, the system can be adapted until it is considered robust which prevents the patient from health risk.

## 7.1. Final Remarks and Future Work

In future work, we aim to extend the supported RTLola grammar of the robustness analysis. Here, it is particularly important to calculate the robustness of specifications including floats and real-time streams. With these, it is possible to analyze more generic specifications, which are especially required in MCPS. Artificial Pancreata (AP) for example measure the glucose values in blood. As these systems warn from hyper- and hypoglycemia, i.e., too high and low blood glucose values, it is typical for AP to observe the glucose levels over time. This can only be expressed with real-time streams, for which timed offsets are required. In addition, another major goal is to adapt the robustness analyses to asynchronous monitoring.

Moreover, writing specifications by hand, which contribute to the medical goal of a patient-specific treatment, is almost impossible because the measured values include noise, gaps, or errors, and are in general difficult to process due to their complexity. The glucose level, for example, depends not only on how much insulin is currently present in the blood, but also on the patients' dietary and activity behaviors. Consequently, finding a treatment specific to the patient — and therefore also making robust predictions — pose a hard problem.

Current research is conducted to apply machine learning in the safety-critical context of medical implants. Here, the goal is to cleanse data, filter noise, close gaps and recognize patterns. Nevertheless, it is controversial to use machine learning in health-critical contexts as wrong decisions can quickly lead to severe health risks. Additionally, machine learning models lack explainability and traceability. Consequently, especially for systems including machine learning, robustness plays a major role. To make RTLola completely applicable to MCPS, machine learning directives are required, which is the long term goal of RTLola. Here, it is especially interesting to investigate how the usage of machine learning influences the specification's robustness, as the complex models would hardly be predictable and reliable without robustness.

# List of Figures

# List of Tables

Appendix A

# Evaluation: Further Specifications

The Rust implementation was further tested on the specifications presented in Figure A.1 and Figure A.2. Note that Figure A.1 only includes linear streams so that it is used for testing the symbolic analysis, whereas Figure A.2 only includes recursive streams and is used for the concolic analysis. Moreover, we used the same settings described in Section 5.3. Additionally, the results of the tests, which can be seen in Figure A.3, Figure A.4, Figure A.5, and Figure A.6, behave as described there.

→ Sec. 5.3, Page 35

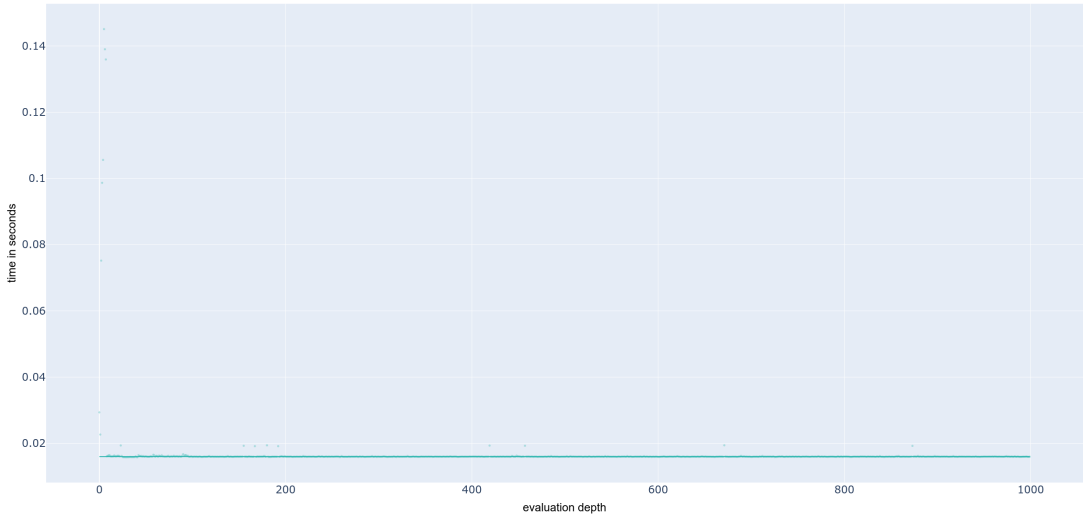
Due to the size of the linear specification in Figure A.1 only a few evaluation steps were tested in some cases, however the low evaluation depths are already sufficient to estimate how drastically the runtime increases with large specifications.

```
input i: Int64
output a0: Int64 := i + i
trigger a0 > 0
output a1: Int64 := i + i
trigger a1 > 1
output a2: Int64 := i + i
trigger a2 > 2
output a3: Int64 := i + i
trigger a3 > 3
output a4: Int64 := i + i
trigger a4 > 4
output a5: Int64 := i + i
trigger a5 > 5
output a6: Int64 := i + i
trigger a6 > 6
output a7: Int64 := i + i
trigger a7 > 7
output a8: Int64 := i + i
trigger a8 > 8
output a9: Int64 := i + i
trigger a9 > 9
output a10: Int64 := i + i
trigger a10 > 10
output a11: Int64 := i + i
trigger a11 > 11
output a12: Int64 := i + i
trigger a12 > 12
output a13: Int64 := i + i
trigger a13 > 13
output a14: Int64 := i + i
trigger a14 > 14
output a15: Int64 := i + i
trigger a15 > 15
output a16: Int64 := i + i
trigger a16 > 16
output a17: Int64 := i + i
trigger a17 > 17
output a18: Int64 := i + i
trigger a18 > 18
output a19: Int64 := i + i
trigger a19 > 19
output a20: Int64 := i + i
trigger a20 > 20
...
output a99: Int64 := i + i
trigger a99 > 99
```

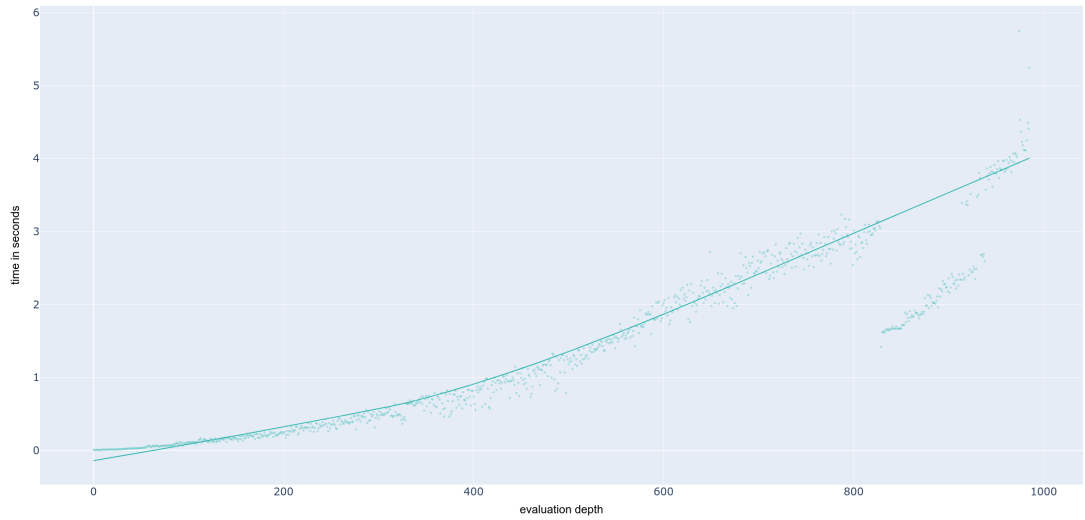
Figure A.1.: Large RTLola specification including 99 linear streams and triggers

```
input i: Int64
output a: Int64 := b + i
output b: Int64 := a.offset(by: -1).defaults(to: 0) + 2
output c: Int64 := b
trigger c > 5
```

Figure A.2.: RTLola specification including circular dependencies

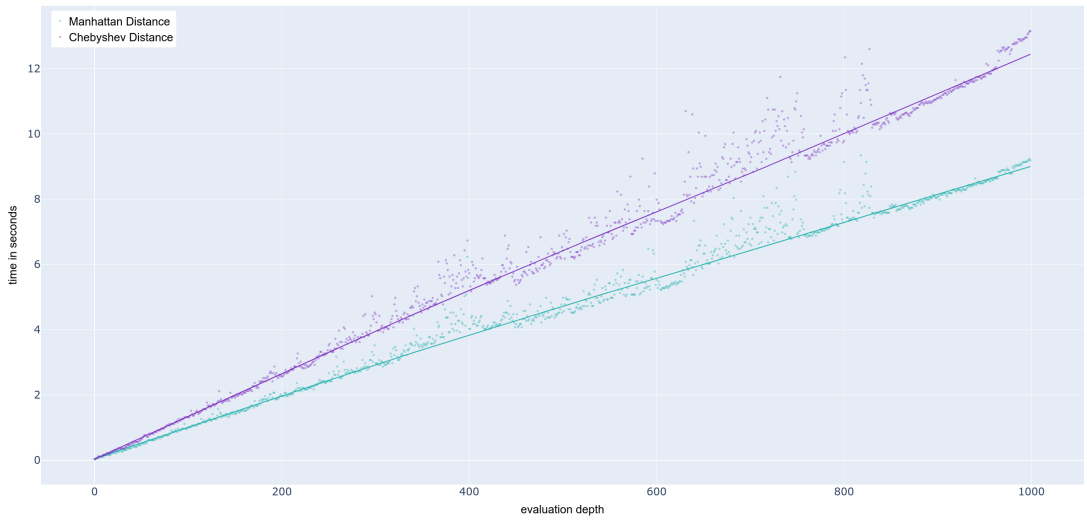



(a) Runtime of the symbolic analysis while incrementing the evaluation depth

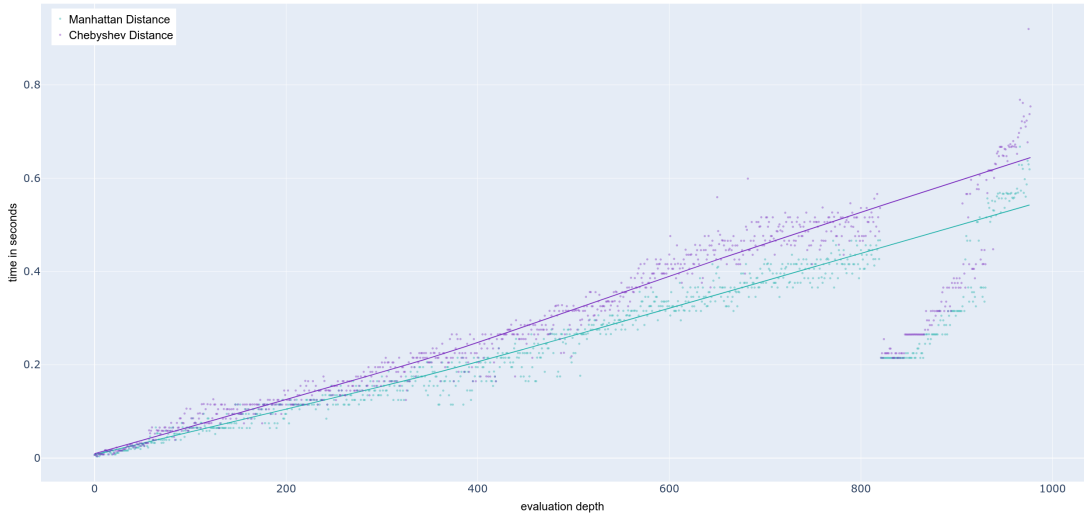



(b) Runtime of the concolic analysis while incrementing the evaluation depth

Figure A.3.: Runtimes of the symbolic and concolic analysis

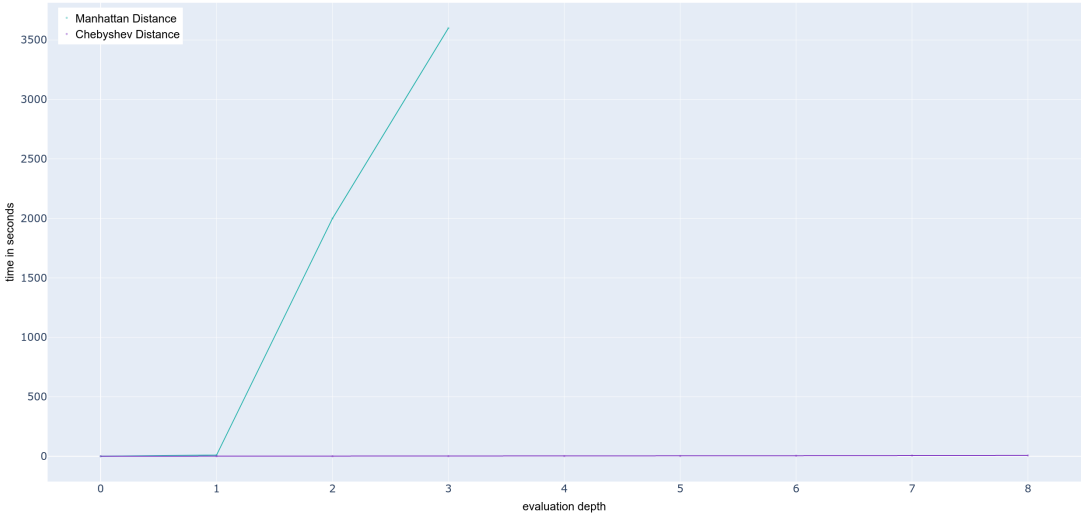



(a) Runtime of the specification including linear streams

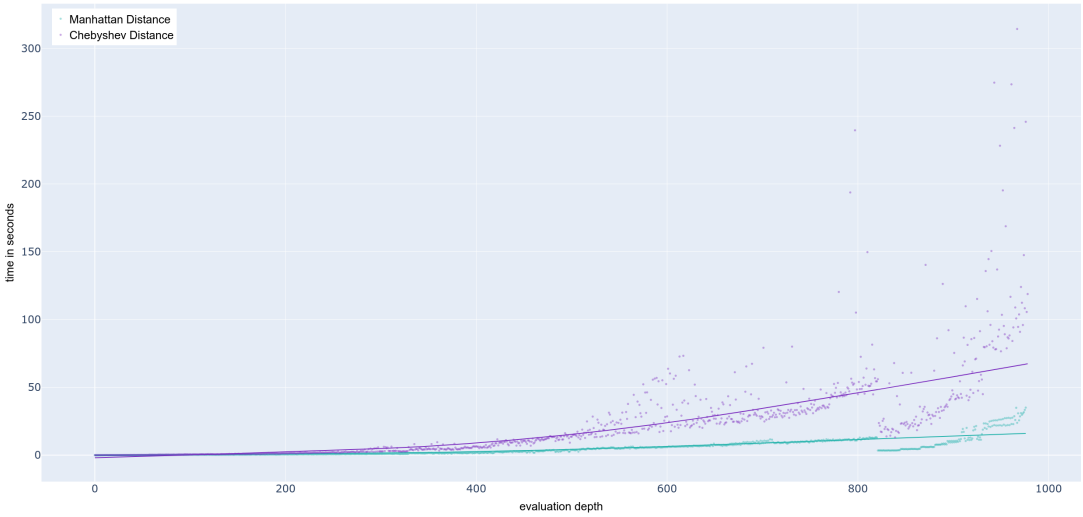



(b) Runtime of the specification including recursive streams

Figure A.4.: Runtimes of the concrete analysis for building constraints (excl. z3 solving) while incrementing the evaluation depth

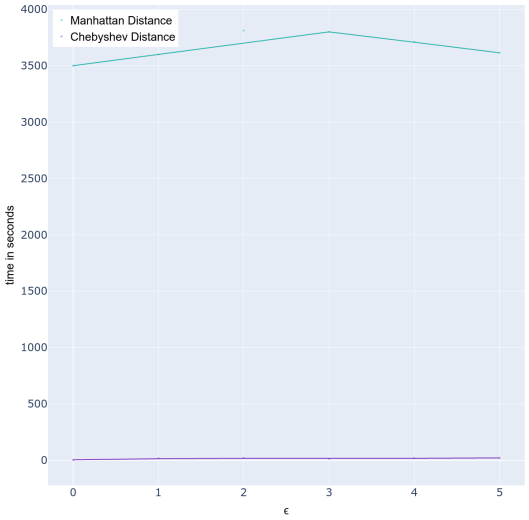



(a) Runtime of the specification including linear streams

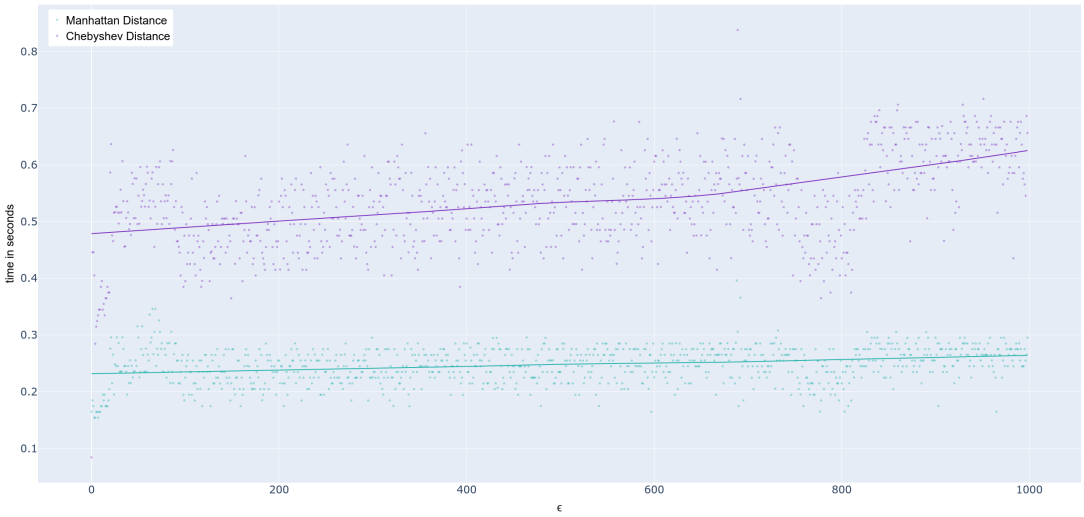



(b) Runtime of the specification including recursive streams

Figure A.5.: Runtimes of the complete concrete analysis (incl. z3 solving) while incrementing the evaluation depth




(a) Runtime of the specification including linear streams




(b) Runtime of the specification including recursive streams

Figure A.6.: Runtimes of the complete concrete analysis (incl. z3 solving) while incrementing $\epsilon$

# Bibliography


[1] Claudio Cobelli, Eric Renard, and Boris Kovatchev. 2011. Artificial pancreas: past, present, future. *Diabetes*, 60, (November 2011), 2672–82. DOI: 10.2337/db11-0654.

[2] Vincent Aravantinos. 2018. Traceability of deep neural networks. *CoRR*, abs/1812.06744. arXiv: 1812.06744. http://arxiv.org/abs/1812.06744.

[3] Francesco Leofante, Nina Narodytska, Luca Pulina, and Armando Tacchella. 2018. Automated verification of neural networks: advances, challenges and perspectives. *CoRR*, abs/1805.09938. arXiv: 1805.09938. http://arxiv.org/abs/1805.09938.

[4] Peter Faymonville, Bernd Finkbeiner, Malte Schledjewski, Maximilian Schwenger, Marvin Stenger, Leander Tentrup, and Hazem Torfah. 2019. Streamlab: stream-based monitoring of cyber-physical systems. In (Lecture Notes in Computer Science). Isil Dillig and Serdar Tasiran, editors. Volume 11561. Springer, 421–431.

[5] Ben D'Angelo, Sriram Sankaranarayanan, César Sánchez, Will Robinson, Bernd Finkbeiner, Henny B. Sipma, Sandeep Mehrotra, and Zohar Manna. 2005. Lola: runtime monitoring of synchronous systems. In IEEE Computer Society Press, Burlington, Vermont, (June 2005), 166–174.

[6] Maximilian Schwenger. 2019. *Let's not Trust Experience Blindly: Formal Monitoring of Humans and other CPS*. Master Thesis. Saarland University.

[7] Florian-Michael Adolf, Peter Faymonville, Bernd Finkbeiner, Sebastian Schirmer, and Christoph Torens. 2017. Stream runtime monitoring on UAS. In *Runtime Verification - 17th International Conference, RV 2017, Seattle, WA, USA, September 13-16, 2017, Proceedings* (Lecture Notes in Computer Science). Shuvendu K. Lahiri and Giles Reger, editors. Volume 10548. Springer, 33–49. DOI: 10.1007/978-3-319-67531-2_3. https://doi.org/10.1007/978-3-319-67531-2%5C_3.

[8] C McPhail, HR Maier, JH Kwakkel, M Giuliani, A Castelletti, and S Westra. 2018. Robustness metrics: how are they calculated, when should they be used and why do they give different results? *Earth's Future*, 6, 2, 169–191.

[9] Chih-Hong Cheng, Frederik Diehl, Yassine Hamza, Gereon Hinz, Georg Nührenberg, Markus Rickert, Harald Ruess, and Michael Truong-Le. 2017. Neural networks for safety-critical applications - challenges, experiments and perspectives. *CoRR*, abs/1709.00911. arXiv: 1709.00911. http://arxiv.org/abs/1709.00911.

[10] Mark Sendak, Michael Gao, Marshall Nichols, Anthony Lin, and Suresh Balu. 2019. Machine learning in health care: a critical appraisal of challenges and opportunities. *eGEMs*, 7, 1.

[11] Alexander Selvikvåg Lundervold and Arvid Lundervold. 2019. An overview of deep learning in medical imaging focusing on mri. *Zeitschrift für Medizinische Physik*, 29, 2, 102–127.

[12] Jose Roberto Ayala Solares, Francesca Elisa Diletta Raimondi, Yajie Zhu, Fatemeh Rahimian, Dexter Canoy, Jenny Tran, Ana Catarina Pinho Gomes, Amir H Payberah, Mariagrazia Zottoli, Milad Nazarzadeh, et al. 2020. Deep learning for electronic health records: a comparative review of multiple deep neural architectures. *Journal of Biomedical Informatics*, 101, 103337.

[13] DF Bedford, G Morgan, and J Austin. 1996. Requirements for a standard certifying the use of artificial neural networks in safety critical applications. In *Proceedings of the international conference on artificial neural networks*. Citeseer.

[14] Sina Mohseni, Mandar Pitale, Vasu Singh, and Zhangyang Wang. 2019. Practical solutions for machine learning safety in autonomous vehicles. *arXiv preprint arXiv:1912.09630*.

[15] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. 2017. Towards practical verification of machine learning: the case of computer vision systems. *CoRR*, abs/1712.01785. arXiv: 1712.01785. http://arxiv.org/abs/1712.01785.

[16] Karsten Scheibler, Leonore Winterer, Ralf Wimmer, and Bernd Becker. 2015. Towards verification of artificial neural networks. In *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen, MBMV 2015, Chemnitz, Germany, March 3-4, 2015*. Ulrich Heinkel, Daniel Kriesten, and Marko Rößler, editors. Sächsische Landesbibliothek, 30–40.

[17] Jean-Baptiste Jeannin, Khalil Ghorbal, Yanni Kouskoulas, Ryan Gardner, Aurora Schmidt, Erik Zawadzki, and André Platzer. 2015. Formal verification of acas x, an industrial airborne collision avoidance system. In *2015 International Conference on Embedded Software (EMSOFT)*. IEEE, 127–136.

[18] Peter Faymonville, Bernd Finkbeiner, Malte Schledjewski, Maximilian Schwenger, Leander Tentrup, and Hazem Torfah. [n. d.] Real-time stream monitoring with streamlab.

[19] Jan Baumeister, Bernd Finkbeiner, Maximilian Schwenger, and Hazem Torfah. 2020. FPGA stream-monitoring of real-time properties. *CoRR*, abs/2003.12477. arXiv: 2003.12477. https://arxiv.org/abs/2003.12477.

[20] Jan Baumeister, Bernd Finkbeiner, Sebastian Schirmer, Maximilian Schwenger, and Christoph Torens. 2020. Rtlola cleared for take-off: monitoring autonomous aircraft. (2020). arXiv: 2004.06488 [cs.RO].

[21] Peter Faymonville, Bernd Finkbeiner, Sebastian Schirmer, and Hazem Torfah. 2016. A stream-based specification language for network monitoring. In *Runtime Verification - 16th International Conference, RV 2016, Madrid, Spain, September 23-30, 2016, Proceedings* (Lecture Notes in Computer Science). Yliès Falcone and César Sánchez, editors. Volume 10012. Springer, 152–168. DOI: 10.1007/978-3-319-46982-9_10. https://doi.org/10.1007/978-3-319-46982-9%5C_10.

[22] Michael R. Clarkson and Fred B. Schneider. 2008. Hyperproperties. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium, CSF 2008, Pittsburgh, Pennsylvania, USA, 23-25 June 2008*. IEEE Computer Society, 51–65. DOI: 10.1109/CSF.2008.7. https://doi.org/10.1109/CSF.2008.7.

[23] Michael R. Clarkson and Fred B. Schneider. 2010. Hyperproperties. *J. Comput. Secur.*, 18, 6, 1157–1210. DOI: 10.3233/JCS-2009-0393. https://doi.org/10.3233/JCS-2009-0393.

[24] Amir Pnueli. 1977. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*. IEEE Computer Society, 46–57. DOI: 10.1109/SFCS.1977.32. https://doi.org/10.1109/SFCS.1977.32.

[25] Oded Maler and Dejan Nickovic. 2004. Monitoring temporal properties of continuous signals. In *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems, Joint International Conferences on Formal Modelling and Analysis of Timed Systems, FORMATS 2004 and Formal Techniques in Real-Time and Fault-Tolerant Systems, FTRTFT 2004, Grenoble, France, September 22-24, 2004, Proceedings* (Lecture Notes in Computer Science). Yassine Lakhnech and Sergio Yovine, editors. Volume 3253. Springer, 152–166. DOI: 10.1007/978-3-540-30206-3_12. https://doi.org/10.1007/978-3-540-30206-3%5C_12.

[26] IV Sergienko and VP Shylo. 2006. Problems of discrete optimization: challenges and main approaches to solve them. *Cybernetics and Systems Analysis*, 42, 4, 465–482.

[27] Sofia Bekrar, Chaouki Bekrar, Roland Groz, and Laurent Mounier. 2011. Finding software vulnerabilities by smart fuzzing. In *Fourth IEEE International Conference on Software Testing, Verification and Validation, ICST 2011, Berlin, Germany, March 21-25, 2011*. IEEE Computer Society, 427–430. DOI: 10.1109/ICST.2011.48. https://doi.org/10.1109/ICST.2011.48.

[28] Rupak Majumdar and Indranil Saha. 2009. Symbolic robustness analysis. In *Proceedings of the 30th IEEE Real-Time Systems Symposium, RTSS 2009, Washington, DC, USA, 1-4 December 2009*. Theodore P. Baker, editor. IEEE Computer Society, 355–363. DOI: 10.1109/RTSS.2009.17. https://doi.org/10.1109/RTSS.2009.17.

[29] Scott Speaks. 2010. Reliability and mtbf overview. *Vicor reliability engineering*.

[30] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. 2018. A survey of symbolic execution techniques. *ACM Comput. Surv.*, 51, 3, 50:1–50:39. DOI: 10.1145/3182657. https://doi.org/10.1145/3182657.