École polytechnique de Louvain

The $\triangle$Q Oscilloscope: Real-Time Observation of Large Erlang Applications using $\triangle$QSD

Author: **Francesco NIERI**
Supervisor: **Peter VAN ROY**
Readers: **Neil DAVIES, Tom BARBETTE, Peer STRITZINGER**
Academic year 2024–2025
Master [120] in Computer Science

Abstract

It is difficult to study the detailed behaviour of large distributed systems while they are running. Many important questions are hard to answer. What happens when there is an overload? How can we feel something is wrong with the system early enough?

The purpose of the thesis is to create the ΔQ oscilloscope, a real time graphical dashboard that can be used to study the behaviour of running Erlang systems and explore tradeoffs in system design. It is based on the principles of ΔQSD.

Furthermore, we have developed an Erlang interface, the ΔQ adapter (`dqsd_otel`). It allows sending real time insights about the running system to the oscilloscope. The adapter works on top of the OpenTelemetry framework macros. This allows for users who have already instrumented their applications with OpenTelemetry to easily integrate the interface.

The oscilloscope performs statistical computations on the time series data it receives from the adapter and displays the results in real time, thanks to the ΔQSD paradigm. We provide a set of triggers to capture rare events, like an oscilloscope would, and give a snapshot of the system under observation, as if it was frozen in time. An implementation of a textual syntax allows the creation of outcome diagrams which give an "observational view" of the system. Additionally, the implementation of efficient algorithms for complex operations, such as convolution, allows for the computations to be done rapidly on precise representations of components.

We introduce the work by giving a summary of ΔQSD concepts. We also provide a summary of the observability tools available for Erlang, namely, OpenTelemetry. We then present the overall design of the project, describing how to build a bridge from OpenTelemetry to the oscilloscope. Subsequently, we explain the user level concepts which are essential to understand how the oscilloscope works and understand what is displayed on the screen, delving later on into the mathematical foundations of the concepts. Lastly, we provide synthetic applications which prove the soundness of ΔQSD and show how the oscilloscope is able to detect problems in a running system, diagnose it and explore design tradeoffs.

Acknowledgments

This thesis is the culmination of my studies, I would like to thank the people who made this possible, those who supported me through the years and those who helped me with the thesis.

My family, especially **my mom, my dad, my brother** and **my sister**, for their help. They were a crucial shoulder I could lay on while writing this thesis and most importantly throughout these five years.

My **friends**, to those who have come from Italy and have taken time out of their lives to listen the thesis presentation, and to those who through the years have been there for me.

A-M., for the moments we shared these four years together in uni.

My dad and **Maurizio**, who nurtured the passion for coding in me.

Peer Stritzinger, **Stritzinger GmbH** and the **EEF Observability Working Group** (Bryan Naegele and Tristan Sloughter) for their help in the EEF Slack, which helped me understand OpenTelemetry and gave me the send_after intuition.

Neil Davies for taking the time to proofread my thesis.

The **PNSol Ltd.** team for their extensive groundwork on **∆QSD** and its dissemination, which made this thesis possible.

Lastly, **Peter Van Roy**, for his year-long relentless interest, support and weekly and constant supervision which made sure the project would come to fruition.

AI disclaimer

AI was employed to help with the graphical dashboard in C++ and the triggers, in positioning the elements, refactoring the code so the widgets would properly interact together, helping understand the FFT algorithm and refactoring the server when communication errors occurred. For the dashboard, 25% of ELOC are **refactored** by AI, they are the constructors of the widgets which nicely place the widgets on screen. The ANTLR CMake was provided by ChatGPT. In total, of around 6000 ELOC, around 10 to 15% has been done or refactored by AI, this is mostly composed of the server and dashboard/trigger code. Comments were generated by ChatGPT and reviewed so they would reflect actual code.

In Erlang, it was used to provide documentation and help with TCP communication exceptions. To give an estimate, around 20% of 350 ELOC are done or refactored by AI and they mostly relate to TCP communication and errors handling. Comments were generated and restructured to present the tools nicely.

The **written master thesis** was written **entirely** without the aid of AI.

Contents

1 Introduction **1**
1.1 Problem . . . 1
1.2 Approach . . . 1
1.3 Objective . . . 2
1.4 Previous work . . . 2
1.5 Contributions . . . 3
1.6 Roadmap . . . 3

2 Background **5**
2.1 An overview of ΔQSD . . . 5
2.1.1 Outcome . . . 5
2.1.2 Failure semantics . . . 6
2.1.3 Quality attenuation (ΔQ) . . . 6
2.1.4 Partial ordering . . . 7
2.1.5 Timeliness . . . 7
2.1.6 QTA, required ΔQ . . . 7
2.1.7 Outcome diagram . . . 8
2.1.8 Outcome diagrams refinement . . . 10
2.1.9 Independence hypothesis . . . 11
2.2 Observability . . . 12
2.2.1 erlang:trace . . . 13
2.2.2 OpenTelemetry . . . 13
2.3 Current observability problems . . . 16
2.3.1 Handling of long spans . . . 16

3 Design **17**
3.1 Measurement concepts . . . 18
3.1.1 Probes . . . 18
3.1.2 Extending failure . . . 18
3.1.3 Time series of outcome instances . . . 19
3.2 Application side . . . 20
3.2.1 System under test . . . 20
3.2.2 ΔQ Adapter . . . 20
3.2.3 Inserting probes in Erlang - From spans to outcome instances . . . 20
3.3 Oscilloscope side . . . 21

3.3.1 Server 21
3.3.2 ΔQ Oscilloscope 21
3.3.3 Inserting probes in the oscilloscope 22
3.4 Sliding execution windows 23
3.4.1 Sampling window 23
3.4.2 Polling window (Observing multiple ΔQs over a time interval) . 24
3.5 Triggers 24
3.5.1 Snapshot 24

4 Oscilloscope: User level concepts **26**
4.1 ΔQSD concepts 26
4.1.1 Representation of a ΔQ 26
4.1.2 dMax = Δt · N 27
4.1.3 QTA 29
4.1.4 Confidence bounds 29
4.2 ΔQ display 30
4.3 Outcome diagram 31
4.3.1 Causal link 31
4.3.2 Sub-outcome diagrams 31
4.3.3 Outcomes 32
4.3.4 Operators 32
4.3.5 Limitations 32
4.3.6 Outcome diagram example 33
4.4 Dashboard 33
4.4.1 Sidebar 33
4.4.2 Plots window 35
4.5 Triggers 35
4.5.1 Load 35
4.5.2 QTA 35

5 Oscilloscope: implementation **36**
5.1 ΔQSD implementation 36
5.1.1 Histogram representation of ΔQ 36
5.1.2 dMax 37
5.1.3 Rebinning 37
5.1.4 Convolution 38
5.1.5 Arithmetical operations 39
5.1.6 Confidence bounds 39
5.2 Adapter 40
5.2.1 API 40
5.2.2 Handling outcome instances 42
5.2.3 TCP connection 42
5.3 Parser 43
5.3.1 ANTLR 43
5.3.2 Grammar 44
5.4 Oscilloscope GUI 44

6 Evaluation on synthetic programs **45**
6.1 System with sequential composition . . . 45
6.1.1 System's outcome diagram . . . 46
6.1.2 Determining parameters dynamically . . . 47
6.1.3 Observing non-linearity in the oscilloscope . . . 47
6.1.4 Detecting non-linearity with triggers . . . 51
6.2 Detecting slower workers in operators . . . 52
6.2.1 First to finish application . . . 52
6.2.2 All to finish application . . . 54

7 Performance study **56**
7.1 Convolution performance . . . 56
7.2 GUI plotting performance . . . 57
7.3 ΔQ adapter performance . . . 58

8 Conclusions and future work **60**
8.1 Future improvements . . . 61
8.1.1 Oscilloscope improvements . . . 61
8.1.2 OpenTelemetry improvements . . . 62
8.1.3 Adapter improvement . . . 62
8.1.4 Real applications . . . 62
8.1.5 Licensing limitations . . . 62

Bibliography **63**

A Tests specifications **67**

I Addendum to Chapters 4 & 5 **68**

B Oscilloscope screenshots **69**
B.1 System/Handle plots tab . . . 69
B.2 Parameters tab . . . 70
B.3 Triggers tab . . . 71
B.4 Snapshot window . . . 72
B.5 Connection controls . . . 73
B.6 Widget view of GUI . . . 74

II User manual **75**

C How to download and launch **76**

D Oscilloscope: How to use **78**
D.1 Establishing the adapter - oscilloscope connection . . . 78
D.2 Sidebar: Outcome diagram and plots . . . 79
D.2.1 Creating the system/outcome diagrams . . . 79
D.2.2 Saving the system definition . . . 81

D.2.3 Loading the system definition . . . 81
D.2.4 Changing the sampling rate . . . 82
D.2.5 Managing the plots . . . 82
D.2.6 Removing a plot . . . 85
D.3 Sidebar: Probes settings . . . 85
D.3.1 Setting a QTA . . . 85
D.3.2 Setting the parameters of a probe . . . 86
D.4 Triggers . . . 88
D.5 Snapshots . . . 88
D.6 Instrumenting the Erlang application . . . 89
D.6.1 Including the adapter . . . 89
D.6.2 Starting spans . . . 90
D.6.3 With spans . . . 90
D.6.4 Ending spans . . . 90
D.6.5 Failing outcome instances . . . 90

III Source Code Appendix 91

E Grammar 92

F C++ Source Files 93
F.1 Root folder . . . 93
F.1.1 main.cpp . . . 93
F.1.2 Application.cpp . . . 94
F.2 Dashboard . . . 99
F.2.1 ColorRegistry.cpp . . . 99
F.2.2 CustomLegendEntry.cpp . . . 99
F.2.3 CustomLegendPanel.cpp . . . 100
F.2.4 DQPlotController.cpp . . . 101
F.2.5 DQPlotList.cpp . . . 111
F.2.6 DelaySettingsWidget.cpp . . . 113
F.2.7 DeltaQPlot.cpp . . . 115
F.2.8 MainWindow.cpp . . . 118
F.2.9 NewPlotList.cpp . . . 123
F.2.10 ObservableSettings.cpp . . . 125
F.2.11 SamplingRateWidget.cpp . . . 125
F.2.12 QTAInputWidget.cpp . . . 126
F.2.13 Sidebar.cpp . . . 129
F.2.14 SnapshotViewerWindow.cpp . . . 131
F.2.15 StubControlWidget.cpp . . . 136
F.2.16 SystemCreationWidget.cpp . . . 138
F.2.17 TriggersTab.cpp . . . 141
F.3 Outcome diagram . . . 146
F.3.1 Observable.cpp . . . 146
F.3.2 Operator.cpp . . . 149
F.3.3 Outcome.cpp . . . 151

F.3.4 Probe.cpp 152
F.3.5 System.cpp 153
F.4 ΔQ (maths) 157
F.4.1 ConfidenceInterval.cpp 157
F.4.2 DeltaQ.cpp 159
F.4.3 DeltaQOperations.cpp 165
F.4.4 Snapshot.cpp 170
F.4.5 TriggerManager.cpp 172
F.4.6 Triggers.cpp 174
F.5 parser 175
F.5.1 SystemBuilder.cpp 175
F.5.2 SystemParserInterface.cpp 180
F.6 server 182
F.6.1 Server.cpp 182

G C++ Header Files 191
G.1 Root 191
G.1.1 Application.h 191
G.2 dashboard 192
G.2.1 ColorRegistry.h 192
G.2.2 CustomLegendEntry.h 192
G.2.3 CustomLegendPanel.h 193
G.2.4 DQPlotController.h 194
G.2.5 DQPlotList.h 196
G.2.6 DelaySettingsWidget.h 198
G.2.7 DeltaQPlot.h 199
G.2.8 MainWindow.h 202
G.2.9 NewPlotList.h 203
G.2.10 ObservableSettings.h 204
G.2.11 SamplingRateWidget.h 205
G.2.12 QTAInputWidget.h 206
G.2.13 Sidebar.h 208
G.2.14 SnapshotViewerWindow.h 210
G.2.15 StubControlWidget.h 211
G.2.16 SystemCreationWidget.h 212
G.2.17 TriggersTab.h 213
G.3 diagram 215
G.3.1 Observable.h 215
G.3.2 Operator.h 218
G.3.3 OperatorType.h 220
G.3.4 Outcome.h 220
G.3.5 Probe.h 220
G.3.6 Sample.h 222
G.3.7 System.h 222
G.4 ΔQ (maths) 224
G.4.1 ConfidenceInterval.h 224

G.4.2 DeltaQ.h . . . 225
G.4.3 DeltaQOperations.h . . . 227
G.4.4 DeltaQRepr.h . . . 228
G.4.5 QTA.h . . . 229
G.4.6 Snapshot.h . . . 230
G.4.7 TriggerManager.h . . . 232
G.4.8 TriggerTypes.h . . . 234
G.4.9 Triggers.h . . . 234
G.5 parser . . . 235
G.5.1 SystemBuilder.h . . . 235
G.5.2 SystemErrorListener.h . . . 236
G.5.3 SystemParserInterface.h . . . 237
G.6 server . . . 238
G.6.1 Server.h . . . 238

H Build Configuration Files **241**
H.1 src/CMakeLists.txt . . . 241
H.2 dashboard/CMakeLists.txt . . . 241
H.3 diagram/CMakeLists.txt . . . 242
H.4 maths/CMakeLists.txt . . . 243
H.5 parser/CMakeLists.txt . . . 244
H.6 server/CMakeLists.txt . . . 245

I Erlang Source Files **247**
I.1 Root . . . 247
I.1.1 dqsd_otel.erl . . . 247
I.1.2 dqsd_otel_app.erl . . . 251
I.1.3 dqsd_otel_sup.erl . . . 252
I.1.4 dqsd_otel_tcp_client.erl . . . 252
I.1.5 dqsd_otel_tcp_server.erl . . . 254

J Erlang Application Files **258**
J.1 Root . . . 258
J.1.1 dqsd_otel.app.src . . . 258

Chapter 1

Introduction

1.1 Problem

Diagnosing and measurement of distributed systems is not a trivial task. The very nature of them makes capturing the behaviour of the different parts that compose it, that may be in different physical locations, a challenging endeavour. [1]

In 2019, the merge of OpenCensus and OpenTracing led to the creation of OpenTelemetry [2]. It aims to provide a standard set of API to monitor running systems and avoid vendor lock-in [3]. As a result, multiple vendors' monitoring tools support OpenTelemetry metrics analysis [4]. They allow engineers to follow requests in system, studying their time of execution, the paths they take, the various events and much more [5] [6]. Nevertheless, open source non-commercial monitoring tools of large distributed system are few, and current solutions which may provide detailed insights about a system are commercial ones. [7]

Furthermore, while the tools may also provide insights about running systems and the execution times of various endpoints [8], they fail to capture essential requirements about timeliness. How can performance issues be recognised instantly? These issues may not be even apparent in these monitoring tools, as they can appear at the microseconds level. They may not be easily spotted by normal analysis of average delays.
This is where the ΔQSD paradigm comes in.

1.2 Approach

In the context of this thesis, the **ΔQSD paradigm** has been used to develop a tool to study the real-time behaviour of running systems.
As described by a tutorial given on ΔQSD [9]:

> "ΔQSD is an industrial-strength approach for large-scale system design that can predict performance and feasibility early on in the design process."

The paradigm has been developed over 30 years by the people around **Predictable Network Solutions Ltd** [10][11]. It has had various successful uses in the context of distributed and large-scale projects. Moreover, it is the basis of Broadband forum's TR452 standard series, used in instrumenting data networks [12].

Thanks to outcome diagrams and statistical representations of component's behaviour, performance and feasibility can be predicted with the paradigm at high load, even if the system is not fully defined. [9] [11]

While the paradigm has been successfully applied in **a posteriori** analysis, there is no way yet to analyse a distributed system which is running in real time with ΔQSD. This is where the **ΔQ oscilloscope** comes in.

1.3 Objective

This project will develop a practical tool, the **ΔQ oscilloscope**, for the Erlang developer community.

The Erlang language and Erlang/OTP platform are widely used to develop distributed applications that must perform reliably under high load [13] [14]. The tool will provide useful information for these applications for understanding their behaviour, for diagnosing performance issues, and for optimising performance over their lifetime. [15]

The ΔQ Oscilloscope will perform statistical computations to show real time graphs about the performance of system components. With the oscilloscope prototype we present in this thesis, we are aiming to show that the ΔQSD paradigm is not only a theoretical paradigm, but it can be employed in a real-time tool to diagnose distributed systems. Its application can then be further extended to large systems once the oscilloscope is refined.

The paradigm ideal target is "large distributed applications handling many independent tasks where performance and reliability are important" [9]. This also applies to the oscilloscope.

1.4 Previous work

The ΔQSD paradigm has been formalised across different papers [11] [16] and was brought to the attention of engineers via tutorials [9], and to students at Université Catholique de Louvain. [17]

A Jupyter notebook workbench has been made available on GitHub [18]. It is meant as an interactive tool to show how the ΔQSD paradigm can be applied to real life examples. It shows real time ΔQ graphs for typical outcome diagrams, but is not adequate to be scaled to real time systems.

Observability tools such as Erlang tracing [19] and OpenTelemetry [20] lack the notions of failure as defined in ΔQSD, which allows detecting performance problems early on. We base our program on OpenTelemetry to incorporate already existing notions of causality and observability to augment their capabilities and make them suitable to work with the ΔQSD paradigm.

1.5 Contributions

We make the following principal contributions in the master thesis:

- The ΔQ oscilloscope, from design to implementation, to plot real-time ΔQ graphs. The oscilloscope contributions include:
  - A graphical interface in Qt.
  - The underlying implementation of ΔQSD concepts, including a textual syntax to create oucome diagrams derived from the original algebraic syntax.
  - Efficient convolution algorithms.
  - A system of triggers to catch rare events, when system behaviour fails to meet quality requirements.
- The ΔQ adapter to communicate from the Erlang application to the oscilloscope.
- The evaluation of the effectiveness of the oscilloscope on synthetic applications.
- The evaluation of the efficiency of the basic operations regarding the oscilloscope, convolution, graphing and the adapter overhead.

These contributions can show that the ΔQSD paradigm can be translated from a posteriori analysis to real-time observation of running system. Furthermore, it reinforces the validity of the paradigm.

1.6 Roadmap

This thesis gives the reader everything that is needed to use the oscilloscope and exploit it to its full potential.

We divided the thesis in multiple chapters:

- Chapter 2 gives the reader a background of the theoretical foundations of ΔQSD, which are the basis of the oscilloscope and are fundamental to understand what is shown in the oscilloscope. Secondly, an introduction to OpenTelemetry, the framework our Erlang adapter is built on top of. Lastly, we provide what we believe are the current limitations of the observability framework and how we plan to tackle them.
- Chapter 3 first provides the "measurement concepts". These concepts serve as an introduction to understand the following chapters and as a bridge from OpenTelemetry to the oscilloscope. We then delve into how the different parts of our design interact together and how to correctly apply the concepts we introduced. The parts are divided in two sides. First, we present the application side, where the Erlang system to be observed is and where the ΔQ adapter will be. Second, the oscilloscope side, where the ΔQ oscilloscope will display graphs about the running system. Lastly, after having introduced the oscilloscope, we explain abstract concepts implemented in it, like sliding windows and triggers.

- Chapter 4 & 5 present the oscilloscope. First providing "user level concepts" of how ΔQSD is used and what the user should expect visually from the dashboard. Chapter 4 also provides a complete explanation of how to write outcome diagrams and what the different sections on the dashboard do. Secondly, in Chapter 5, we give a more low level explanation, which goes into more technical details of the parts that compose the oscilloscope. Namely, we provide mathematical explanations of ΔQSD concepts explained in the previous chapter.

- Chapter 6 provides synthetic applications which have been tested with the oscilloscope that demonstrate the usefulness of the oscilloscope in a distributed setting. In Chapter 7 we perform evaluations of the performance of the different parts we have developed to understand the overheads that are present.

Chapter 8 provides future possibilities which can be explored to improve the application. In the appendix, we provide screenshots of the application. We also provide a user manual to help users use the oscilloscope, along with C++ and Erlang source code of the oscilloscope and the adapter.
The oscilloscope (`https://github.com/fnieri/DeltaQOscilloscope`) and adapter (`https://github.com/fnieri/dqsd_otel`) can be found on GitHub as open source projects.

Chapter 2

Background

This chapter aims to provide firstly a background of the concepts key to understanding the ΔQSD paradigm.

Secondly, we explain the observability solutions that have been explored for the oscilloscope, delving deeper into OpenTelemetry and its macros.

We finish by explaining what we believe are the current limitations of OpenTelemetry and explaining where the paradigm and the oscilloscope comes in.

2.1 An overview of ΔQSD

ΔQSD is defined as [11]:

> "A metrics-based, quality-centric paradigm that uses formalised outcome diagrams to explore the performance consequences of design decisions".

The key concepts that give the paradigm its name are **quality attenuation (ΔQ)** and **outcome diagrams**. [9]

The dependency and causality properties of a system can be captured by outcome diagrams, while the probability distribution representation (ΔQ) can precisely model a system's behaviour. [11]

The following sections are a summary of multiple articles and presentations formalising the paradigm. Some of the graphs we present have too been adapted from the articles.

2.1.1 Outcome

To build outcome diagrams, we need to first introduce outcomes.

Outcomes represent system behaviours or tasks that "can start at some point in time and ***may*** be observed to complete at some later time" [12]. The result produced by performing a system's task is mapped to an outcome. [11]

The particularity of outcomes is that they can represent multiple levels of granularity. Suppose an outcome is beyond the current system's control (e.g. a database/cloud

request), is non-atomic (can be broken down in multiple sub-outcomes). These outcomes can be represented as black boxes: you can observe their start and end, but do not know what is being executed. As the system gets refined, these outcomes can then be refined to model a single outcome or multiple outcomes, if needed. [11]
Even though these outcomes are defined as "black boxes", they still have timeliness constraints like any other outcome.

Let us define some important concepts about outcomes.

Observables Key to outcomes is the notion of event.
An outcome has observable starting events end ending events [9]. As the events may occur in different locations in a distributed setting, we say the outcome has a starting set and ending set of events. They are called "**observables**". [11]
We stated previously that an outcome **may** be observed to complete, this is because a starting event has no guarantee that it will be ended. An outcome can be said "done" if an end event occurs for a start event.

Outcome instance An outcome instance is the result of an execution of an outcome given a starting event e_{in} and an end event e_{out}. [16]

Graphical Representation Outcomes are represented as circles, with the starting and terminating set of events being represented by boxes. [11]

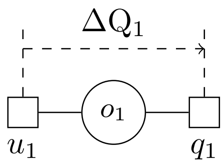

Figure 2.1: The outcome (circle) and the starting set (left) and terminating set (right) of events.

2.1.2 Failure semantics

Failure indicates "an input message m_{in} **that has no output message** m_{out}" [16]. It models the probability that the delay is infinite.

2.1.3 Quality attenuation (ΔQ)

ΔQ is defined as "a cumulative distribution function (CDF) that defines both *latency* and *failure probability* between a start and end event". [9]

As multiple instances of an outcome are executed, multiple delays can be observed for the executions. The observed delays can be represented as a CDF, we call it ΔQ_{obs}. In the ΔQ's CDF, $\mathbf{\Delta Q(x)}$ is the probability that an outcome O occurs in time $t \leq x$ [16]. We may sometimes use the derivative of a ΔQ, which is the probability density function (PDF). We show a typical CDF of a ΔQ in Fig. 2.2.

The key feature that makes ΔQ stands out is the notion of failure incorporated in the representation of an outcome's behaviour.
Ideally, a system would execute without errors, failure or delay. This is never the case. Since the ideal cannot be attained, the quality of the system's responses are "*attenuated to the relative ideal*" [11]. This quality attenuation gives the name to ΔQ.

Since multiple factors can influence the delays of a response (geographical, physical) [9], ΔQ can be firstly modeled as a random variable. Nevertheless, as it incorporates failures, which are discrete variables, and delays, which are continuous random variables, the authors describe it as an *Improper Random Variable*, as the probability of a finite or bounded delay is < 1. [11]

Intangible mass Depicted in Fig. 2.2, the ***intangible mass*** $1 - \lim_{x\to\infty} \Delta Q(x)$ of a ΔQ encodes the probability of failure/timeout/exception occurring. [16]

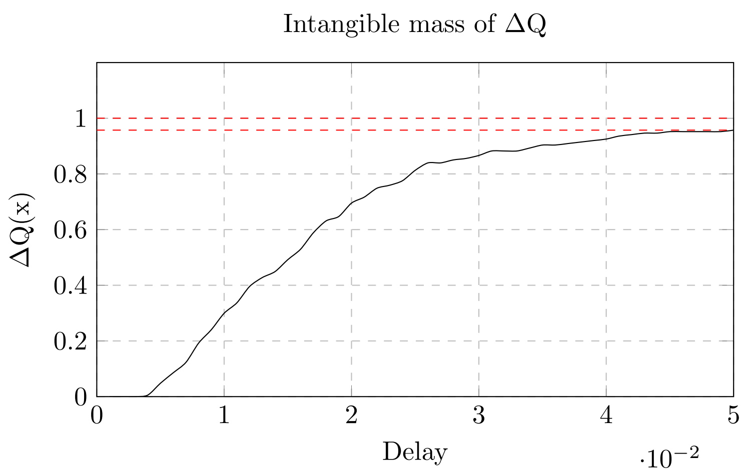

Figure 2.2: ΔQ CDF representation with intangible mass (red, dotted). The failure rate is about 5%

2.1.4 Partial ordering

We say ΔQ_1 is *less than* another ΔQ_2 if its CDF is everywhere to the left and above the other's CDF. This concept is represented mathematically as a partial order. [9]

2.1.5 Timeliness

Timeliness is "delivering results within required time bounds (sufficiently often)". [9]

2.1.6 QTA, required ΔQ

The Quantitative Timeliness Agreement (QTA) specifies the precise timeliness requirements (ΔQ_{req}) of a ΔQ_{obs}. [12] [16]

Leveraging the definition of partial ordering and timeliness, we can say that a system *satisfies timeliness* if $\Delta Q_{obs} \leq \Delta Q_{req}$. [16]

Slack and hazard There is performance *slack* when "a ΔQ is strictly less than the requirement." ($\Delta Q_{obs} < \Delta Q_{req}$).
There is performance *hazard* when "an observed ΔQ_{obs} intersects or is strictly greater than the required ΔQ_{req}" ($\Delta Q_{obs} \not< \Delta Q_{req}$). [11]

QTA example Imagine a system where 25% of the executions should take < 15 ms, 50% < 25 ms and 75% < 35 ms, all queries have a maximum delay of 50ms and 5% of executions can timeout, the QTA can be represented as a step function.
We present in Fig. 2.3 an example of systems showing slack and hazard.

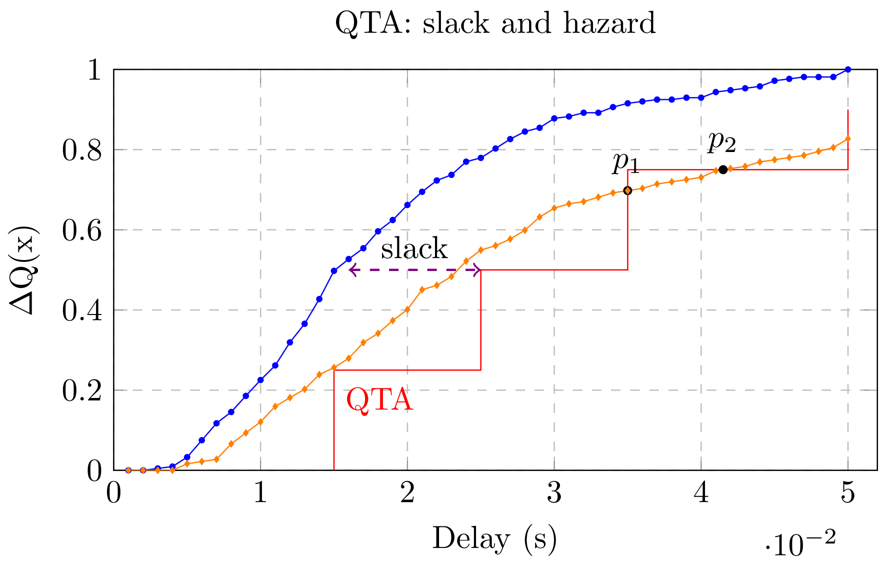

Figure 2.3: The system in blue (circle) is showing slack and satisfies the requirement. The system in orange (diamond) is showing signs that it cannot handle the stress, it is not respecting the system requirements imposed by the QTA. It intersects with the QTA in p_1, p_2, so it is not respecting the timeliness requirements. There is performance hazard.

2.1.7 Outcome diagram

An outcome diagram is central to capture the causal relationships between the outcomes [11]. It shows the causal connections between all the outcomes we are interested in. Given an outcome diagram, one can compute the ΔQ for the whole system.
There are four different operators that represent the relationships between outcomes. [9]

Sequential composition

If we assume two outcomes O_A, O_B where the end event of O_A is the start event of O_B, then we say the two outcomes are sequentially composed. The total delay ΔQ_{AB} is given by the convolution of the PDFs of O_A and O_B ($O_A \circledast O_B$).

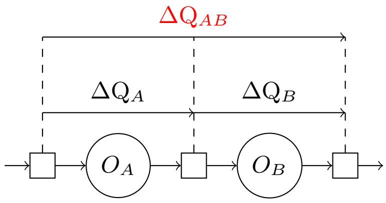

Figure 2.4: Sequential composition of O_A and O_B.

Where convolution ($\circledast$) between two PDF is:

$$PDF_A \circledast PDF_B(t) = PDF_{AB}(t) = \int_0^t PDF_A(\delta) \cdot PDF_B(t-\delta)\; d\delta \tag{2.1}$$

Thus, ΔQ$_{AB}$:

$$\Delta Q_{AB} = \int_0^t PDF_A \circledast PDF_B \; d\delta \tag{2.2}$$

Convolution is the only operation which is based on the PDFs, the following operations are based on the CDF of the ΔQs (hence the use of the ΔQ notation).

First to finish (FTF)

If we assume two independent outcomes O_A, O_B with the same start event, first-to-finish occurs when at least one end event occurs, it can be calculated as:

$$\begin{aligned} (1 - \Delta Q_{FTF(A,B)}) &= Pr[d_A > t \wedge d_B > t] \\ &= Pr[d_A > t] \cdot Pr[d_B > t] = (1 - \Delta Q_A) \cdot (1 - \Delta Q_B) \\ \Delta Q_{FTF(A,B)} &= \Delta Q_A + \Delta Q_B - \Delta Q_A \cdot \Delta Q_B \end{aligned} \tag{2.3}$$

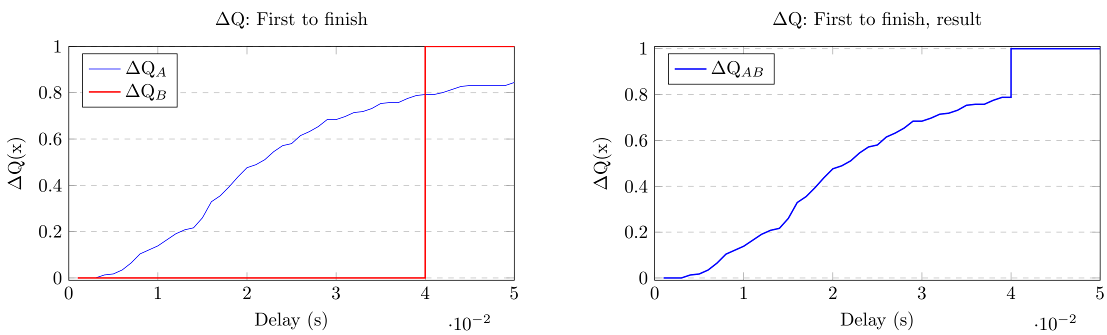

Figure 2.5: Left: ΔQ$_{(A,B)}$. Right: FTF$_{(A,B)}$ = ΔQ$_{AB}$

All to finish (ATF)

If we assume two independent outcomes O_A, O_B with the same start event, all-to-finish occurs when both end events occur, it can be calculated as:

$$\begin{aligned} \Delta Q_{ATF(A,B)} &= Pr[d_A \le t \wedge d_B \le t] \\ &= Pr[d_A \le t] \cdot Pr[d_B \le t] = \Delta Q_A \cdot \Delta Q_B \\ \Delta Q_{ATF(A,B)} &= \Delta Q_A \cdot \Delta Q_B \end{aligned} \tag{2.4}$$

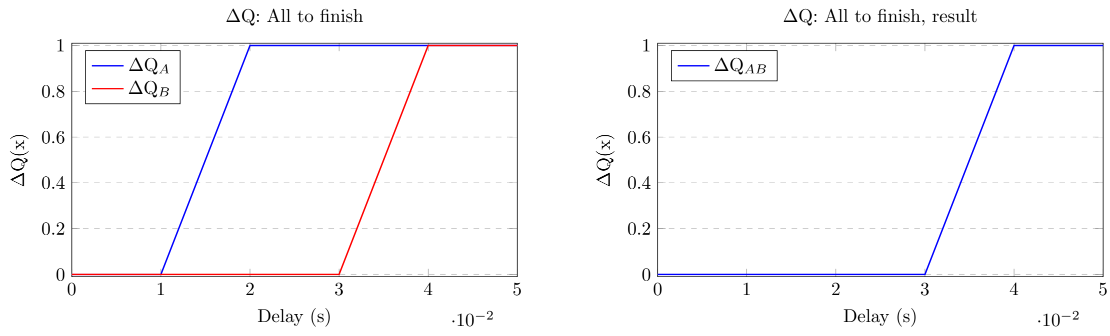

Figure 2.6: Left: $\Delta Q_{(A,B)}$. Right: $\mathrm{ATF}_{(A,B)} = \Delta Q_{AB}$

Probabilistic choice (PC)

If we assume two possible outcomes O_A and O_B and exactly one outcome is chosen during each occurrence of a start event and:

- O_A happens with probability $\dfrac{p}{p+q}$
- O_B happens with probability $\dfrac{q}{p+q}$

$$\Delta Q_{PC}(A,B) = \frac{p}{p+q}\Delta Q_A + \frac{q}{p+q}\Delta Q_B \tag{2.5}$$

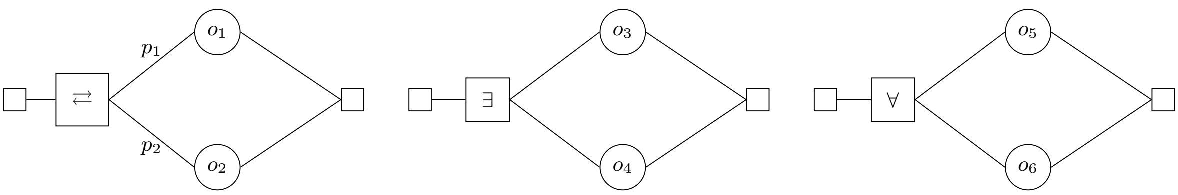

Figure 2.7: Graphical representation of the possible operators in an outcome diagram. From left to right: Probabilistic choice, first-to-finish, all-to-finish. [11]

First-to-finish, All-to-finish and probabilistic-choice are calculated on the CDF of the ΔQs of their components.

These operators can be assembled together to create an outcome diagram. In Chapter 4 we will see how one can go from the graphical representation to outcome diagrams which can be used in the ΔQ oscilloscope.

2.1.8 Outcome diagrams refinement

An important feature of outcome diagrams is the ability to be able to design a system even with *"black boxes"*, before the complete details of it are known. [11]
An outcome diagram can be "unboxed" by refining the outcomes that compose it. We can adapt a situation described by the previously cited article, called "Mind your Outcomes", to understand how refinement can allow the user to have a very precise representation of a system.

We first start with a black box, unnamed outcome with start event A and end event Z, somewhere in the system. The first refinement step would be giving the outcome a name.

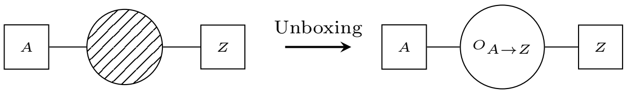

Figure 2.8: Refinement from black box to named outcome.

The system can be further refined by adding outcomes that represent tasks. For example, the engineer might believe that it will take two tasks to get from A to Z. We can then add another outcome, sequentially composed, to represent this situation.

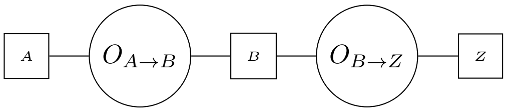

Figure 2.9: Further refinement from one task to two tasks.

We can also model the chance of executing two tasks as a probabilistic choice, where there is p_2 probability that the execution from A to Z will execute two tasks. The outcome diagram can be refined as a probabilistic choice.

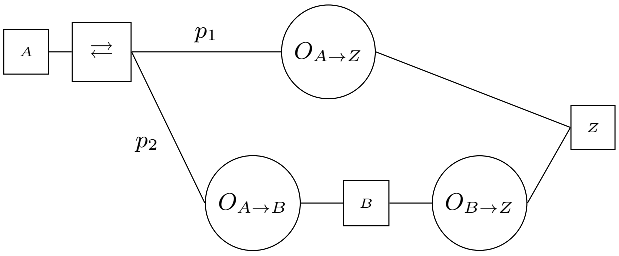

Figure 2.10: Refinement as probabilistic choice of executing either one or two tasks.

In essence, the refinement could model a very fine-grained representation of the system by further refining the system, to represent the possibility of executing n tasks. This demonstrates the power of outcome diagrams to represent system diagrams with high precision. They can help explore design decisions thanks to outcomes and operators.

2.1.9 Independence hypothesis

An important aspect of sequential composition is the assumption of outcomes having independent behaviour [15]. Let us explain the following assumption clearly.

Assume the situation described by Fig. 2.11, two sequentially composed outcomes o_1, o_2 running on the same processor. The overall delay of execution can be observed from the start event of o_1 (u_1) to the end event of o_2 (r_1).

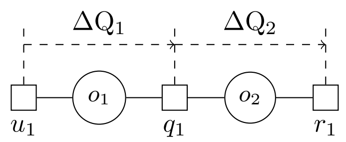

Figure 2.11: Two sequentially composed outcomes, o_1, o_2. The end event of o_1 is the start event of o_2.

At low load, the two components behaviour will be independent (Fig. 2.12), the system will behave **linearly**. According to the superposition principle, the overall delay will be the sum of the two delays, as will the overall processor utilisation. [21]

When load increases, the two components will start to show dependent behaviour due to the processor utilisation increasing. The ΔQ of the observed total delay will then deviate from the sum of the two delays ($o_1 \circledast o_2$). This situation is observed in Fig. 2.12.

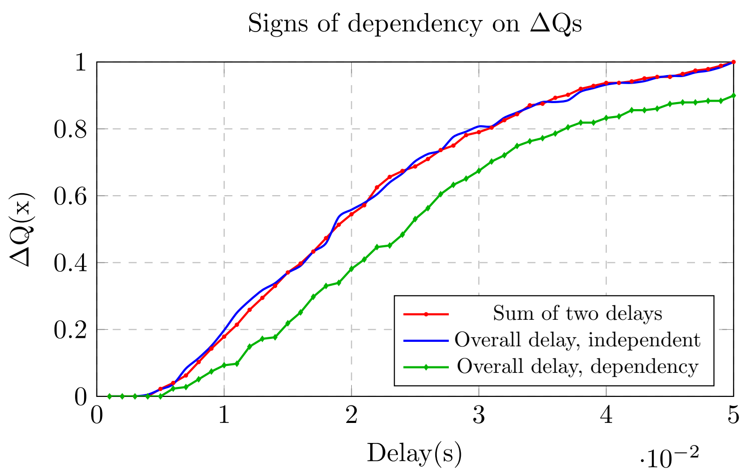

Figure 2.12: When the components are independent, the sum of the two delays (blue) and the overall delay (red) can be superposed. As o_1 and o_2 show signs of dependency, the overall delay (green) can be seen deviating from the sum of the two delays. There is **non-linearity**.

When the system is far from being overloaded, the effect is noticeable. As the cliff edge of overload is approached, the non-linearity will increase [9] [15]. These theoretical results can be observed in the oscilloscope. We will explore such cases in Chapter 6.

2.2 Observability

OpenTelemetry refers to observability as [22]:

> "The ability to understand the internal state by examining its output. In the context of a distributed system, being able to understand the internal state of the system by examining its telemetry data."

In the case of the Erlang programming language, we describe respectively two different ways to observe a running Erlang system: erlang:trace and OpenTelemetry.

2.2.1 erlang:trace

The Erlang programming language gives the users different ways to observe the behaviour of a system, one of those is the interface `erlang:trace`. According to the documentation: "The Erlang run-time system exposes several trace points that can be observed, observing the trace points allows users to be notified when they are triggered" [23]. One can observe function calls, messages being sent and received, process being spawned, garbage collecting and more.

```
-spec trace(PidPortSpec, How, FlagList) -> integer()
   when
       PidPortSpec ::
           pid() |
           port() |
           all | processes | ports | existing | existing_processes |
            ↪  existing_ports | new |
           new_processes | new_ports,
       How :: boolean(),
       FlagList :: [trace_flag()].
```

Figure 2.13: erlang:trace/3 specification. [23]

Nevertheless, in Erlang trace there is no default way to follow a message and get its whole execution trace. This is a missing feature that is crucial for observing a program functioning and being able to connect an application to our oscilloscope. This is where the OpenTelemetry framework comes in.

2.2.2 OpenTelemetry

According to OpenTelemetry's website [22]: OpenTelemetry is an open-source, vendor-agnostic observability framework and toolkit designed to generate, export and collect telemetry data, in particular traces, metrics and logs. OpenTelemetry provides a standard protocol, a single set of API and conventions and lets the user own the generated data, allowing to switch between observability backends freely.

OpenTelemetry is available for a plethora of languages [24], including Erlang. As of writing this, logs and metrics are unstable in Erlang. [25]

The Erlang Ecosystem Foundation has a working group focused on evolving the tools related to observability, including OpenTelemetry and the runtime observability monitoring tools. [26]

Traces

Traces are why we are basing our program on top of OpenTelemetry, traces follow the whole path of a request in an application, and they are comprised of one or more spans [6]. Traces can propagate to multiple services and record multiple paths in different microservices [27].

Span A span is a unit of work or operation. Multiple spans can be assembled into a trace and can be causally linked (Fig. 2.15). The spans can have a hierarchy, where *root spans* represent a request from start to finish and a child span the requests that are completed inside the root span [27]. We will see in later sections how this can relate to what the oscilloscope does.

The notion of spans and traces allows us to follow the execution of a request and carry a context. Spans can be linked to mark causal relationships between multiple spans [6]. This relation can be represented in the oscilloscope via **probes**, we will present how spans relate to probes in following sections.

```json
{
  "name": "oscilloscope-span",
  "context": {
    "trace_id": "5b8aa5a2d2c872e8321cf37308d69df2",
    "span_id": "5fb397be34d26b51"
  },
  "parent_id": "051581bf3cb55c13",
  "start_time": "2022-04-29T18:52:58.114304Z",
  "end_time": "2022-04-29T22:52:58.114561Z",
  "attributes": {
    "http.route": "some_route"
  },
}
```

Figure 2.14: Example of span from the OpenTelemetry website [6]. The span has a parent, indicating that child and parent spans are related and are both part of the same trace.

Monitoring OpenTelemetry spans

In OpenTelemetry, the user can export their traces export to backends and monitoring such as Jaeger (Fig. 2.15), Zipkin, Datadog [28]. There, a user can analyse the traces to troubleshoot their programs by observing the flow of the requests [29]. These monitoring tools give extensive details about a running system, but may fail to capture essential timeliness requirements and performances issues early enough.

Our oscilloscope is a kind of monitoring tool, one that gives precise statistical insights about a running system. It is clear that the oscilloscope does not have the same capabilities as Datadog [30] might have, where you can observe cloud instances, instances cost, dependency graphs. But the oscilloscope can nevertheless provide precise insights about dependency, overload and much more, thanks to the ΔQSD paradigm.

This is also the reason why the adapter includes OpenTelemetry macros. The oscilloscope can be put next to a monitoring tool where one exports spans to, so that an engineer might consult the monitoring tool to get the global picture of a running app, and the oscilloscope to provide precise insights to understand the system's behaviour.

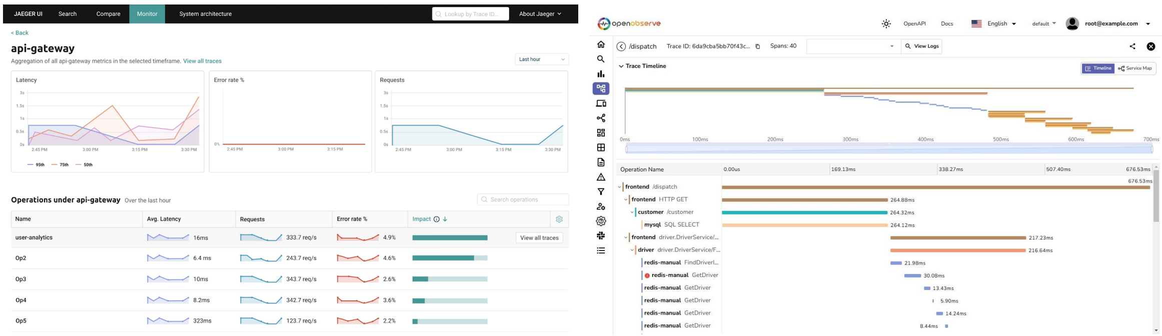

Figure 2.15: **Left**: Jaeger interface [31]. **Right**: Analysis of a span on OpenObserve. [32]

Span macros

OpenTelemetry provides macros to start, end and interact with spans in Erlang. The following code excerpts are taken from the OpenTelemetry instrumentation wiki. [25]

?with_span ?with_span creates active spans. An active span is the span that is currently set in the execution context and is considered the "current" span for the ongoing operation or thread. [33]

```
parent_function() ->
    ?with_span(parent, #{}, fun child_function/0).
child_function() ->
%% this is the same process, the span parent set as the active
%% span in the with_span call above will be the active span in this
↪ function
    ?with_span(child, #{},
               fun() ->
     %% when this function returns, child will complete.
               end).
```

?start_span ?start_span creates a span which isn't connected to a particular process, it does not set the span as the current active span.

```
SpanCtx = ?start_span(child),
Ctx = otel_ctx:get_current(),
proc_lib:spawn_link(fun() ->
                        otel_ctx:attach(Ctx),
                        ?set_current_span(SpanCtx),
                        ?end_span(SpanCtx)
                    end),
```

?end_span `?end_span` ends a span started with `?start_span`.

2.3 Current observability problems

The problem we are trying to tackle can be described by the following situation: imagine an Erlang application instrumented with OpenTelemetry. Suddenly, the application starts slowing down, and the execution of a function takes 10 seconds. The engineer knows it should take at most 1 second. In the 10 seconds, no information about the span appears on the dashboard.

We believe that this is a problem. One would like to know right away if something is wrong with their application. This is where the ΔQSD paradigm and the ΔQ oscilloscope come in handy.
Using ΔQSD, we can set a maximum delay ($dMax$) for a task. As soon as the maximum delay is hit, the oscilloscope is notified right away that there is a problem.

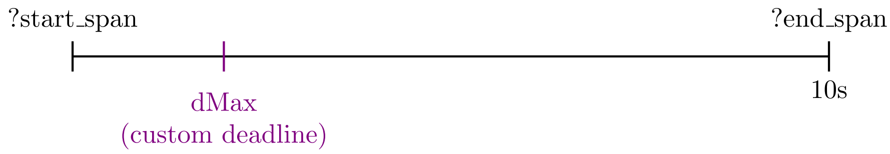

Figure 2.16: Execution of a span in OpenTelemetry. Normally, the user will be notified after 10 seconds that the function has ended. The $dMax$ deadline gives an early notification that the span has taken too long.

2.3.1 Handling of long spans

OpenTelemetry presents a bigger problem, what happens when there are long-running spans? Even worse, what happens when spans are not actually terminated?

An article has already tackled this problem [34]. In the article it is stated that OpenTelemetry limits the length of its spans, moreover, those which are not terminated are lost and not exported.
In addition, it is said that if the span is the parent/root span, its effect could trickle down to child spans. We can quickly see how this can become problematic, all the information about an execution of your task is lost. A span can apparently not be terminated for trivial reasons: refreshing a tab, network failures, crashes. The author states that there are a few solutions that can be implemented: having shorter spans, carrying data in child spans, saving spans in a log to track spans which were not ended to manually set an end time. The solutions have been implemented by the Embrace (commercial) monitoring tool [35], but it is apparently the only tool out of the ecosystem of monitoring tools.

We believe that the adapter we provide can be a great start to improve observability requirements surrounding OpenTelemetry. Data about spans will always be carried to the oscilloscope, whether the span is long or non-terminated. This is further developed in Chapters 3 & 5.

Chapter 3

Design

This chapter aims to first extend the concepts of ΔQSD, giving more insights into how the different parts of the system need to be instrumented to correctly work together.

- We first provide concepts of probes, we extend the ΔQSD notion of failure and describe how time series will work in our oscilloscope, this part is crucial to understand how the measurements are done in real time.
- We then explain the global design of the system (see Fig. 3.1) in two parts:
  Firstly, we explain the application side, where the Erlang running system is. Consequently, it's where the ΔQ adapter interface will be. It performs the translation of spans to outcome instances thanks to the inserted probes.
  Secondly, the oscilloscope side. There, the server receives information about outcome instances from the adapter. The ΔQ oscilloscope can then plot ΔQ graphs from the received instances. In the oscilloscope one can define outcome diagrams, set parameters for probes and control the adapter.
- Lastly, we provide high level concepts of execution windows, triggers and snapshot. These are the key design elements of the oscilloscope.

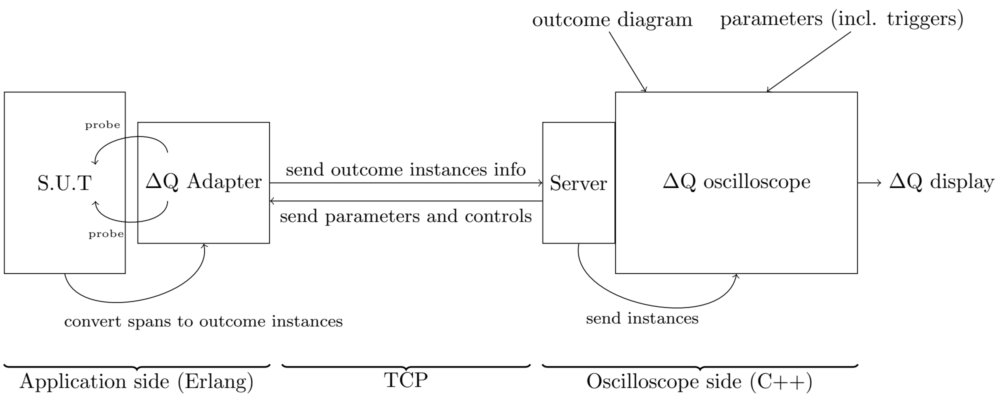

Figure 3.1: Global system design diagram. The two sides communicate via TCP sockets to share information about outcome instances and probe parameters.

3.1 Measurement concepts

3.1.1 Probes

A system instrumented with OpenTelemetry has spans and traces to observe the execution of an operation [6]. The same level of observability must be assured in the oscilloscope, this is why we provide the concept of probes, which, like spans, follow an execution from start to end. **Note** that a definition of probes has already been introduced in a previous article related to ΔQSD [12], but the concept we present here is not the same.

To observe a system, we must put probes in it. For each outcome of interest, a probe (observation point) is attached to measure the delay of the outcome, like one would in a true oscilloscope [15].

Consider Fig. 3.2 below. A probe is attached at every component (for example, a database [9]) to measure their ΔQs (p_2, p_3). Another probe (p_1) is inserted at the beginning and end of the system to measure the global execution delay. Thanks to this probe, the user can observe the ΔQ *"observed at p_1"*, which is the ΔQ which was calculated from the data received by inserting probe p_1. The ΔQ *"calculated at p_1"* is the resulting ΔQ from the convolution of the observed ΔQs at c_2 and c_3.
Probe p_1 is the equivalent of a "root/parent span" which observe the whole execution of c_1, c_2. p_2 and p_3 are child spans which represent single tasks.

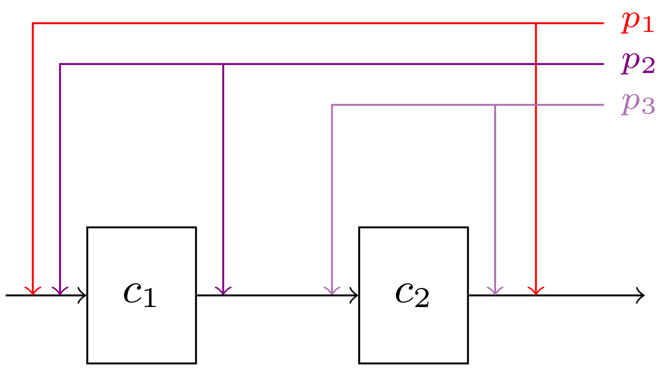

Figure 3.2: Probes inserted in a component diagram. In an application instrumented with OpenTelemetry, p_1 could be considered the root span, c_1 and c_2 its children spans sharing a causal link.

3.1.2 Extending failure

Recall the definition of failure (Section 2.1.2): *"an input message m_{in} that has no output message m_{out}"*. We also introduced the notion of a maximum delay (Fig. 2.16). The notion of failure is extended to the following definition:

"An input message m_{in} that has no output message m_{out} after $dMax$"

By extending the notion of failure to include $dMax$, we can know right away when execution is straying away from engineer defined behaviour, avoiding having to wait until the execution is done. In ΔQSD, an execution may as well take 10 or 15 seconds, but if the delay of execution is $> dMax$, we consider that **failed** right away, we do not need to know the total execution time, the execution has already taken too much [11]. The adapter does not interfere with OpenTelemetry, the full span will be exported regardless to monitoring tools which were set up by the user.

The user can observe both real time information with ΔQSD notion of failure on the ΔQ oscilloscope, and observe those spans in their monitoring tools if they wish.

3.1.3 Time series of outcome instances

Consider a probe p with two distinct sets of events, the starting set of events s and ending set of event e. The outcome instance of a message $m_s \rightarrow m_e$ contains:

- The probe's p name.
- The start time t_s.
- The end time t_e.
- Its status.
- Its elapsed time of execution.

The instance has three possible statuses: `success, timeout, failure`. It can thus be broken down in three possible representations, based on its status:

- (t_s,t_e): This representation indicates that the execution was successful (t $<$ $dMax$).
- $(t_s,\mathcal{T})$: This representation indicates that the execution has timed out (t $>$ $dMax$). The end time is equal to $t_s +$ timeout.
- $(t_s,\mathcal{F})$: This representation indicates the execution has failed given a user defined requirement (i.e. a dropped message given buffer overload in a queue system). It must not be confused with a program failure (crash), if a program crashes during the execution of event e, it will time out since the adapter will not receive an end message.

The **time series** of a probe is the sequence of n outcome instances. The collected elapsed times of execution (delays) from the outcome instances can be represented as a CDF, which is a ΔQ.

What can be considered a failed execution? The choice of what is considered a failed execution is left up to the user who is handling the spans and is program-dependent. Exceptions or errors can be kinds of failure.

Imagine a queue with a buffer: the buffer queue being full and dropping incoming messages can be modeled as a failure.

On another note, the way of handling "errored" spans in OpenTelemetry can differ from user to user [36], so the adapter will not handle ending and setting statuses for "failed" spans.

In any case, **timed out and failed will both be considered as a failure** in a ΔQ. The distinction in an outcome instance is there for future refinements of the oscilloscope, where more statistics can be displayed about a ΔQ.

3.2 Application side

3.2.1 System under test

The system under test **(S.U.T)** is the Erlang system the engineer wishes to observe (Fig. 3.1). It ideally is a system which already is instrumented with OpenTelemetry. The ideal system where ΔQSD is more useful is a system that executes many independent instances of the same action [9].

3.2.2 ΔQ Adapter

The ΔQ adapter is the `dqsd_otel` Erlang interface [37]. It starts and ends OpenTelemetry spans and translates them to outcome instances which are useful for the oscilloscope. This can be done thanks to probes being attached to the system under test, like an oscilloscope would. The outcome instances end normally like OpenTelemetry spans or, additionally, can timeout after a custom timeout ($dMax$), and fail, *according to user's definition of failure.*

Handling of OpenTelemetry spans which goes beyond starting and ending them is delegated to the user, who may wish to do further operations with their spans. The adapter is called from the system under test and communicates outcome instances data to the oscilloscope via TCP sockets.

The adapter can receive messages from the oscilloscope, the messages are about updating probe's $dMax$ or starting and stopping the sending of data to the oscilloscope.

3.2.3 Inserting probes in Erlang - From spans to outcome instances

OpenTelemetry spans are useful to carry context, attributes and baggage in a program [6]. The plethora of attributes they have is nevertheless too much for the oscilloscope. To get the equivalent of spans for the oscilloscope, the adapter needs to be called at the starting events of a probe to start an instance of a probe, and at the ending events to end the outcome instance. The name given with `"start_span/with_span"` is the name of the probe. The PID which is returned by starting a span must be carried throughout the whole execution, and used when ending spans to create the correlation between a probe and an outcome instance.

```
% Start the outcome instance of probe. The call to dqsd_otel starts an
↪  OpenTelemetry span, as it contains a call to ?start_span(Name)
{ProbeCtx, ProbePid} = dqsd_otel:start_span(<<"probe">>),

% Start and fail span directly
{WorkerCtx, WorkerPid} = dqsd_otel:start_span(<<"worker_1">>),
dqsd_otel:fail_span(WorkerPid),
%Here, you would need to end the span manually with ?end_span

%Example of with_span, the call to OpenTelemetry ?with_span is inside
↪  the adapter function, the function fun() -> ok end is executed
↪  inside dqsd_otel.
dqsd_otel:with_span(<<"worker_2">>, fun() -> ok end),
%End the outcome instance of probe. This ends the OpenTelemetry span
↪  aswell. If the outcome instance has already timed out (the time
↪  from start_span to end_span > dMax), the oscilloscope receives no
↪  message where the status is successful. Otherwise, this sends a
↪  message with startTime, endTime, the name "probe" and success
↪  status.
dqsd_otel:end_span(ProbeCtx, ProbePid),
```

Figure 3.3: Example usage of the adapter

Further details about the implementation of the adapter are explained in Chapter 5. A user guide on how to include the adapter in a project and how to instrument a program is found in the appendix (Section D.6).

3.3 Oscilloscope side

3.3.1 Server

The server is a simple TCP server in C++. It is responsible for receiving the messages containing the outcome instances from the adapter. The server forwards the instances to the oscilloscope.

3.3.2 ΔQ Oscilloscope

The oscilloscope is a C++ graphical application which implements a dashboard to observe ΔQs of probes inserted in the system under test [38]. It receives the instances corresponding to probes from the server and adds them to the time series of the probes whose instance is being received. The oscilloscope has a graphical interface which allows the user to create an outcome diagram of the system under test, display real time graphs which show detail about the execution of the system, and allows the user to set parameters for probes. It can also display snapshots of the system as if it was frozen in time.

3.3.3 Inserting probes in the oscilloscope

Probes are automatically inserted in the oscilloscope when creating outcome diagrams. They are inserted on the outcomes observables, operators observables and to the sub-outcome diagrams observables (probes that observe the causal links of multiple outcomes/operators), we will see later on how they can be defined and how an outcome diagram can be created.
The names that are given to outcomes, operators and sub-outcome diagrams are the names of the probes that observe them. Giving these probes a name allows the oscilloscope to match the outcome instances received by the adapter to the probes' time series.

In the system below (Fig. 3.4), probes are automatically attached to outcomes o_1, o_2. The user who wants to observe the result of the sequential composition can insert probes at the start and end of the routine.

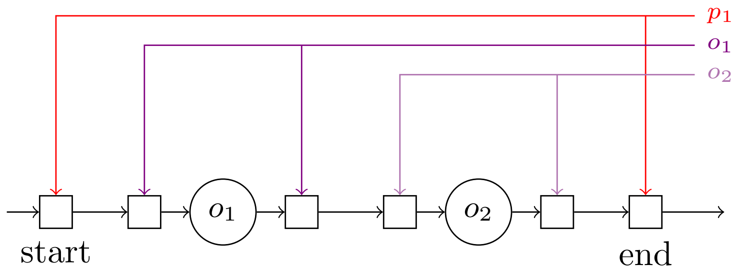

Figure 3.4: Probes inserted in the outcome diagram of the previous component diagram in Fig. 3.2.

The **observables** are an abstract representation of events. Consider the previous code snippet Fig. 3.3: the *start* event of "probe" and worker$_1$'s start event are subsequent instructions. The probe's start event is practically the same as worker$_1$'s start event, indeed, they could be overlapped in the graph above (Fig. 3.4). We nevertheless show the distinction to show that probe and worker$_1$ need to be started differently in Erlang as the information they carry is about two distinct instances. It is the same concept as starting a parent/child span. Furthermore, this difference is remarked in the definition of outcome diagrams for the oscilloscope, for which we provide a syntax in the Chapter 4.

As for operators, probes are automatically attached to the components inside them and to the operators' observables (Section 3.3.3).

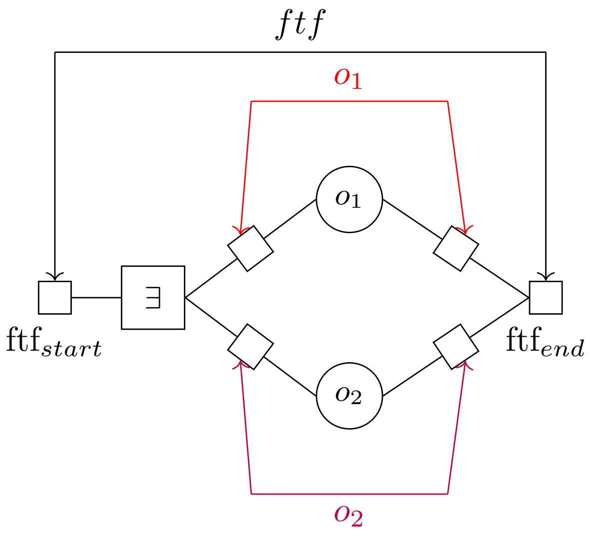

Figure 3.5: Probes inserted into an operator.

The **observed ΔQ** for the first-to-finish operator is the ΔQ for the observables (**start**, **end**). The **calculated ΔQ** is the ΔQ which is the result of the first-to-finish operator being applied on o_1, o_2.

3.4 Sliding execution windows

There are two important windows that we consider in our oscilloscope, the *sampling window* and the *polling window*.

3.4.1 Sampling window

Suppose we are at time t, the observed ΔQ of a sampling window at time t we display is the ΔQ obtained from the outcome instances which ended within a **window of time** $(t-1)_l$ **-** $(t-1)_u$, with $(t-1)_u$ equal to $t-x$, and x the sampling rate (see Fig. 3.6). The sampling rate x is how often ΔQs are calculated. x can be chosen by the user in the oscilloscope.
This is to account for various overheads that need to be taken into consideration. They could be network overheads, the adapter overhead, C++ latency and more. Imagine multiple outcome instances that are ended at a time slightly lower but close to t, and due to the overheads the messages arrive at a time slightly higher but close to t, the outcome instance would not be taken into consideration for the calculation of a ΔQ.

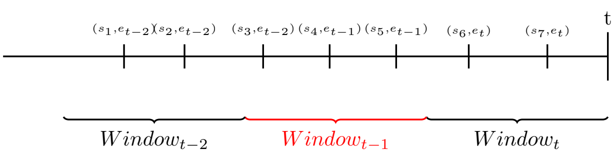

Figure 3.6: The window$_{t-1}$ is the current window being displayed at time t.

The sampling window then advances every x seconds, setting the new window:

From: $(t-1)_l$, $(t-1)_u \xrightarrow{t+1} t_l, t_u$. Where: $t_l = (t-1)_u$ and $t_u = (t-1)_u + x$

The ΔQs which are observed and calculated in a sampling window are not precise, this is why we need to introduce the polling window.

3.4.2 Polling window (Observing multiple ΔQs over a time interval)

The polling window is the window of past ΔQs which are stored to keep a snapshot of the system over time and over which confidence bounds are calculated. The polling window serves to improve the precision of the measurement of the confidence bounds for ΔQs.

Suppose we are at time $t = 0$, the polling window will have 0 ΔQs. As the sampling window advances, more ΔQs are sampled, which in turn are added to the polling window, to the snapshot, and to the confidence bounds calculation.

The current limit of ΔQs for a polling window is 30 ΔQs. At $t = 31$, the older ΔQs will be removed from the polling window and in turn from the confidence bounds. Newer sampled ΔQs will be added, keeping the limit of ΔQs in a polling window to 30. A larger polling window could be used to increase precision of ΔQ computation, i.e. to reduce the width of the confidence interval. This is nevertheless only valid for stationary systems where the ΔQ does not vary over time.

3.5 Triggers

Much like an oscilloscope that has a trigger mechanism to capture periodic signals or investigate a transient event [39], the ΔQ oscilloscope has a similar mechanism that can recognise when an observed ΔQ violates certain conditions regarding required behaviour and record snapshots of the system.

Each time an observed ΔQ of a sampling window is calculated, it is checked against the requirements set by the user. If these requirements are not met, a trigger is fired and a snapshot of the system is saved to be shown to the user.

3.5.1 Snapshot

A snapshot of the system gives insights into the system before and after a trigger was fired. It gives the user a *still* of the system, as if it was frozen in time.

We can define a **"snapshot window"** which is equal to the polling window when no triggers are fired. The maximum size of the snapshot window under normal conditions (no triggers fired) is 30 ΔQs, like the polling window. All the ΔQs which are observed and calculated in a polling window are stored away in both the snapshot and polling windows. Then, if no trigger is fired, older ΔQs are removed from both.

Instead, if a trigger is fired, the ΔQs sampled for the next 5 seconds are added to the snapshot window, without removing older ones from the window. This it to allow the user to look at the state of the system before and after the trigger. After these 5 seconds the recorded snapshot window will contain the 30 ΔQs before the trigger being fired, which is equal to the polling window before the trigger, plus the ΔQs sampled in the 5 seconds after the trigger being fired. This snapshot window is stored away and can be observed by the user.

The polling window will always keep 30 ΔQs, it still follows the same strategy of removing older ΔQs, even if a trigger was fired. Meanwhile, the snapshot window's maximum size increases during the recording of one, keeping older ΔQs even if the size of the snapshot window is > 30. When the snapshot finishes recording at a time t, the snapshot window will be the same as the polling window at time t, they will contain the same ΔQs.

Chapter 4

Oscilloscope: User level concepts

The following chapter gives insights on the user level concepts of ΔQSD in the oscilloscope. They are the concepts needed by the user to understand how the oscilloscope works.

- We first provide insights into how the concepts related ΔQSD are implemented in the oscilloscope, the parameters that define a probe's ΔQ, its representation. We show how probe's ΔQ(s) will be shown in the oscilloscope.
- We then provide a language to write outcome diagrams based on an already existing syntax and provide an example.
- Lastly, we explain the different widgets on the oscilloscope dashboard and the available triggers.

4.1 ΔQSD concepts

We provide in this section the concepts needed to understand what is displayed on the oscilloscope.

4.1.1 Representation of a ΔQ

We provide a class to calculate the ΔQ of a probe in a sampling window between a lower time bound t_l and an upper time bound t_u. It can be calculated in two ways:

Observed ΔQ The first way is by having n collected outcome instances whose end time is between t_l and t_u, calculating the PDF of the delays of the outcome instances, and calculating the resulting CDF based on the PDF. This is called the **Observed ΔQ**.

Calculated ΔQ A ΔQ can also be calculated by performing operations on two or more observed ΔQs (convolution, operators operations), the notion of outcome instances is lost between calculations, as the interest shifts towards calculating the resulting PDFs and CDFs. This is called the **Calculated ΔQ**. A simple outcome can **not** have a "calculated ΔQ", we can only observe the delay from its observables.

If you recall Fig. 3.4, the probes o_2 and o_3 observe simple outcomes, they can only display the observed ΔQs of o_2, o_3. The probe p_1 instead observes the sequential composition of said outcomes. We can display its "observed ΔQ" from the execution from *start* to *end* and the "calculated ΔQ" as the convolution of the observed ΔQs of o_1, o_2

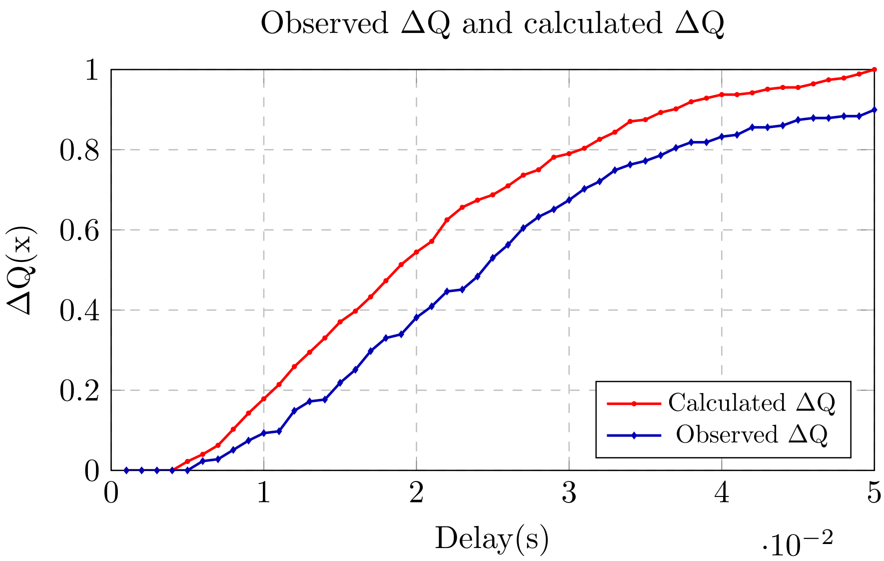

Figure 4.1: (Red, circle, above): Sampling window **Calculated ΔQ**. (Blue, diamond, below): Sampling window **Observed ΔQ**

4.1.2 dMax = Δt · N

The key concept of ΔQSD is having a maximum delay after which we consider that the execution is timed out. This is represented in the oscilloscope as $dMax$. Understanding this equation is key to correctly using the oscilloscope and exploring tradeoffs

Let us explain the following equation:

$$dMax = \Delta t \cdot N \tag{4.1}$$

- $dMax$: The maximum delay, it represents the maximum delay that an outcome instance of a probe can have. The execution is considered "timed out" (failure) after $dMax$.
- Δt: The resolution of a ΔQ. It is the bin width of a bin in a probe's ΔQ.
- N: The precision of a ΔQ. It is the number of bins in a probe's ΔQ.

It can be informally described as a "two out of three" equation. If the user wants higher precision but the same $dMax$, the resolution must change, and so on for every parameter.

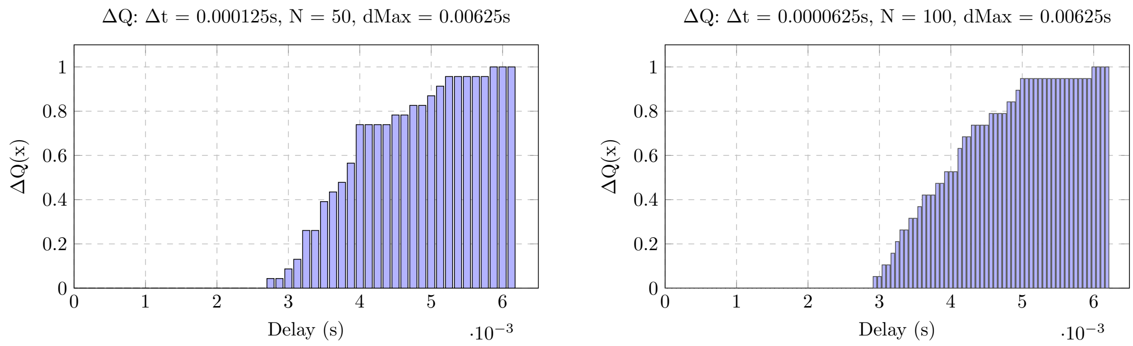

Figure 4.2: **Left**: Sample ΔQ representation as a histogram with higher resolution but lower precision.
Right: Sample ΔQ representation as a histogram with lower resolution but higher precision.
Both ΔQs have the same $dMax$, but the amount of precise information they provide is far different.

Setting a maximum delay for a probe is not a job that can be done one-off and blindly, it is something that is done with an underlying knowledge of the system inner-workings and must be thoroughly fine-tuned during the execution of the system by observing the resulting distributions of the obtained ΔQs.

Some tradeoffs must though be acknowledged when setting these parameters, a higher number of bins corresponds to a higher number of calculations and space complexity, a lower $dMax$ may correspond to more failures. The user must set these parameters carefully during execution.

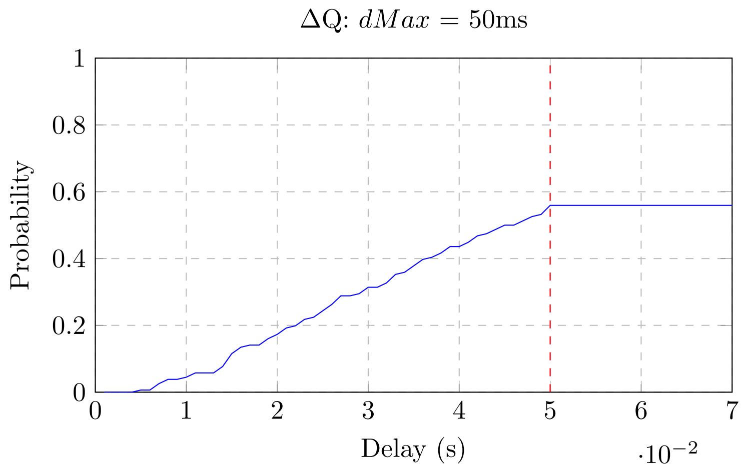

Figure 4.3: ΔQ: $dMax$ = 50ms, the ΔQ will stay constant when delay > $dMax$.

dMax limitation

$dMax$ can **not** be lower than 1 millisecond and will be rounded to the **nearest** integer in the adapter, this is a limitation of Erlang `send_after` function which only accepts integers and milliseconds values. For example, if on the oscilloscope the $dMax$ is equal to $1.56ms$, the adapter will fail spans after 2 ms.

4.1.3 QTA

A simplified QTA is defined for probes. We define 4 points for the step function at 25, 50, 75 percentiles and the minimum percentage p of successful instances accepted for an observable ($0 < p < 1$). The QTA is comparable to the one we have shown in Fig. 2.3. The user can set the QTAs they want for the probes they have inserted, but the delays at the percentiles must be $< dMax$.

4.1.4 Confidence bounds

To observe the stationarity of a probe we must observe its ΔQs over a polling window and calculate confidence bounds over said ΔQs. A single ΔQ may fluctuate. This is why we include the mean and confidence bounds of ΔQs in the plot, which give a probability range over which the true CDF of the ΔQ should fall. [40]

Confidence bounds are given for *observed* and *calculated* ΔQs in a polling window (Fig. 4.4).
We first calculate a mean of the ΔQs in the polling window, this gives an idea of how the probe has been behaving during the polling window. Given this mean, we can calculate its confidence bounds.

The bounds are updated dynamically by inserting or removing a ΔQ to the current polling window. Every time a new ΔQ is sampled, the oldest ΔQ in a window is removed if #ΔQs(polling window) > limit. The new ΔQ is added to the calculation of the mean and confidence bounds as it is sampled.

This allows us to consider a small window of execution rather than observing the execution since the start of the system for the bounds. This can help in observing stationarity of the system, where a fewer number of past ΔQs can help observe short term behaviour.

With a big window of ΔQs, temporary overload may not greatly affect the mean and bounds, while, if we consider the current size of the polling window (30 ΔQs), a few ΔQs which deviate from stationary behaviour have a greater impact on the bounds and mean.

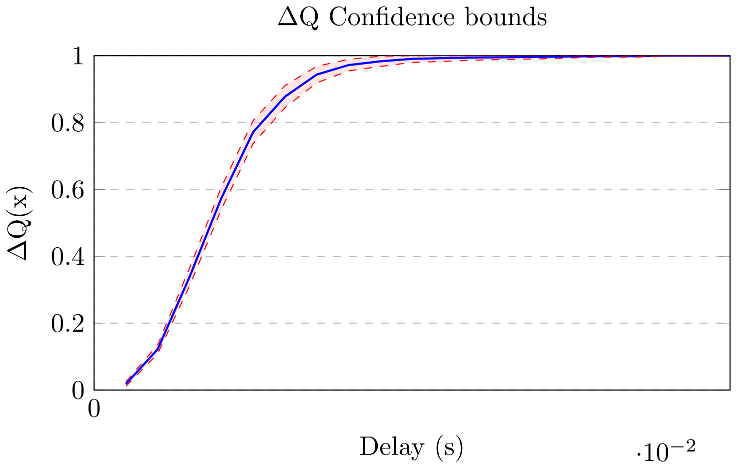

Figure 4.4: Upper and lower bounds (dashed, red) of the mean (blue) of multiple ΔQs. Confidence bounds can help recognise non-linearity. If the confidence bounds of calculated ΔQs are outside the observed ΔQs bounds, we know that the difference between the two is not just a measurement fluctuation, there is instead divergence from linear behaviour.

4.2 ΔQ display

Now that we have introduced the required concepts, we can put everything together to be plotted. In summary, a probe's displayed graph must contain:

- The observed ΔQ of the sampling window, with the mean and confidence bounds calculated over the polling window of observed ΔQs.
- If applicable, the calculated ΔQ of the sampling window from the causally linked components observed in a probe, with the mean and confidence bounds calculated over the polling window of calculated ΔQs.
- Its QTA (if defined).

This allows for the user to observe if a ΔQ has deviated from normal execution, analyse its stationarity, non-linearity and observe a sampled ΔQ over a polling window.

In the screenshot below (Fig. 4.5) we can observe the multiple elements as they are displayed in real time in the oscilloscope.

- (1, green): The mean of the polling window observed ΔQs (yellow) with the confidence bounds. Upper bound (dark green) and lower bound (light green). The observed ΔQ of the sampling window can be observed going out of the confidence bounds at delay 0.00125 s. The ΔQ in a sampling window is less precise than the mean and confidence bounds calculated in the polling window.
- (2, red): The mean of the calculated ΔQs (ochre) with the confidence bounds of the mean. Upper bound (purple) and lower bound (magenta). The calculated ΔQ of the sampling window is inside its confidence bounds.
- (3, blue): The **QTA**.

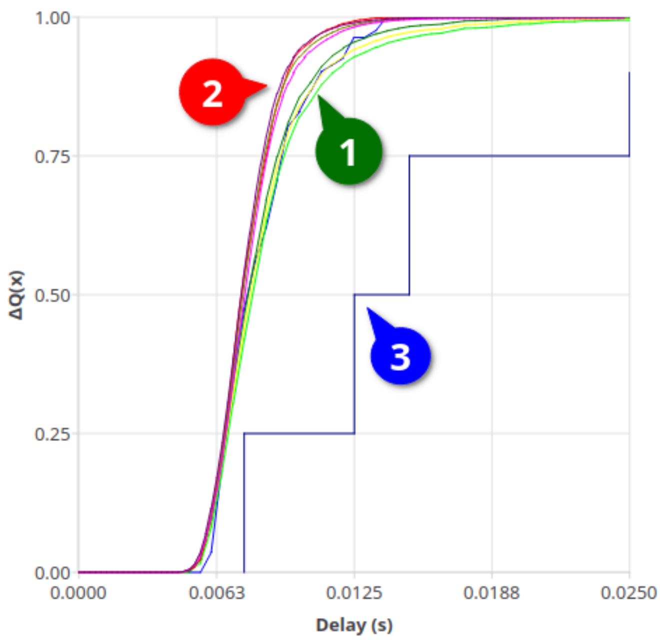

Figure 4.5: ΔQs, confidence bounds, means and QTA for a probe observing the causal link of multiple components.

4.3 Outcome diagram

An abstract syntax for constructing outcome diagrams has already been defined in a previous paper [16], nevertheless, the oscilloscope needs a textual way to define an outcome diagram.
We define a grammar to create an outcome diagram in our oscilloscope, which is a textual interpretation of the abstract syntax.

4.3.1 Causal link

A causal link between two components can be defined by a right arrow from `component_i` to `component_j`:

```
component_i -> component_j
```

4.3.2 Sub-outcome diagrams

Multiple sub-outcome diagrams can be created for multiple parts of the system. They can then be linked together to form the global system outcome diagram. Sub-outcome diagrams can observe one or multiple components. Recall Fig. 3.4, we defined a probe which observes the sequential composition of o_1, o_2. The probe (sub-outcome diagram) p_1 can be defined as:

```
p_1 = o_1 -> o_2;
```

A probe is attached at the start and end events of p_1, it will observe the whole system and the calculated ΔQ will be the convolution of o_1, o_2.

The lines defining these diagrams must be semicolon terminated. Outcomes and operators cannot be defined on their own, they must be observed in a sub-outcome diagram.

Sub-outcome diagrams can be reused in other diagrams by adding `s:` (sub-outcome diagram) before they are used.

```
p_3 = s:p_1 -> s:p_2;
```

This allows for easy composition and reuse of different parts of the system, allowing for independent refining of diagrams.

4.3.3 Outcomes

To attach a probe to an outcome observables, it is enough to declare an outcome with its name inside a diagram.

```
... = outcomeName;
```

4.3.4 Operators

First-to-finish, all-to-finish and probabilistic choice operators must contain at least two components.

All-to-finish operator

An all-to-finish operator needs to be defined as follows:

```
a:name(component1, component2...)
```

First-to-finish operator

A first-to-finish operator needs to be defined as follows:

```
f:name(component1, component2...)
```

Probabilistic choice operator

A probabilistic choice operator needs to be defined as follows:

```
p:name[probability_1, probability_2, ... probability_i](component_1,
↪  component_2, ..., component_i)
```

In addition to being comma separated, the number of probabilities inside the brackets must match the number of components inside the parentheses. For n probabilites p_i, $0 < p_i < 1$, $\sum_{i=0}^{n} p_i = 1$

4.3.5 Limitations

Our system has a few limitations compared to the theoretical applications of ΔQ, namely, no cycles are allowed in the definition of outcome diagrams.

```
p_1 = s:p_2;
p_2 = s:p_1;
```

The above example is not allowed and will raise an error when defined.

4.3.6 Outcome diagram example

We provide a sample example of an outcome diagram definition. We also provide its resulting outcome diagram with probes inserted.

```
two_hops = o2 -> o3;
total = p:pc[0.9, 0.1](o1, s:two_hops);
```

Figure 4.6: Sample textual definition of an outcome diagram.

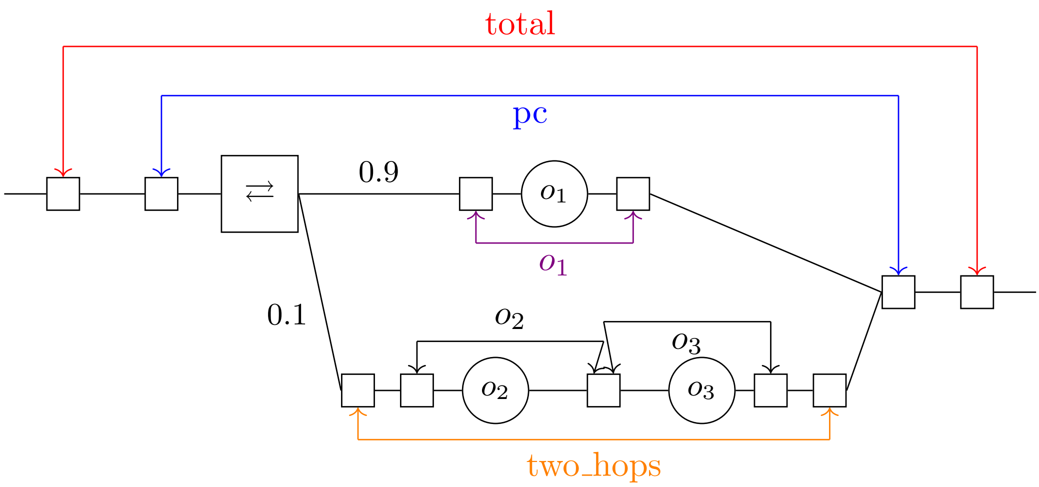

Figure 4.7: Resulting outcome diagram for definition 4.6.

4.4 Dashboard

The dashboard is devised of multiple sections where the user can interact with the oscilloscope, create the system, observe the behaviour of its components, set triggers.

4.4.1 Sidebar

The sidebar has multiple tabs, we explain here the responsibility of each one.

System/Handle plots tab

In this tab, one can create the outcome diagram, add plots and modify current plots. A photo of the tab can be found in Section B.1.

System creation In this wdget the user can create its system with the outcome diagram grammar. They can save the outcome diagram text or load it, the outcome diagram definition is saved to a file with any extension, we nevertheless define an extension to save the system to, the extension `.dq`. If the definition of the input is wrong, they will be warned with a pop-up giving the error the parser generator encountered in the creation of a system.

Adding a plot Once the system is defined, the user can choose the probes they want to plot. They can select multiple probes per plot and display multiple plots on the oscilloscope window.

Sampling rate The user can choose the sampling rate of the system: How often ΔQs are sampled and displayed in the oscilloscope.

Editing a plot By clicking onto a plot that is being shown, the user can choose to add or remove probes to and from it, thanks to the widget in the lower right corner. Multiple probes can be selected to either be removed or added.

Parameters tab

In this tab, the user can define parameters for the probes they have defined. A photo of this tab can be found in Section B.2.

Set a QTA The user is given the choice to set a QTA for a given probe. They have 4 fields where they can fill in which correspond to the percentiles and the minimum amount of successful instances accepted. They can change this dynamically during execution. If a new $dMax$ is set, where $dMax <$ the defined delays in the QTA, the QTA is reset.

dMax, bins The user can set the parameters we explained previously, Δt and N. When this information is saved by the user, the new $dMax$ is transmitted to the adapter and saved for the selected observable.

Triggers tab

In the triggers tab the user can set triggers and observe the snapshots of the system. A photo of the tab can be found in Section B.3.

Set triggers The user can set which triggers to fire for the probes they desire, they are given checkboxes to decide which ones to set as active or not (by default, the triggers are deactivated).

Fired triggers Once a trigger is fired, the oscilloscope starts a timer for 5 seconds, during which all probes start recording the observed ΔQs (and the calculated ones if applicable) without discarding older ones. Once the timer expires, the snapshot is saved for the user in the triggers tab. In the dashboard, it indicates when the trigger was fired (timestamp) and the name of the probe which fired it.

Snapshot window Clicking on a snapshot, a new window opens. The user can explore a frozen state of the system, being able to explore all the ΔQs saved in a snapshot. A screenshot of the snapshot window is provided in Section B.4.

Connection controls

Here, the user can connect to the Erlang endpoint. They can also start the oscilloscope server to receive outcome instances from the adapter. An image of the tab can be found in Section B.5.

Erlang controls The user can set the IP and the port where the ΔQ adapter is listening from. Two additional buttons communicate with the adapter by sending messages, they can start and stop the adapter's sending of outcome instances.

C++ server controls The user can set the IP and the port for the oscilloscope's server, where it will receive outcome instances from the adapter.

4.4.2 Plots window

To the left, the main window shows the plots of the probes being updated in real time. The window can be seen in Section B.2, Section B.3.

4.5 Triggers

There are two available triggers which can be selected by the user, the triggers are evaluated on the **observed ΔQ** of a sampling window.

Part of future work is extending the available triggers in the oscilloscope. We are aware that the number of available triggers may not seem like much. We believe nevertheless that the triggers we provide are a sufficient basis to observe rare events and detect non-linearity or overload.

4.5.1 Load

A trigger on an observed ΔQ *of a sampling window* can be fired if the amount of outcome instances received for a probe in a sampling window is greater than what the user defines:

$$\#\mathrm{instances}_{probe}(\Delta\mathrm{Q}) > \mathrm{maxAllowedInstances}_{probe}$$

4.5.2 QTA

A trigger on an observed ΔQ *of a sampling window* can be fired if:

$$\Delta\mathrm{Q}_{obs} \not< \mathrm{observableQTA}$$

Chapter 5

Oscilloscope: implementation

The following chapter gives a more technical description of the internals of the oscilloscope.

- We provide a more in-depth look at the ΔQSD concepts introduced in the previous chapter.
- We then explain how the ΔQ adapter works, its API and the underlying mechanism that let us export outcome instances to the oscilloscope.
- Lastly, we briefly talk about the parser generator used to parse the outcome diagram syntax and the dashboard graphical framework.

5.1 ΔQSD implementation

In this section we provide the mathematical foundations of ΔQSD as they are implemented in the oscilloscope.

5.1.1 Histogram representation of ΔQ

We approximate the probability distribution of ΔQ via histograms (Fig. 5.1).

PDF

We partition the values into N bins of equal width, Given $[x_i, x_{i+1}]$ the interval of a bin i, where $x_i = i\Delta t$, and $\hat{p}(x_i)$ the value of the PDF at bin i, for n bins [41]:

$$\begin{cases} \hat{p}(i) = \dfrac{s_i}{n}, \text{if } i \leq n \\ \hat{p}(i) = 0, \text{if } i > n \end{cases} \tag{5.1}$$

Where s_i the number of successful outcome instances whose elapsed time is contained in the bin i, n the total number of instances.

CDF

The value $x_i = \hat{f}(i)$ of the CDF at bin i with n bins can be calculated as:

$$\begin{cases} \hat{f}(i) = \sum_{j=1}^{i} \hat{p}(j), & \text{if } i \leq n \\ \hat{f}(i) = \hat{f}(n), & \text{if } i > n \end{cases} \tag{5.2}$$

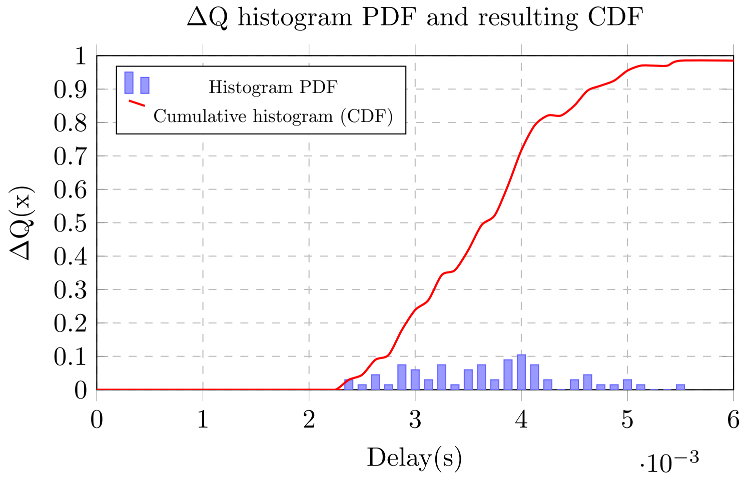

Figure 5.1: Blue bins: PDF of a sample ΔQ. Red: Resulting CDF of ΔQ PDF, the CDF is what is displayed on the dashboard.

5.1.2 dMax

We introduced $dMax$ in the previous chapters (Eq. (4.1)), we provide here the full equation that allows $dMax$ to be calculated:

$$\begin{aligned} \Delta t &= \Delta t_{base} \cdot 2^n \\ dMax &= \Delta t_{base} \cdot 2^n \cdot N \end{aligned} \tag{5.3}$$

Where:

- Δt_{base} represents the base width of a bin, equal to 1ms.
- n the exponent that is set by the user in the dashboard. It is limited to [-10, 10].
- N the number of bins, it is limited to [1, 1000].

We chose 1 ms in combination with 2^n as it allows us to go from very fine bin widths ($\approx$ 1 μs) to large bin widths ($\approx$ 1 s), thanks to the [-10, 10] bounds. Moreover, scaling by a power of 2 allows all the probe's ΔQs to have a common factor to perform operations on them after rebinning (Eq. (5.4)).

5.1.3 Rebinning

Rebinning refers to the aggregation of multiple bins of a bin width i to another bin width j [42]. Previous operations between ΔQs must be done on ΔQs that have the same bin width. This is why it is fundamental that all probes have a common Δt_{base} and why we have a 2^n factor to calculate the total bin width.

Given two ΔQs ΔQ_i, ΔQ_j, the common bin width Δt_{ij} is:

$$\Delta t_{ij} = \max \{\Delta t_i, \Delta t_j\}$$

The PDF of the rebinned ΔQ (see Fig. 5.2) at bin b, from the original PDF of n bins, where $\Delta t_i > \Delta t_j$ and k = $\frac{\Delta t_i}{\Delta t_j}$:

$$p'_b = \sum_{n=b\cdot k}^{b+1\cdot k-1} p_n, \quad b = 0, 1, \dots \lceil \frac{N}{k} \rceil \tag{5.4}$$

We perform rebinning to a higher bin width for a simple reason. While this leads to loss of information for the ΔQ with the lowest bin width, rebinning to a lower bin width would imply inventing new values for the ΔQ with the highest bin width.

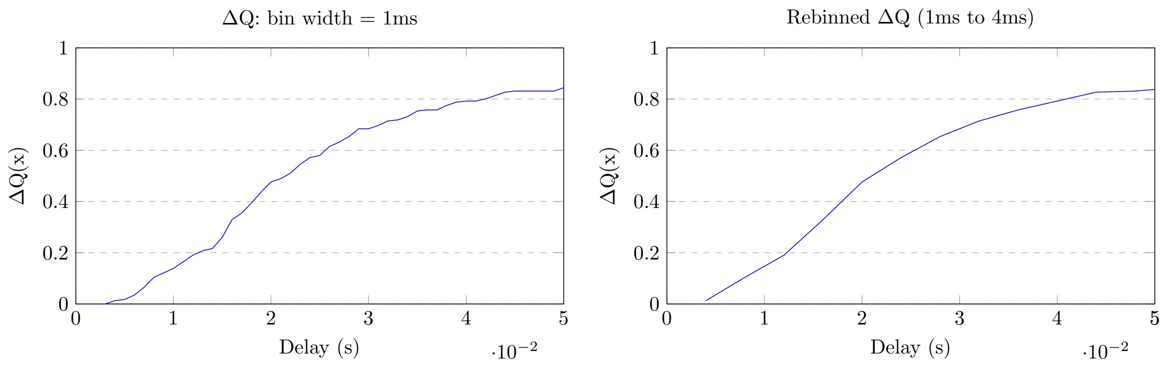

Figure 5.2: **Left**: Sample ΔQ with 1ms bins. **Right:** ΔQ on the left after rebinning to 4ms bins.

5.1.4 Convolution

We present the two solutions to perform convolution we explored during the implementation.

Naïve convolution

Given two ΔQ binned PDFs f and g with equal bin widths, the result of the convolution $f \circledast g$ is given by [43]:

$$(f \circledast g)[n] = \sum_{m=0}^{N} = f[m]g[n-m] \tag{5.5}$$

The naïve way of calculating convolution has a time complexity of $\mathcal{O}(N^2)$, this quickly becomes a problem as soon as the user wants to have a more fine-grained understanding of a component. The oscilloscope presented noticeable lag and frame skipping with high resolution. This is why we decided to explore Fast Fourier Transform convolution.

Fast Fourier Transform Convolution

FFTW (Fastest Fourier Transform in the West) is a C subroutine library [44] for computing the discrete Fourier Transform in one or more dimensions, of arbitrary input size, and of both real and complex data. We use FFTW in our program to compute the convolution of ΔQs. We adapt our script from an already existing one found on GitHub. [45]

Whilst the previous algorithm is far too slow to handle a high number of bins, convolution leveraging Fast Fourier Transform (FFT) allows us to reduce the amount of calculations to $\mathcal{O}(N \log N)$. This is why the naïve convolution algorithm is not used. We will analyse the time gains in Chapter 7.

FFT and naïve convolution produce the same results in our program, barring ε differences (around 10^{-18}) in bins whose result should be 0. This is most likely due to rounding error.

FFTs algorithms are plenty, the choice of the one to use is left up to the subroutine via the parameter `FFTW_ESTIMATE`. [46]

The explanation of the algorithm is beyond the scope of this master thesis, we refer the reader to the following book as a reference. [47]

5.1.5 Arithmetical operations

The FTF, ATF and PC operators on ΔQs use a simple set of arithmetical operations to calculate a ΔQ. The time complexity of FTF, ATF and PC is trivially $\mathcal{O}(N)$ where N is the number of bins.

Scaling (multiplication) A ΔQ can be scaled w.r.t. a constant $0 \leq j \leq 1$. It is equal to binwise multiplication on CDF bins. It is used for the probabilistic choice operator.

$$\hat{f}_r(i) = \hat{f}(i) \cdot j \tag{5.6}$$

Operations between ΔQs Addition, subtraction and multiplication can be done between two ΔQ of equal bin width (but not forcibly of equal length) by calculating the operation between the two CDFs of the ΔQs:

$$\Delta \mathrm{Q}_{AB}(i) = \hat{f}_A(i)[\cdot, +, -]\hat{f}_B(i) \tag{5.7}$$

They are used for all operators.

5.1.6 Confidence bounds

Here is how we calculate the mean and lower/upper confidence for the ΔQs CDF at bin $i \quad \forall i < N$. [41]

For x_{ij}, the value of a CDF j at bin i, the mean of n CDFs for the bin i in a polling window is:

$$\mu_i = \frac{1}{n} \sum_{j=1}^{n} x_{ij} \tag{5.8}$$

Its variance:

$$\sigma_i^2 = \frac{1}{n}\sum_{j=1}^{n} x_{ij}^2 - \mu_i^2 \tag{5.9}$$

The lower confidence bound $CI_{i,l}$ and upper confidence bound $CI_{i,u}$ for a bin i can be calculated as:

$$CI_{i,l} = \mu_i - \frac{\sigma_i}{\sqrt{n}}\ ,\ CI_{i,u} = \mu_i + \frac{\sigma_i}{\sqrt{n}} \tag{5.10}$$

The confidence bounds for the CDF are the lower and confidence bounds for every bin i.

5.2 Adapter

The adapter, called `dqsd_otel` is a rebar3 [48] application built to replace OpenTelemetry calls and create outcome instances. It is designed to be paired with the oscilloscope to observe an Erlang application.

5.2.1 API

The adapter functions to be used by the user are made to replace OpenTelemetry calls to `?start_span` and `?with_span` and `?end_span` macros. This is to make the adapter less of an encumbrance for the user and to extend the tool usefulness in observing distributed programs.

The adapter will always start OpenTelemetry spans but only start outcome instances if the adapter has been activated. The adapter can be activated by the oscilloscope by pressing the *"start adapter"* button and can be stopped via the *"stop adapter"* button.

start_span/1, start_span/2

```
start_span/1: -spec start_span(binary()) -> {opentelemetry:span_ctx(),
↪  pid() | ignore}.
start_span/2: -spec start_span(binary(), map()) ->
↪  {opentelemetry:span_ctx(), pid() | ignore}.
```

Parameters:

- Name: Name of the probe in Erlang binary representation. For example, if the name of the probe is "probe", the binary representation would be constructed with "`<<"probe">>`" [49] [50]. We will refer to this as "binary name" from now on.
- Attributes: The OpenTelemetry span attributes (Only for start_span/2).

`start_span` incorporates OpenTelemetry `?start_span(Name)` macro.

Return: The function returns either:

- `{SpanCtx, span_process_PID}` if the adapter is active and the probe's $dMax$ has been set.
- `{SpanCtx, ignore}` if one of the two previous conditions was not respected.

With SpanCtx being the context of the span created by OpenTelemetry.

with_span/2, with_span/3

```
with_span/2: -spec with_span(binary(), fun(() -> any())) -> any().
with_span/3: -spec with_span(binary(), fun(() -> any()), map()) ->
↪  any().
```

Parameters:

- Name: Binary name of the probe.
- Fun: Zero-arity function representing the code of block that should run inside the `?with_span` macro.
- Attributes: The OpenTelemetry span attributes (Only for with_span/3).

`with_span` incorporates, thus replaces the OpenTelemetry `?with_span` macro.

Return: `with_span` returns what `Fun` returns (`any()`).

end_span/2

```
-spec end_span(opentelemetry:span_ctx(), pid() | ignore) -> ok |
↪  term().
```

Parameters:

- SpanCtx: The context of the span returned by `start_span`.
- Pid: `span_process_PID` || `ignore`.

`end_span` the OpenTelemetry `?end_span(Ctx)` macro.

fail_span/1

```
-spec fail_span( pid() | ignore) -> ok | term().
```

Parameter:

- Pid: `ignore` || `span_process_PID`.

`fail_span` does not incorporate any OpenTelemetry macro, it is let up to the user to decide how to handle failures in execution.

span_process

`span_process` is the process, spawned by `start_span`, responsible for handling the `end_span, fail_span, timeout` messages.

Upon being spawned, the process starts a timer with time equal to the $dMax$ set by a user for the probe being observed, thanks to `erlang:send_after`. When the timer runs out, it sends a `timeout` message to the process.

The process can receive three kinds of messages:

- `{end_span, end_time}`: This will send an outcome instance to the oscilloscope with the start and end time of the execution of the probe and success status.
- `{fail_span, end_time}`: This will send an outcome status to the oscilloscope indicating that an execution of a probe has failed.
- `{timeout, end_time(StartTime +` $dMax$`)}`: If the program hasn't ended the span before $dMax$, the timer will send a `timeout` message. It will send an outcome instance to the oscilloscope indicating that an execution of a probe has timed out.

The `span_process` is able to receive one and only message, if the execution times out and subsequently the span is ended, the oscilloscope will not be notified as the process is defunct. This is assured by Erlang documentation:

If the message signal was sent using a process alias that is no longer active, the message signal will be dropped. [51]

5.2.2 Handling outcome instances

To create outcome instances of a probe we must obtain three important informations:

- Its name.
- The time when the span was started.
- Its $dMax$.

The start time and end time are supplied upon starting/ending a span by calling this function:

```
StartTime/EndTime = erlang:system_time(nanosecond).
```

The name is given when starting a span and the $dMax$ is stored in a dictionary in the adapter.

The outcome instance is created only if two conditions are met: the adapter has been set as active and the user set a timeout for the probe. The functions will then spawn a `span_process` process, passing along all the necessary informations.
Once the span is subsequently ended/timed out/failed, the function `send_span` creates a message carrying all the informations and sends it to the C++ server. The formatting of the messages is the following:

```
n:probe name,
b:Start time (beginning),
e:End time (end time or deadline),
s:The status
```

5.2.3 TCP connection

The adapter is composed of two `gen_server` [52] which handle communication to and from the oscilloscope. This gen_server behaviour allows the adapter to send spans asynchronously to the oscilloscope.

TCP server

The TCP server `dqsd_otel_tcp_server` is responsible for receiving commands from the oscilloscope. It can be run by setting its IP and port via:

```
-spec start_server(string() | binary() | tuple(), integer()) -> ok |
↪ {error, Reason}
```

The oscilloscope can send commands to the adapter, these commands are:

- `start_stub`: This command sets the adapter as active, it can now send outcome instances to the oscilloscope if the probe's $dMax$s are defined.
- `stop_stub`: This commands sets the adapter as inactive, it will no longer send outcome instances to the oscilloscope.
- `set_timeout;probeName;timeout`: This command indicates to the adapter to set the $dMax =$ timeout for a probe, a limit of the adapter is that erlang:send_after does not accept floats as timeouts, so the timeout will be rounded to the nearest integer.

TCP client

The TCP client `dqsd_otel_tcp_client` allows the adapter to send the spans to the oscilloscope. The client connects over TCP to the oscilloscope by connecting to the oscilloscope server's address and opens a socket where it can send the outcome instances.

```
-spec try_connect(string() | binary(), integer()) -> ok.
```

5.3 Parser

To parse the system, we use the C++ ANTLR4 (ANother Tool for Language Recognition) library [53].

5.3.1 ANTLR

According to ANTLR website [53]:

> "ANTLR is a parser generator for reading, processing, executing or translating structured text files. ANTLR generates a parser that can build and walk parse trees."

ANTLR is just one of the many parsers generators available in C++ (flex/bison [54], lex/yacc [55]). Although it presents certain limitations, its generated code is simpler to handle and less convoluted with respect to the other possibilities.

ANTLR uses an Adaptive LL(*) *(ALL(*))* parsing strategy, namely, it will "move grammar analysis to parse-time, without the use of static grammar analysis". [56]

5.3.2 Grammar

ANTLR provides a yacc-like metalanguage [56] to write grammars. Due to page limitations, the grammar we have written can be found in Chapter E.

Limitations

A previous version was implemented in Lark [57], a python parsing toolkit. The python version was quickly discarded due to a more complicated integration between Python and C++. Lark provided Earley(SPPF) strategy which allowed for ambiguities to be resolved, which is not possible in ANTLR.
For example the following system definition presents a few errors:

```
probe = s -> a -> f -> p;
```

While Lark could correctly guess that everything inside was an outcome, ANTLR expects `":"` after "s, a, f" and "p". Thus, one can not name an outcome by these characters, as the ANTLR parser generator thinks that an operator or a probe will be next.

5.4 Oscilloscope GUI

Our oscilloscope graphical interface has been built using the QT framework for C++. Qt is "a cross-platform application development framework for creating graphical user interfaces" [58]. We chose Qt as we believe that it is the most documented and practical library for GUI development in C++. Using Qt allows us to create usable interfaces quickly, while being able to easily pair the backend code of C++ to the frontend.

The interface is composed of a main window, where widgets can be attached to it easily. Everything that can be seen is customisable widgets. This allows for easy reusability, modification and removal without great refactoring due in other parts of the system. We provide a screenshot of the "widget view" in Section B.6.

Chapter 6

Evaluation on synthetic programs

This chapter evaluates the usefulness of the oscilloscope by testing it on three distinct synthetic Erlang applications. Each application was first represented by an outcome diagram in the oscilloscope. It was then instrumented with the adapter to communicate to the oscilloscope. We provide three different examples with simple use cases. Although these use cases may be simple and concise, we will show that the oscilloscope can be a powerful tool in detecting non-linear behaviour with microseconds precision. The examples are:

- A system with sequentially composed outcomes. It server to show how non-linear behaviour can be detected with the paradigm and the oscilloscope, even when the difference in execution times is minimal. The example leverages M/M/1/K queues to show how typical queue behaviour is represented on the oscilloscope.
- Then, we provide two applications that perform synchronisation between components. For this application we use two different operators: first-to-finish and all-to-finish. There, we show how these operators can aid in detecting slower components in the system.

6.1 System with sequential composition

The first application has two sequentially composed components. We choose to model the two components as M/M/1/K queues. This is because an average component in a distributed system can be modeled as an M/M/1/K queue. We assume inter-arrival times, execution times for both components are exponentially distributed and a buffer of size K. When the buffer is full, messages are discarded.

Let us first provide a refresher about M/M/1/K queues:

- λ: The arrival rate.
- The service time s: is the time it takes to serve a message. μ: The service **rate** and E[s] $= \frac{1}{\mu}$.
- Offered load: $\rho = \frac{\lambda}{\mu}$.

We will control λ to show its effects on the offered load. This is because the offered load can tell much about the system:

- At low load ($\rho < 0.8$) the failure will tend to 0. The system is behaving correctly and the ΔQ will show that, as the delay will tend to 1.
- Once ρ is approaching high load ($\rho > 0.8$) we can observe the failure increasing quickly. However, we can observe the system starting to get bad after $\rho > 0.5$. [9]

6.1.1 System's outcome diagram

The system (Fig. 6.1) has two components `worker_1`, `worker_2`. Each individual component is composed of a queue of size $K = 1000$ and a worker process.

The system sends n messages per second following a Poisson distribution to `worker_1`'s queue. The queue notifies its worker if the worker is not busy. The worker performs N loops of fictional work, they are defined upon start and are done to simulate a component performing a task. Once done, `worker_1` then passes a message to `worker_2`'s queue, which has another queue of size $= 1000$, it then passes the message to `worker_2`'s worker, which does the same amount of loops as `worker_1`. When a worker completes its work, it notifies its queue, freeing one message from its buffer size and allowing the next message to be processed.

If the queue's buffer is overloaded, it will drop the incoming message and consider the execution a failure.

A probe named "*probe*" is defined, which observes the execution from when the message is sent to `worker_1` up until `worker_2` is done.

Lastly, both workers share the same processor, to observe the effects of non-linearity in a distributed setting.

The system can be defined via the previously defined syntax (Section 4.3) as:

```
probe = worker_1 -> worker_2;
```

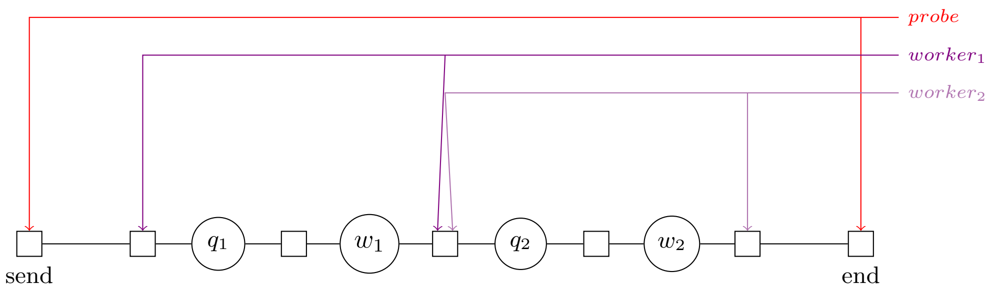

Figure 6.1: Refined outcome diagram of the system. The labeled coloured lines represent the probes that were inserted. $q_{1,2}$ outcomes represent the queues. $w_{1,2}$ represent the workers. As we do not wish to observe the queues, but the whole behaviour of the worker components, we can insert probes from when the message is received to when the worker loops end. We can imagine that $q_{1,2}$ and $w_{1,2}$ are refinements of outcomes $worker_{1,2}$, this is why the probes names differ from the outcomes they observe.

6.1.2 Determining parameters dynamically

We stated previously that determining parameters is something that must be done with an underlying knowledge of the system (Section 4.1.2). The oscilloscope can provide knowledge of the system. Fig. 6.2 shows an example of worker_1 and worker_2 as observed in the oscilloscope. We will send 50 messages per second to observe the system under test to get key properties. The workers do a million loops. The engineer supposes that the workers executions should take a maximum of 4 ms to complete, but doesn't actually know how long the executions should take. The engineer, after having set $dMax = 4ms$, observes the graph Fig. 6.2 on the left in the oscilloscope.

The oscilloscope shows the engineer that their assumptions do not correspond to the actual system ΔQ. The user can then modify the parameters to observe the actual worker's behaviour. By setting $dMax$ to 8 ms, they can observe the worker's ΔQs failure approaching 0 on the right Figure in Fig. 6.2.

On the other hand, the engineer's assumption could have been what they truly expected from the system. In this case, the oscilloscope tells them that the system is not behaving as expected.

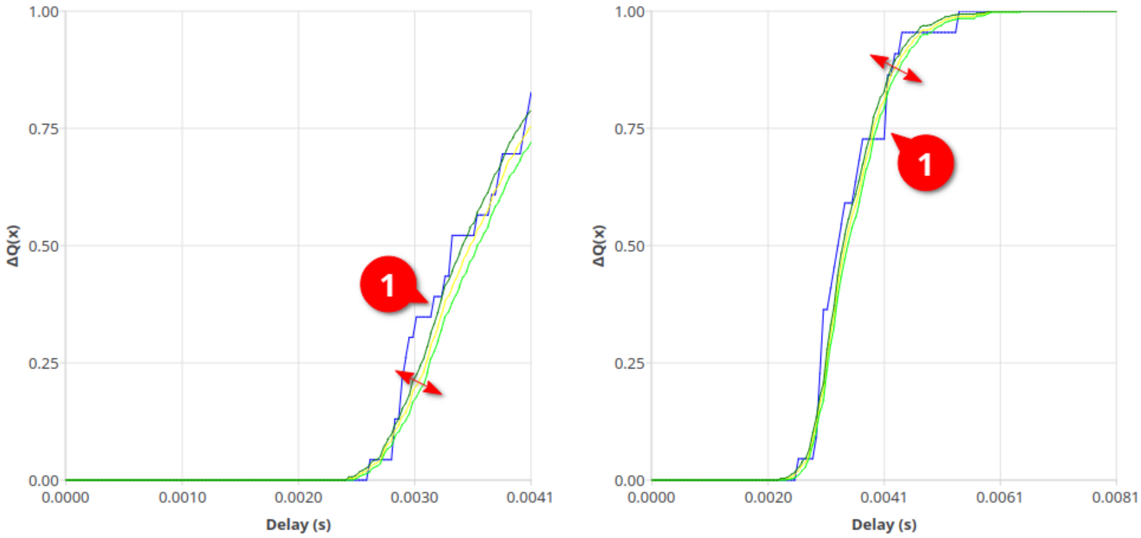

Figure 6.2: **Left**: worker_1 sampling window observed ΔQ with $dMax = 4ms$. Failure tends to 0.25. **Right**: worker_1 sampling window observed ΔQ with $dMax = 8$ ms. Failure tends to 0. Legend for both: (1, red) Sampling window observed ΔQ. (Between red arrow): Its confidence bounds.
In both graphs we can observe how the sampling window observed ΔQ (1) is not precise. The step function representation of it fluctuates. The confidence bounds (between red arrow) provide a more precise representation of worker_1.

6.1.3 Observing non-linearity in the oscilloscope

We can observe the system under different loads to observe how non-linearity can appear in the oscilloscope and how it can be detected.

Low Load

We will send 50 messages per second to observe the system under test to get key properties. The workers do a million loops.
We can observe in the left graph in Fig. 6.3 that the average worker's execution takes $\approx$ 33ms. We then have $\mu_{worker} = \frac{1}{0.0033s} \approx 300$ req/s. Thus, $\rho = \frac{50}{300} = 0.1\overline{6}$, we can assume the system is at low load.

At low load (Fig. 6.3), the system is behaving **linearly**. Recall Section 2.1.9, at low load the sum of the two delays will overlap with the observed total delay. We can observe in the oscilloscope the probe **observed ΔQ** and **calculated ΔQ** confidence bounds overlap.

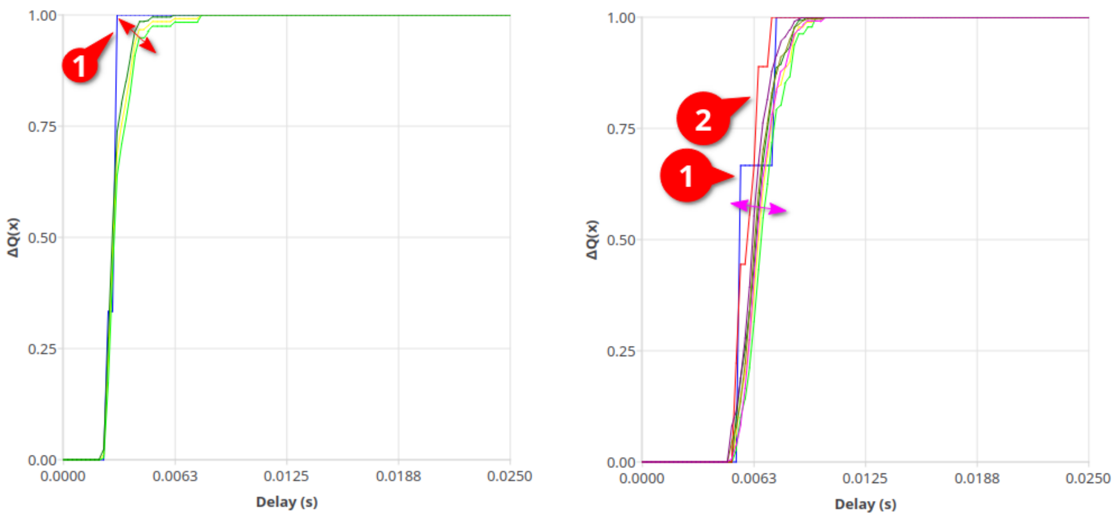

Figure 6.3: Graphs for $\lambda = 50$. **Left**: (1, red) worker_1 sampling window observed ΔQ. (Between red arrow): Its confidence bounds.
Right: (1, red) "probe" sampling window observed ΔQ. (2, red) Sampling window calculated ΔQ. (Between magenta arrows) The observed and calculated ΔQs confidence bounds overlapping.

Early signs of overload

At load = 0.5 the system should start showing early signs of dependent behaviour. We can observe what happens when $\lambda = 150 \rightarrow \rho = 0.5$.

Recall 2.12, a non-linear system does not obey superposition. The sum of the delay of the workers will deviate from the overall delay of "probe". In Fig. 6.4, this is the case. We can start to observe early signs of dependency even at $\rho = 0.5$. The mean of the observed ΔQ is deviating from the calculated one. At the 50th percentile the deviation is *minimal*, around 0.4 ms. Nevertheless, the precision of the paradigm and the oscilloscope allows even for these minimal deviations to appear on the graphs, being able to recognise *non-linearity* and *early signs of overload.*
The superposition principle is not respected anymore, there is apparent **non-linearity**, and the fact that ΔQ can recognise non-linear behaviour with **0.4ms precision** is

impressive. The oscilloscope is a powerful tool that can help assess dependent behaviour in running systems early on.

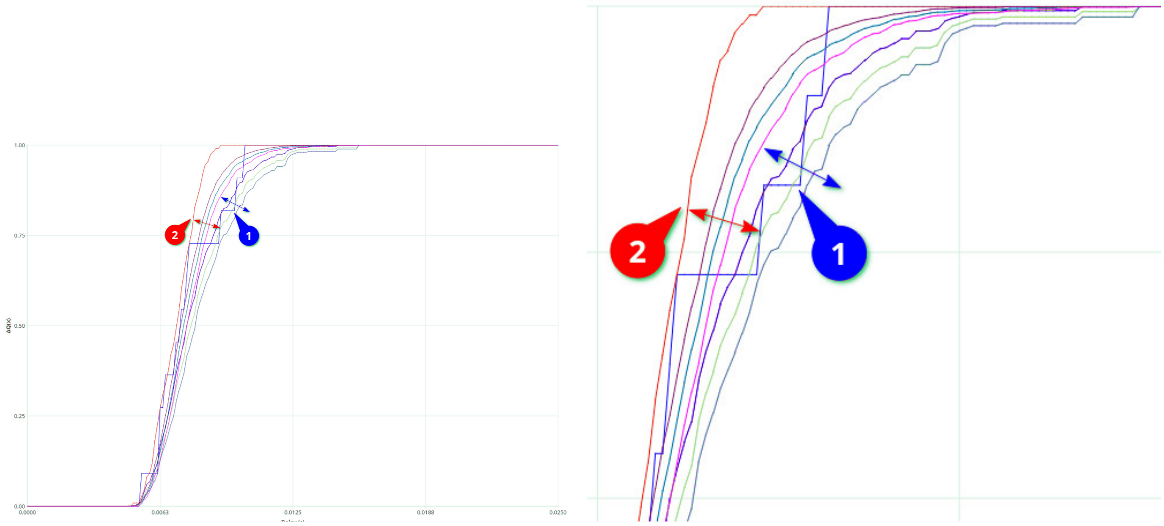

Figure 6.4: **Left**: The probe "probe" ΔQs as observed in a snapshot. **Right**: Zooming in on the oscilloscope, we can observe non-linearity, the ΔQs are diverging. Legend for both: (1, blue) Sampling window observed ΔQ. (Between blue arrow, above): Its confidence bounds. (2) Sampling window calculated ΔQ. (Between red arrow, below) Its confidence bounds.
The observed confidence bounds are > than the calculated ΔQ bounds. The deviation at the median is of 0.4 ms, but it is noticeable in the oscilloscope.

High load

Performance degradation at 0.5 offered load is already remarkable, the observed ΔQs and the calculated ones are slowly but surely deviating. The difference is seemingly minimal, but noticeable. We can go even further and observe the system under high load situations (Fig. 6.5). We set $\lambda = 250 \rightarrow \rho = 0.83$, just above the high load threshold.

The results in Fig. 6.5 confirm what is expected by queueing theory. The oscilloscope is capable of observing the basic observation requirements and capable of recognising dependency. While the worker's observed ΔQ is a nice normally distributed CDF with little to no failure, what we observe on the probe is degraded performance in its observed ΔQ mean.
Due to dependency, high load and processor utilisation, the queue is filling up and the workers are taking more time to complete. If you recall Fig. 6.3, the worker's delay distribution was less than the current one. We can see that it has completely degraded, with the average request taking almost double the time as under normal queueing conditions.

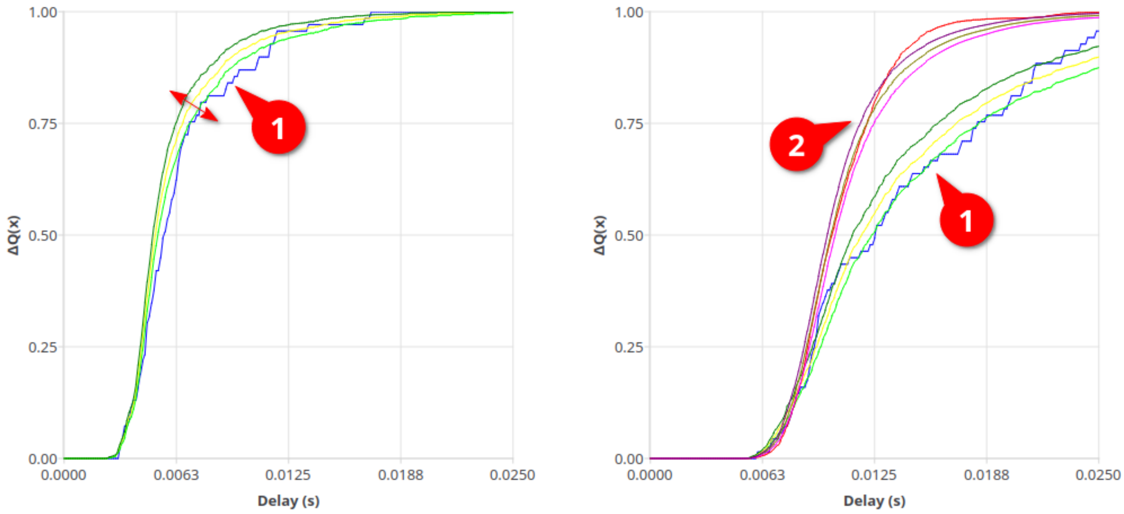

Figure 6.5: Graphs for $\lambda = 250$. **Left**: (1, red) worker_1 sampling observed ΔQ. (Between red arrow) Its confidence bounds. **Right**: "probe" ΔQs (1, red) Sampling window observed ΔQ inside its confidence bounds, (2, red) Sampling window calculated ΔQ inside its confidence bounds.

Further degradation can be observed by increasing $\lambda = 300, 350 \rightarrow \rho \geq 1$ (Fig. 6.6).

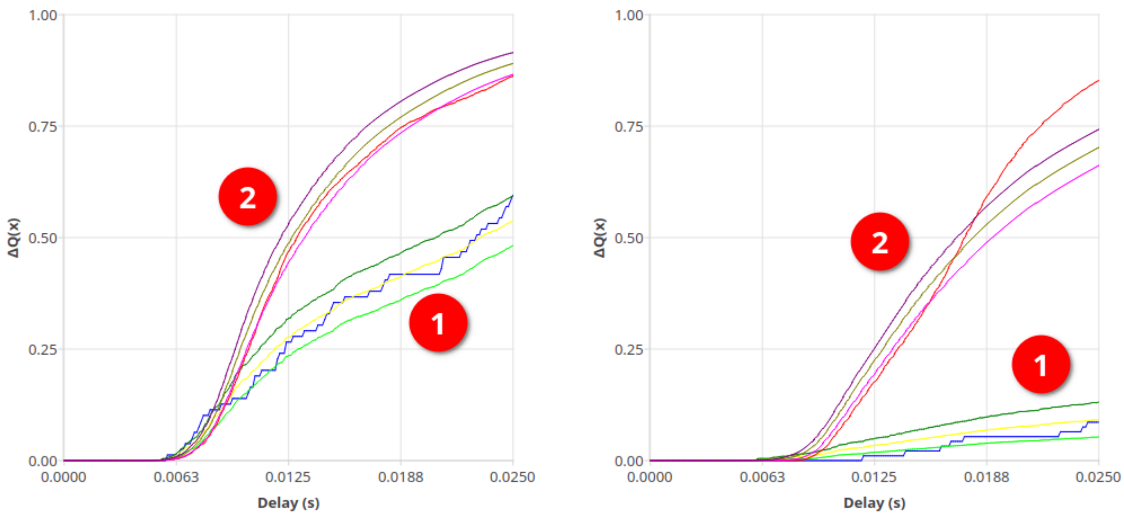

Figure 6.6: "probe" ΔQs. **Left**: $\lambda = 300$. **Right**: $\lambda = 350$. Legend for both: (1, red) Sampling window observed ΔQ inside its confidence bounds. (2) Sampling window calculated ΔQ inside its confidence bounds.

The system degradation is clear, the ΔQs show how almost all messages are being dropped or take $> dMax$. We can now look at triggers and how they can be useful to diagnose such cases.

6.1.4 Detecting non-linearity with triggers

We show examples of how triggers can be set to detected non-linearity after having observed the running system in the oscilloscope.

QTA trigger

Recall the previous definition of ΔQ (Section 4.1.3), by observing the system, we create a QTA for the probe “probe” with: 25% = 0.0075 s, 50% = 0.0125 s, 75% = 0.015s and minimum success rate = 0.9.

By setting the trigger to fire for $\Delta Q_{obs} < QTA$. We captured a handful of snapshots. Here, $\lambda = 150$.

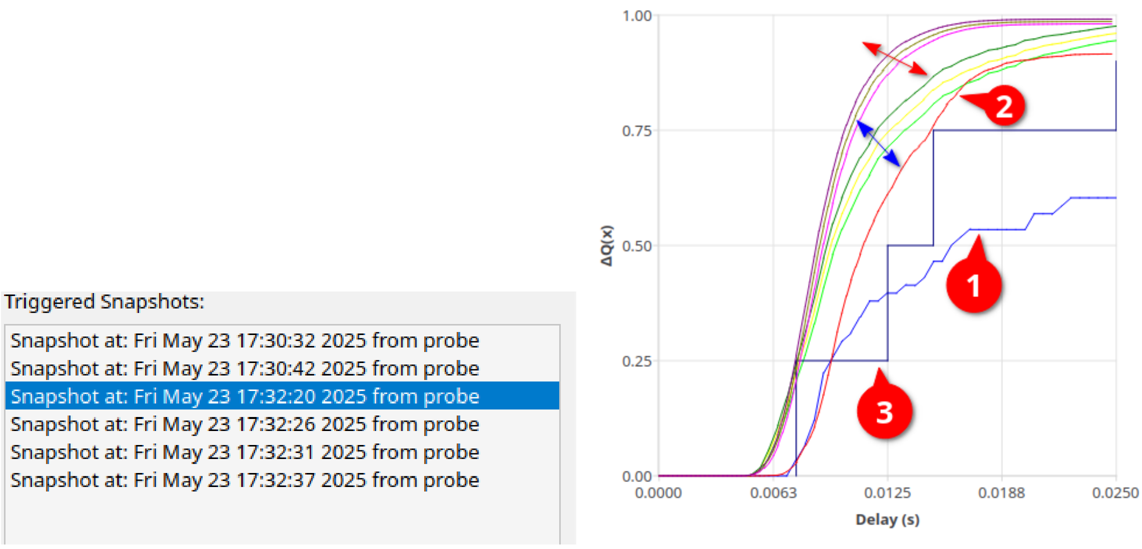

Figure 6.7: **Left**: Snapshots for fired triggers. **Right**: “probe” ΔQs. (1, red) Sampling window observed ΔQ violating the QTA (3, red). (Between blue arrow, below) Its confidence bounds are deviating from the observed confidence bounds. (2) The sampling window calculated ΔQ is behaving worse than its confidence bounds (Red arrow, above). The system is overloaded, degrading and showing non-linear behaviour, this is captured by the QTA violation.

Trigger on number of instances

QTA triggers can help detect non-linearity even before it becomes evident.
By observing the system under test in high load cases, we have found that the system start showing non-linear behaviour with $\lambda = 150$. We set the sampling window to 1 seconds and then set a trigger on the load of “probe” when outcome instances $\gtrsim 150$.

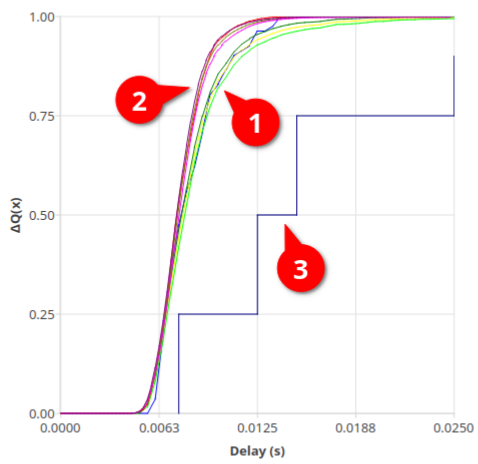

Figure 6.8: Graph of a snapshot for a fired trigger on the load of "probe". (1, red) Sampling window observed ΔQ inside its confidence bounds. (2, red) Sampling window calculated ΔQ confidence bounds. The trigger fires for observed load > 150 in a sampling window of 1 second. Even if the QTA requirement (3, red) is not being violated, the system is showing early signs of non-linear behaviour.

By knowing the inner details of the system, setting a QTA on the number of instances can be useful. Fig. 6.8 is an example of a fired trigger on the number of instances. Even though the QTA requirement isn't being violated, the number of instances fires a trigger, where the user can observe that the system is showing early signs of overload.

6.2 Detecting slower workers in operators

6.2.1 First to finish application

Next, we provide a synthetic application modeling an application that can be modeled by a first to finish operator. The outcome diagram is what is shown in Fig. 2.7. Assume a send request to "the cloud" that waits for a response or a timeout, it is modeled by an FTF operator. [9]

Using the wrong operator

What happens if the wrong operator is chosen to represent the causal relationships between the outcomes? What if the user believes that the system diagram is the one we presented before in Fig. 6.1 (1 in Fig. 6.9)? The result on the oscilloscope will clearly show that something is wrong.

```
ftf = worker_1 -> worker_2; (1)
... = f:ftf(worker_1, worker_2); (2)
```

Figure 6.9: Two outcome diagram definitions proposed by the engineer for ftf.

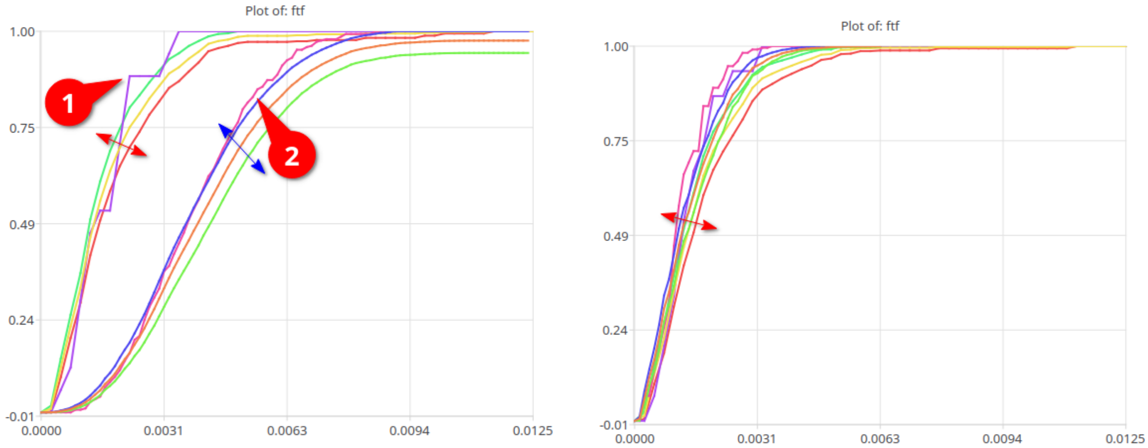

Figure 6.10: **Left**: FTF plot **with wrong outcome diagram definition** as shown in the oscilloscope. (1) Observed ΔQ in sampling window with its confidence bounds between arrows next to it. (2) Calculated ΔQ in sampling window with its confidence bounds between arrows next to it.
Right: FTF plot **with correct outcome diagram definition**. (Red between arrow) Observed ΔQ and calculated ΔQ confidence bounds overlapping.

On the left in Fig. 6.10, we can observe how the sampling window **calculated ΔQ** and its confidence bounds (2) are clearly greater than the sampling window **observed ΔQ** (1) and its confidence bounds. This difference tells us that the proposed outcome diagram does not correctly represent the actual system. On the right, if no dependencies are present and the correct operator (2 in Fig. 6.9) is chosen (right figure in Fig. 6.9), the two ΔQs (observed and calculated) will overlap, as shown on the right in Fig. 6.10.

Introducing a slower component

Let us introduce a slower worker into the system, we introduce an artificial delay into worker_2 (about 20ms). If the oscilloscope works correctly, the paradigm operations are sound and no dependencies are present in the system, we should not see any difference in the observed and calculated ΔQs of the FTF operator. Moreover, the FTF ΔQ could be overlaid on top of the faster worker (`worker_1`). In Fig. 6.11, this is the case, the FTF operator can be overlaid on top of worker_1's graph.

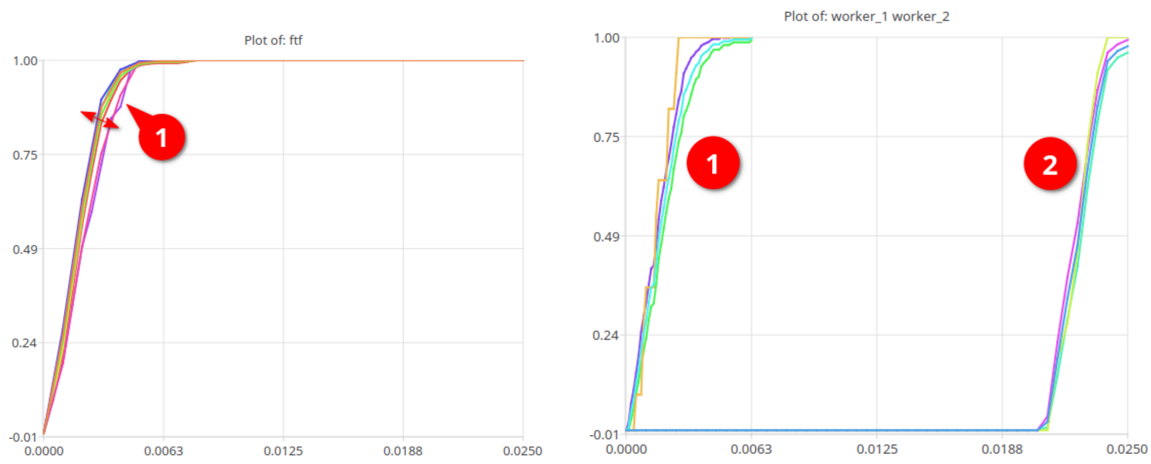

Figure 6.11: **Left**: "ftf" ΔQs. Sampling window observed and calculated ΔQs (1) and their confidences bounds (between red arrow) overlapping. **Right**: worker_1 (1) and worker_2 (2) ΔQs confidence bounds. The FTF plot correctly displays how worker_2 does not have an effect on the FTF plot.

6.2.2 All to finish application

We can extend the previous application to an all-to-finish operator. This operator can for instance represent parallel work, a task that requires a lot of computation and can be done in separate pieces by separate workers. [9]
The outcome diagram is the one we presented in Fig. 2.7 and itcan be represented textually by:

```
... = a:atf(worker_1, worker_2);
```

Introducing a slower component

Like we did for the FTF operator, let's introduce a slower worker into the system. We introduce a slight delay to show how even a few milliseconds can be noticeable right away by a keen eye (or by triggers, which avoids having to look constantly at the graphs). Worker_2 is 2ms slower.

The difference in the worker's ΔQ can be noticed with $\Delta Q_{w2} > \Delta Q_{w1}$ on the right in Fig. 6.12. The difference can then be observed in the all-to-finish plot on the left in Fig. 6.12, where the operator's ΔQs confidence bounds (both observed and calculated) can be overlaid on top of worker_2 ΔQ. This shows once again that the ΔQSD algebraic foundation is sound. Moreover, the oscilloscope can be useful in detecting slower components in a system.

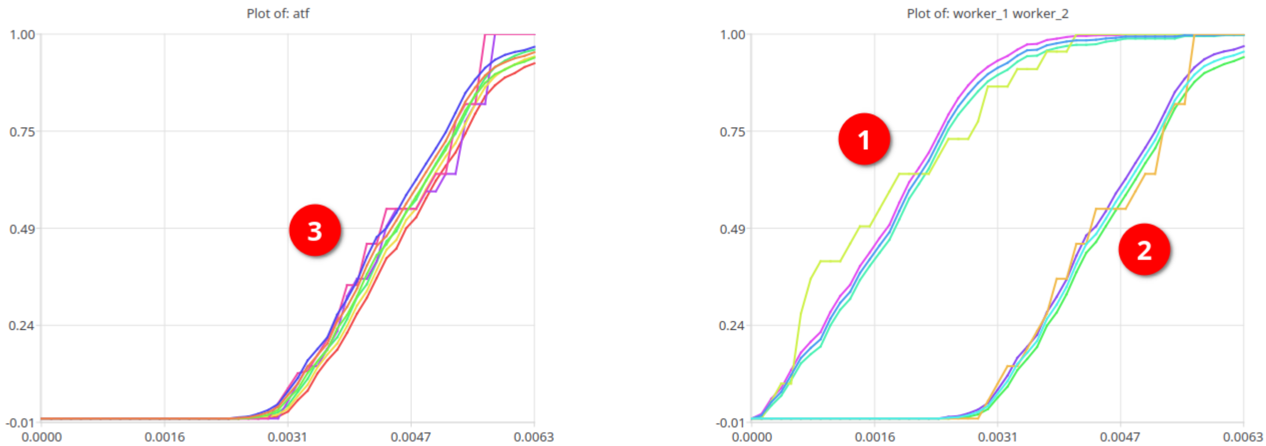

Figure 6.12: **Left**: ATF plot with observed and calculated ΔQ confidence bounds overlapping. **Right**: Worker's plots, worker_2 (2) is slower than worker_1 (1).

These plots show the usefulness of ΔQSD, the system can be decomposed to understand which part of the system is showing hazards thanks to the notion of outcome diagrams. Furthermore, the causal relationships can be observed to determine the behaviour of a part down to the single component.

Chapter 7

Performance study

This chapter evaluates the components and operations we introduced in previous sections, analysing their performances.

- We first evaluate the convolutions algorithms we previously introduced. Previously, stated that naïve convolution would have $\mathcal{O}(N^2)$ time complexity, while FFT convolution would have $\mathcal{O}(N \log N)$ complexity. We will evaluate both algorithms to see if what we observe corresponds to theory.
- We then evaluate the ΔQ adapter API performances, to see the overhead it introduces into a system.
- Lastly, we want to evaluate the QT framework plotting performances, we believe it is the weakest link in the oscilloscope and thus want to evaluate the QtCharts class when plotting ΔQs.

7.1 Convolution performance

We implemented two versions of the convolution algorithm as described before, the naïve version and the FFT version F.4.3. We compared their performance when performing convolution on two ΔQs of equal bins. In theory, we should observe the naïve version delay quickly grow, while the FFT version have a log-linear growth. We observe the result in Fig. 7.1. As expected, the naïve version has a time complexity of $\mathcal{O}(N^2)$ and quickly scales with the number of bins. This is clearly inefficient, as a more precise ΔQ will result in a much slower program.

As for the FFT algorithm, it is slightly slower when the number of bins is lower than 100. This is due to the FFTW3 routine having slightly higher overhead. Moreover, if we look closer at the FFT graph, we can observe slight increases after we surpass powers of 2 (e.g. at $600 > 512$, $300 > 256 \ldots$). This is because the algorithm is based on ΔQs which are zero-padded to the nearest power of 2, to make the calculation faster.

While we limit the number of bins to 1000 right now, this limit could potentially be increased as the FFT convolution algorithm is very efficient (0.5 ms for 1024 bins).

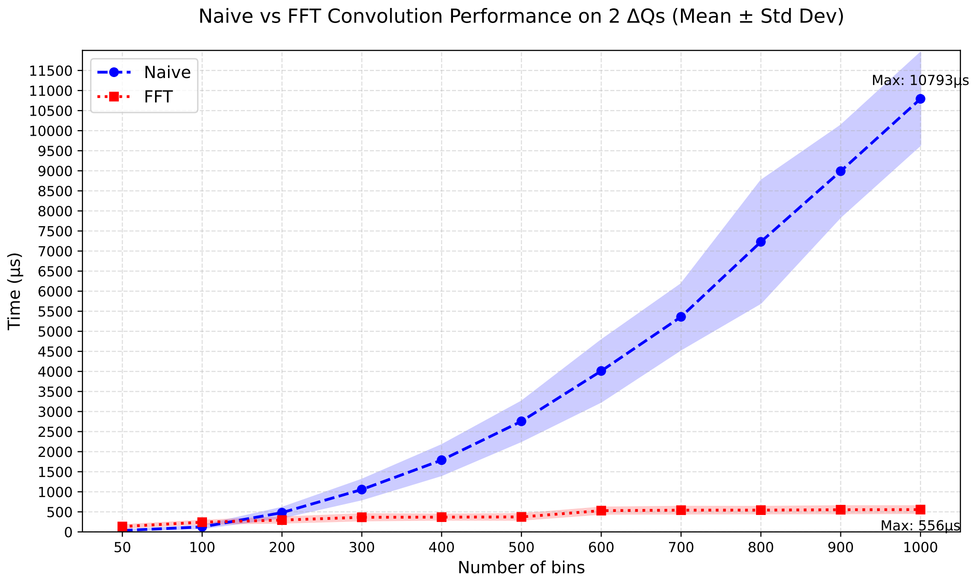

Figure 7.1: Performance comparison of FFT convolution and naïve convolution algorithms.

7.2 GUI plotting performance

We evaluated the performance of the GUI plotting routine. We evaluated the routine when plotting the sampling window observed ΔQ, the mean and confidence bounds of the polling window of observed ΔQs.

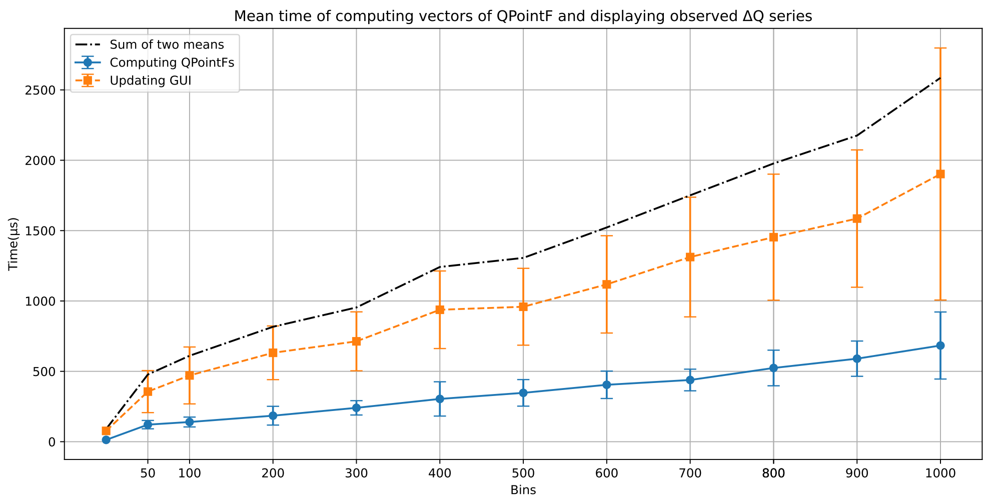

Figure 7.2: Performance of plotting sampling window and polling window observable ΔQ. **(Blue, circle)**: QPoinfF vectors setup performance. **(Orange, square)**: Plotting performance. **(Black, dotted)**: Sum of the previous two.

The procedure for preparing and plotting the observed and calculated ΔQ (along with the confidence bounds) is the same, so we would need to double the results we have to obtain the total time for the plot of a sub-outcome diagram. The routine first prepares vectors of QPointF [59], representing all the x and y values of the ΔQs CDF. The vectors are created for the lower bounds, the upper bounds, the mean of the window of ΔQs and the observed ΔQ.

Then, once the vectors are prepared, Qt replaces the old points of a series with the new points for every series being plotted.

The result scales up to almost 2.5 ms for 1000 bins. We believe that these performances are a big bottleneck of the oscilloscope. If we were to plot the calculated ΔQ and its confidence bounds, the time increase would be twofold. If the sampling rate was 100ms and many probes plots were being displayed, some frames would probably be skipped if the number of bins is high. The high results may nevertheless be explained by the specifications of the PC where we ran the tests, namely by the CPU and the GPU (Chapter A).

7.3 ΔQ adapter performance

We evaluated the performance of the adapter to measure its impact in a normal execution, namely we tested the following calls which represent a normal usage of the adapter.

- `start_span` → `end_span`.
- `with_span` with the following function: `fun() → ok.`
- `start_span` → `fail_span`.

We ran the simulation for 25000 subsequent iterations, the results are shown in Fig. 7.3:

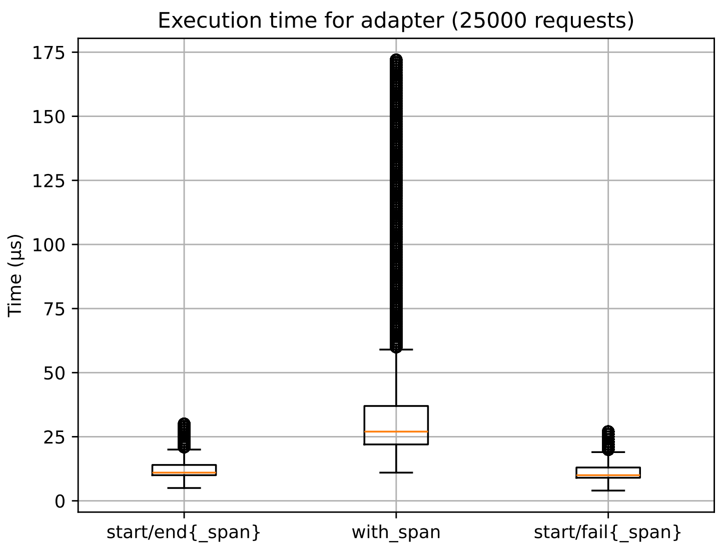

Figure 7.3: Adapter performance evaluation.

The overhead is minimal, around 10 microseconds on average to start and end/fail a span. The same cannot be said about with span, the increased overhead is nevertheless due to a function needing to be called inside it for it to record a span.

Chapter 8

Conclusions and future work

As we introduced the thesis, its background and current problems we think exist in observability tools, we set out a clear goal: design a graphical dashboard, the **ΔQ oscilloscope**, to observe running Erlang applications and plot the system's behaviour in real time thanks to the ΔQSD paradigm. While we can not say that we are fully finished with the development of the oscilloscope, we can clearly say that a **working prototype** that reflects the theoretical findings of the paradigm and fulfills the initial goals was created.
This was successfully achieved thanks to multiple important implementations which make it fast, reliable and moreover capable of accurately detecting deviations from required behaviour.

The ΔQ adapter, named `dqsd_otel`, an Erlang interface which is able to work alongside OpenTelemetry to add the notion of failure according to ΔQSD to spans. The adapter can communicate data about outcome instances from a running system directly to the oscilloscope and can directly receive commands from the oscilloscope. The `gen_server` behaviour of the adapter client and server allows for this to be done fast and asynchronously.

The ΔQ oscilloscope, a fully fledged Qt dashboard application that is able to observe running systems and provide real time plotting capabilities to the user. Moreover, it provides full control to the user of the outcome diagrams, the parameters of probes, their QTAs, the triggers they want to set for a given probe, the snapshots to observe the system as if it was frozen in time. In fine, the FFT convolution algorithms allows us to scale down from $\mathcal{O}(N^2)$ complexity to $\mathcal{O}(N \log N)$, bringing the time to provide precise insights significantly down.

The synthetic applications further prove the oscilloscope's usefulness in detecting early signs of overload and dependent behaviour. This reinforces the solid theoretical basis of ΔQSD, which we remind has already been applied in many industrial projects.

Many crucial features are still missing from the dashboard, and it could require less code modifications in the Erlang side. The next important step of the oscilloscope is its trial in a true distributed application.

8.1 Future improvements

We list here some improvements which could be made to both the oscilloscope and the adapter.

8.1.1 Oscilloscope improvements

- The oscilloscope could be turned into a **web app**, we feel that a C++ oscilloscope is a good prototype and proof of concept, but its usability would be greater in a browser context. It would be great as a plugin for already existing observability platforms like Grafana.
- A wider selection of **triggers**, as of writing this thesis, only the QTA trigger and load are available, this is a limitation due to time constraints. Nevertheless, triggers can be easily implemented in the available codebase.
- **Better communication between stub - server - oscilloscope**. The current way of sending outcome instances may be a limiting factor under high load, if hundred of thousands of spans were to be sent, the current way the server and oscilloscope are tied together may throttle communications. TCP socket connections could quickly become the chokepoint which makes the oscilloscope temporarily unusable.

  Future improvements on the server side could implement epoll system server calls to make the server more efficient; **Detaching server from client**, as of right now, the oscilloscope and the server are tied together, using ZeroMQ to assure real time server-client communications could be an interesting solution to explore.
- **Improve real time graphs**. The class QtCharts does not perform correctly with high frequencies update. Moreover, since we are plotting multiple series (from a minimum 4 to a maximum of 9) per probe, which allows up to 1000 bins per probe, the performance quickly degrades with more probes being displayed. A better graphing class for Qt could definitely improve the experience.
- **Saving probe parameters**: As of writing this thesis, there is no way to save the parameters one may have set.
- **Deconvolution**: An important aspect of ΔQSD, which was not introduced in this paper is deconvolution. It is used to check for infeasability in system desing. Since convolution has already been implemented, this could be integrated using the FFTW3 library.
- **Exporting graphs**: The graphs can only be observed in the oscilloscope and have no way to be exported to other programs via standard formats.
- **Many more**: This oscilloscope is just a start and part of the dissemination project of ΔQSD. If we were to list everything we may want to add, it would take many pages. What we provide is a sufficient enough basis to provide possibilities to observe a running system and understand the power of ΔQSD in analysing its behaviour.

8.1.2 OpenTelemetry improvements

As we explored in a previous chapter (Section 2.3.1), long-lived spans are currently a problem. ΔQSD requires that it must be possible to consider a span as failed if it takes longer than the maximum delay required by the user. This delay can be set by the user and depends on the application's required behaviour (Section 4.1.2). This is not trivial to do within the current implementation of the OpenTelemetry standard in Erlang. We believe that what we have shown in this thesis should be the standard to improve observability requirements to capture essential timeliness requirements. This could be done via events that are triggered when the delay surpasses the required one and are directly exported to a monitoring tool. We are nevertheless aware that events are currently unstable in the current version of OpenTelemetry in Erlang.

8.1.3 Adapter improvement

As suggested by Bryan Naegele, a member of the observability group of Erlang, the adapter, instead of calling OpenTelemetry macros inside the interface, could directly use the OpenTelemetry span processor. Leveraging `erlang:send_after` as we already do, we could create outcome instances with telemetry [60] events (which must not be mistaken with OpenTelemetry ones) to handle successful executions and timeouts. The span processor will then be responsible for creating outcome instances, without creating the need for calling OpenTelemetry in the adapter, like we do now.

8.1.4 Real applications

The current prototype of the ΔQ oscilloscope has not been tested on real distributed applications. While their usefulness has been proven on synthetic applications, the lack of real life applications is a weakness.

8.1.5 Licensing limitations

Lastly, a notable limitation is created by **Qt**, namely, QtCharts. QtCharts has a GPLv3 license. Therefore, the usage of Qt does not allow us to release our project under BSD/MIT licenses or LGPL license, but rather a GPLv3 one. [61]

Bibliography

[1] Solarwinds Loggly. *Distributed Systems Monitoring: The Essential Guide*. Accessed: (31/05/2025). URL: https://www.loggly.com/use-cases/distributed-systems-monitoring-the-essential-guide/.

[2] Aaron Abbott. *Sunsetting OpenCensus*. Accessed: (31/05/2025). 2023. URL: https://opentelemetry.io/blog/2023/sunsetting-opencensus/.

[3] Juraci Paixao Krohling David Allen. *OpenTelemetry and vendor neutrality*. Accessed: (31/05/2025). 2024. URL: https://grafana.com/blog/2024/09/12/opentelemetry-and-vendor-neutrality-how-to-build-an-observability-strategy-with-maximum-flexibility/.

[4] OpenTelemetry Authors. *Vendors who natively support OpenTelemetry*. Accessed: (31/05/2025). 2025. URL: https://opentelemetry.io/ecosystem/vendors/.

[5] The Zipkin Authors. *Zipkin*. Accessed: (01/06/2025). 2025. URL: https://zipkin.io/.

[6] OpenTelemetry Authors. *OpenTelemetry - Traces*. Accessed: (19/05/2025). 2025. URL: https://opentelemetry.io/docs/concepts/signals/traces/.

[7] Magsther. *Awesome OpenTelemetry*. Accessed: (31/05/2025). 2025. URL: https://github.com/magsther/awesome-opentelemetry?tab=readme-ov-file.

[8] The Jaeger Authors. *Service Performance Monitoring (SPM)*. Accessed: (31/05/2025). 2025. URL: Service%20Performance%20Monitoring%20(SPM).

[9] Peter Van Roy and Seyed Hossein Haeri. *The ΔQSD Paradigm: Designing Systems with Predictable Performance at High Load. Full-day tutorial*. 15th ACM/SPEC International Conference on Performance Engineering. 2024. URL: https://webperso.info.ucl.ac.be/~pvr/ICPE-2024-deltaQSD-full.pdf.

[10] *PNSol: the network performance science company*. Accessed: (31/05/2025). 2025. URL: https://www.pnsol.com/.

[11] Seyed H. Haeri et al. "Mind Your Outcomes: The ΔQSD Paradigm for Quality-Centric Systems Development and Its Application to a Blockchain Case Study". In: *Comput.* 11.3 (2022), p. 45. DOI: 10.3390/COMPUTERS11030045. URL: https://doi.org/10.3390/computers11030045.

[12] Peter Thompson and Rudy Hernandez. *Quality Attenuation Measurement Architecture and Requirements*. Tech. rep. MSU-CSE-06-2. Sept. 2020. URL: https://www.broadband-forum.org/pdfs/tr-452.1-1-0-0.pdf.

[13] J. H. Nyström, P. W. Trinder, and D. J. King. "Evaluating distributed functional languages for telecommunications software". In: *Proceedings of the 2003 ACM SIGPLAN Workshop on Erlang*. ERLANG '03. Uppsala, Sweden: Association for

Computing Machinery, 2003, pp. 1–7. ISBN: 1581137729. DOI: 10.1145/940880.940881. URL: https://doi.org/10.1145/940880.940881.

[14] Erlang/OTP. *What is Erlang*. Accessed: (26/05/2025). 2025. URL: https://www.erlang.org/faq/introduction.html.

[15] Francesco Nieri. *UCLouvain Master thesis poster presentation - ΔQ oscilloscope*. 2024.

[16] Seyed Hossein Haeri et al. "Algebraic Reasoning About Timeliness". In: *Proceedings 16th Interaction and Concurrency Experience, ICE 2023, Lisbon, Portugal, 19th June 2023*. Ed. by Clément Aubert et al. Vol. 383. EPTCS. 2023, pp. 35–54. DOI: 10.4204/EPTCS.383.3. URL: https://doi.org/10.4204/EPTCS.383.3.

[17] Peter Van Roy. *LINFO2345 lessons on ΔQSD*. Accessed: (19/05/2025). UCLouvain, 2023. URL: https://www.youtube.com/watch?v=tF7fbU9Gce8.

[18] Neil J. Davies and Peter W. Thompson. *ΔQSD workbench - GitHub*. Accessed: (19/05/2025). 2022. URL: https://github.com/DeltaQ-SD/dqsd-workbench.

[19] Erlang programming language. *Erlang tracing*. Accessed: (19/05/2025). 2024. URL: https://www.erlang.org/doc/apps/erts/tracing.html.

[20] OpenTelemetry Authors. *OpenTelemetry in Erlang/Elixir*. Accessed: (19/05/2025). 2025. URL: https://opentelemetry.io/docs/languages/erlang/.

[21] Isaak D. Mayergoyz and W. Lawson (Auth.) *Basic Electric Circuit Theory. A One-Semester Text*. pg. 116. Academic Press, 1996. ISBN: 9780080572284; 0080572286; 9780124808652; 0124808654.

[22] OpenTelemetry Authors. *What is OpenTelemetry?* Accessed: (19/05/2025). 2025. URL: https://opentelemetry.io/docs/what-is-opentelemetry/.

[23] Erlang Programming Language. *Erlang - trace*. Accessed: (01/06/2025). 2025. URL: https://www.erlang.org/doc/apps/kernel/trace.html.

[24] OpenTelemetry Authors. *Language API & SDKs*. Accessed: (28/05/2025). 2025. URL: https://opentelemetry.io/docs/languages/.

[25] OpenTelemetry Authors. *Instrumentation for OpenTelemetry Erlang/Elixir*. Accessed: (19/05/2025). 2025. URL: https://opentelemetry.io/docs/languages/erlang/instrumentation/.

[26] Erlang Ecosystem Foundation. *Observability Working Group*. Accessed: (28/05/2025). 2025. URL: https://erlef.org/wg/observability.

[27] OpenTelemetry Authors. *Observability primer - Distributed traces*. Accessed: (28/05/2025). 2024. URL: https://opentelemetry.io/docs/concepts/observability-primer/.

[28] OpenTelemetry Authors. *Exporters*. Accessed: (28/05/2025). 2025. URL: https://opentelemetry.io/docs/languages/erlang/exporters/.

[29] The Jaeger Authors. *Jaeger*. Accessed: (19/05/2025). 2025. URL: https://www.jaegertracing.io/.

[30] Datadog. *Datadog - Product*. Accessed: (28/05/2025). 2025. URL: https://www.datadoghq.com/product/.

[31] Dotan Horovits. *From Distributed Tracing to APM: Taking OpenTelemetry & Jaeger Up a Level*. Accessed: (19/05/2025). 2021. URL: https://logz.io/blog/monitoring-microservices-opentelemetry-jaeger/.

[32] Sampath Siva Kumar Boddeti. *Tracing Made Easy: A Beginner's Guide to Jaeger and Distributed Systems*. Accessed: (19/05/2025). 2024. URL: https://

openobserve.ai/blog/tracing-made-easy-a-beginners-guide-to-jaeger-and-distributed-systems/.

[33] OpenTelemetry Authors. *Active spans, C++ Instrumentation*. Accessed: (19/05/2025). 2025. URL: https://opentelemetry.io/docs/languages/cpp/instrumentation/.

[34] Hazel Weakly. *OpenTelemetry Challenges: Handling Long-Running Spans*. Accessed: (21/05/2025). 2024. URL: https://thenewstack.io/opentelemetry-challenges-handling-long-running-spans/.

[35] Inc. Embrace Mobile. *Embrace*. Accessed: (31/05/2025). 2025. URL: https://embrace.io/.

[36] OpenTelemetry authors. *Error handling*. Accessed: (29/05/2025). 2022. URL: https://opentelemetry.io/docs/specs/otel/error-handling/.

[37] Francesco Nieri. *dqsd_otel*. Accessed: (28/05/2025). 2025. URL: https://github.com/fnieri/dqsd_otel.

[38] Francesco Nieri. *ΔQOscilloscope*. Accessed: (28/05/2025). URL: https://github.com/fnieri/DeltaQOscilloscope.

[39] KeySight. *What is an Oscilloscope Trigger?* Accessed: (23/05/2025). 2022. URL: https://www.keysight.com/used/id/en/knowledge/glossary/oscilloscopes/what-is-an-oscilloscope-trigger.

[40] Alex Tsun. *Statistical inference - Confidence intervals*. Accessed: (29/05/2025). 2020. URL: https://courses.cs.washington.edu/courses/cse312/20su/files/student_drive/8.1.pdf.

[41] FM Dekking et al. *A modern introduction to probability theory and statistics*. Vol. Springer texts in statistics. Springer, 2005. ISBN: 1-85233-896-2.

[42] Brian Pauw. *What does rebinning do?* Accessed: (31/05/2025). 2015. URL: https://lookingatnothing.com/index.php/archives/2097.

[43] Steven B. Damelin and Willard Miller Jr. *The Mathematics of Signal Processing*. USA: Cambridge University Press, 2012. ISBN: 1107601045.

[44] FFTW3. *Fastest Fourier Transform in The West*. Accessed: (19/05/2025). 2025. URL: https://www.fftw.org/.

[45] Jeremy Fix. *FFTConvolution*. Accessed: (21/05/2025). 2013. URL: https://github.com/jeremyfix/FFTConvolution/blob/master/Convolution/src/convolution_fftw.h.

[46] *Planning-rigor flags*. Accessed: (23/05/2025). URL: https://www.fftw.org/doc/Planner-Flags.html.

[47] K. R. Rao, D. N. Kim, and J.-J. Hwang. *Fast Fourier Transform - Algorithms and Applications*. 1st. Springer Publishing Company, Incorporated, 2010. ISBN: 1402066287.

[48] Rebar3. *Rebar3 Basic Usage*. Accessed: (25/05/2025). 2025. URL: https://rebar3.org/docs/basic_usage/.

[49] Erlang. *Binary module*. Accessed: (30/05/2025). URL: https://www.erlang.org/docs/24/man/binary.

[50] Erlang System Documentation. *Bit syntax*. Accessed: (30/05/2025). 2025. URL: https://www.erlang.org/doc/system/bit_syntax.html.

[51] Erlang System Documentation. *Signals/Sending signals*. Accessed: (24/05/2025). 2025. URL: https://www.erlang.org/doc/system/ref_man_processes.

[52] Erlang Reference Manual. *gen_server*. Accessed: (01/06/2025). 2024. URL: https://www.erlang.org/docs/24/man/gen_server.

[53] ANTLR. *What is ANTLR4?* Accessed: (19/05/2025). 2025. URL: https://www.antlr.org/.

[54] skenz.it. *Flex Bison*. Accessed: (31/05/2025). 2024. URL: https://www.skenz.it/compilers/flex_bison.

[55] Thomas Niemann. *Lex/Yacc*. Accessed: (31/05/2025). URL: https://arcb.csc.ncsu.edu/~mueller/codeopt/codeopt00/y_man.pdf.

[56] Terence Parr and Kathleen Fisher. "LL(*): the foundation of the ANTLR parser generator". In: *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*. Ed. by Mary W. Hall and David A. Padua. ACM, 2011, pp. 425–436. DOI: 10.1145/1993498.1993548. URL: https://doi.org/10.1145/1993498.1993548.

[57] lark-parser. *Lark - A parsing toolkit for Python*. Accessed: (24/05/2025). 2025. URL: https://github.com/lark-parser/lark.

[58] Wikipedia. *Qt (software) Wikipedia, The Free Encyclopedia*. Accessed: (24/05/2025). 2025. URL: https://en.wikipedia.org/wiki/Qt_(software).

[59] The Qt Company. *QPointF class*. Accessed: (01/06/2025). 2025. URL: https://doc.qt.io/qt-6/qpointf.html.

[60] ExDoc for the Erlang programming language. *telemetry*. Accessed: (01/06/2025). 2025. URL: https://hexdocs.pm/telemetry/telemetry.html.

[61] The Qt Company. *Add-ons available under Commercial Licenses, or GNU General Public License v3*. Accessed: (23/05/2025). 2025. URL: https://doc.qt.io/qt-5/qtmodules.html.

Appendix A

Tests specifications

The tests were run on a laptop with the following specifications.

- `OS`: Manjaro Linux x86_64, `Kernel:` 6.12.11-1-MANJARO
- `CPU`: Intel i5-6300U (4) @ 3.000GHz
- `GPU`: Intel Skylake GT2 [HD Graphics 520]
- `RAM`: 16 GB DDR4 SDRAM—2133MHz

Part I

Addendum to Chapters 4 & 5

Appendix B

Oscilloscope screenshots

B.1 System/Handle plots tab

Here is a screenshot from the tab to create the system and handle plots (Fig. B.1).

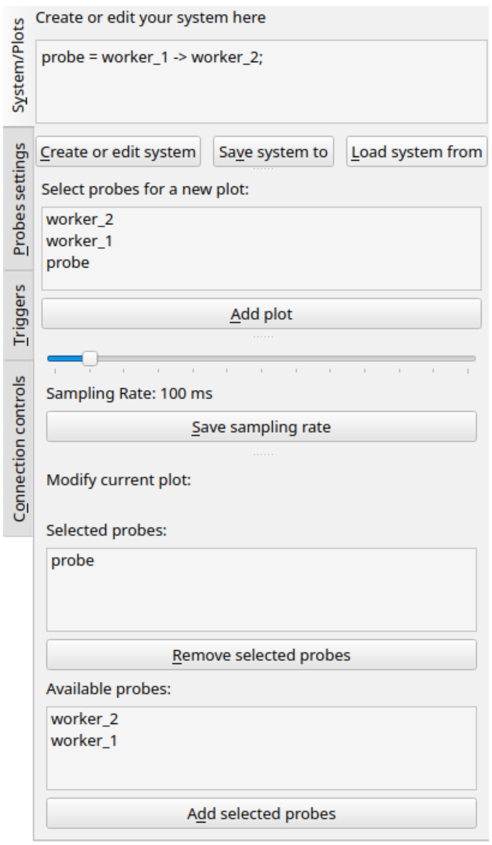

Figure B.1: System/Handle plots tab in oscilloscope.

B.2 Parameters tab

Here is a screenshot of the parameters tab, with a set QTA showing up on a plot (Fig. B.2).

Figure B.2: Parameters tab in oscilloscope. The QTA set for worker_1 can be seen on the plot.

B.3 Triggers tab

Here is a screenshot of the triggers tab, with saved snapshots, QTA violation set for probe and a running plot (Fig. B.3).

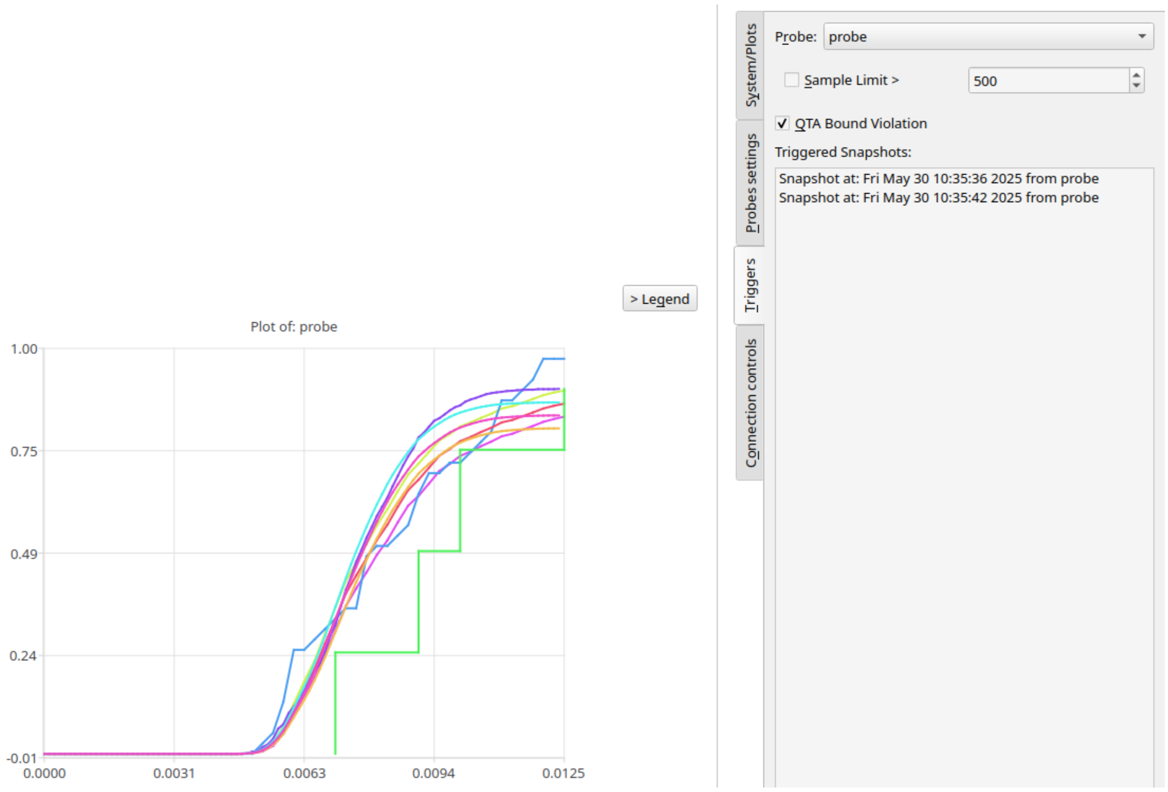

Figure B.3: Triggers tab in oscilloscope. The snapshots for the whole system are saved and can be observed.

B.4 Snapshot window

Here is a screenshot from the window observing a snapshot (Fig. B.5).

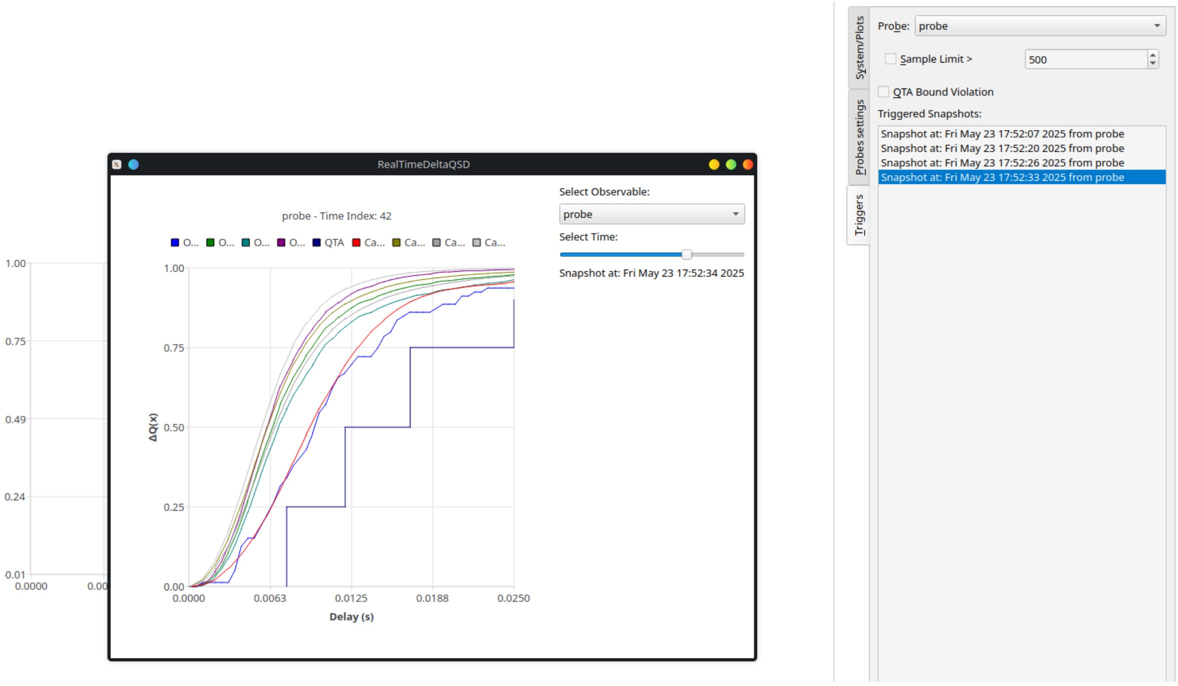

Figure B.4: Snapshot window (above) over the running oscilloscope below.

B.5 Connection controls

Here is a screenshot of the available controls in the oscilloscope (Fig. B.5).

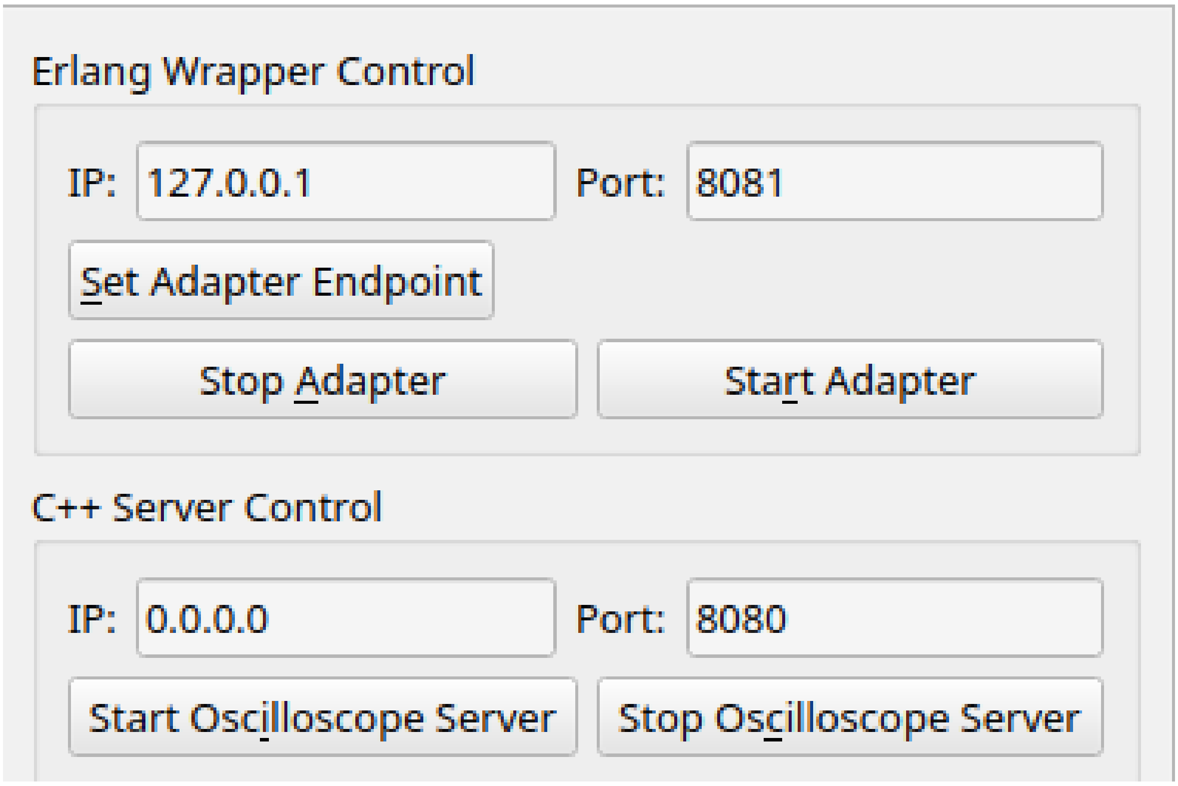

Figure B.5: Connections control tab in the oscilloscope with the erlang and oscilloscope endpoints.

B.6 Widget view of GUI

The GUI is composed of multiple building block, the widgets. The screenshot we provide here highlights the multiple widgets present in the GUI (Fig. B.6). These widgets could be broken down into the multiple subwidgets that compose them.

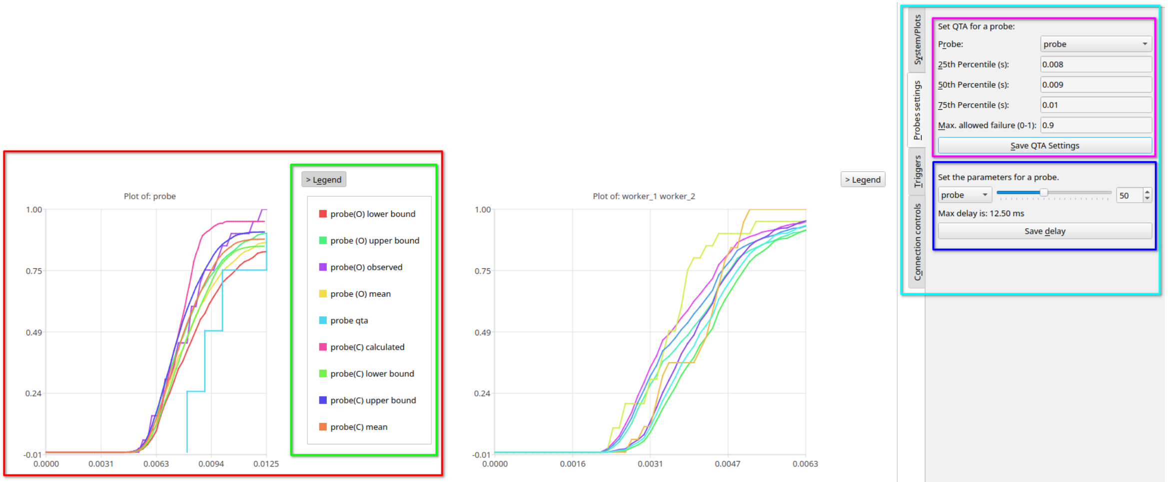

Figure B.6: Widget view of the oscilloscope dashboard.

Part II

User manual

Appendix C

How to download and launch

To download the oscilloscope, go to the GitHub repository (`https://github.com/fnieri/DeltaQOscilloscope/`) and go to the releases page, there, you will find the different versions of the oscilloscope. For instruction purposes, we have made a pre-release to show where to find it.

Figure C.1: Releases tab.

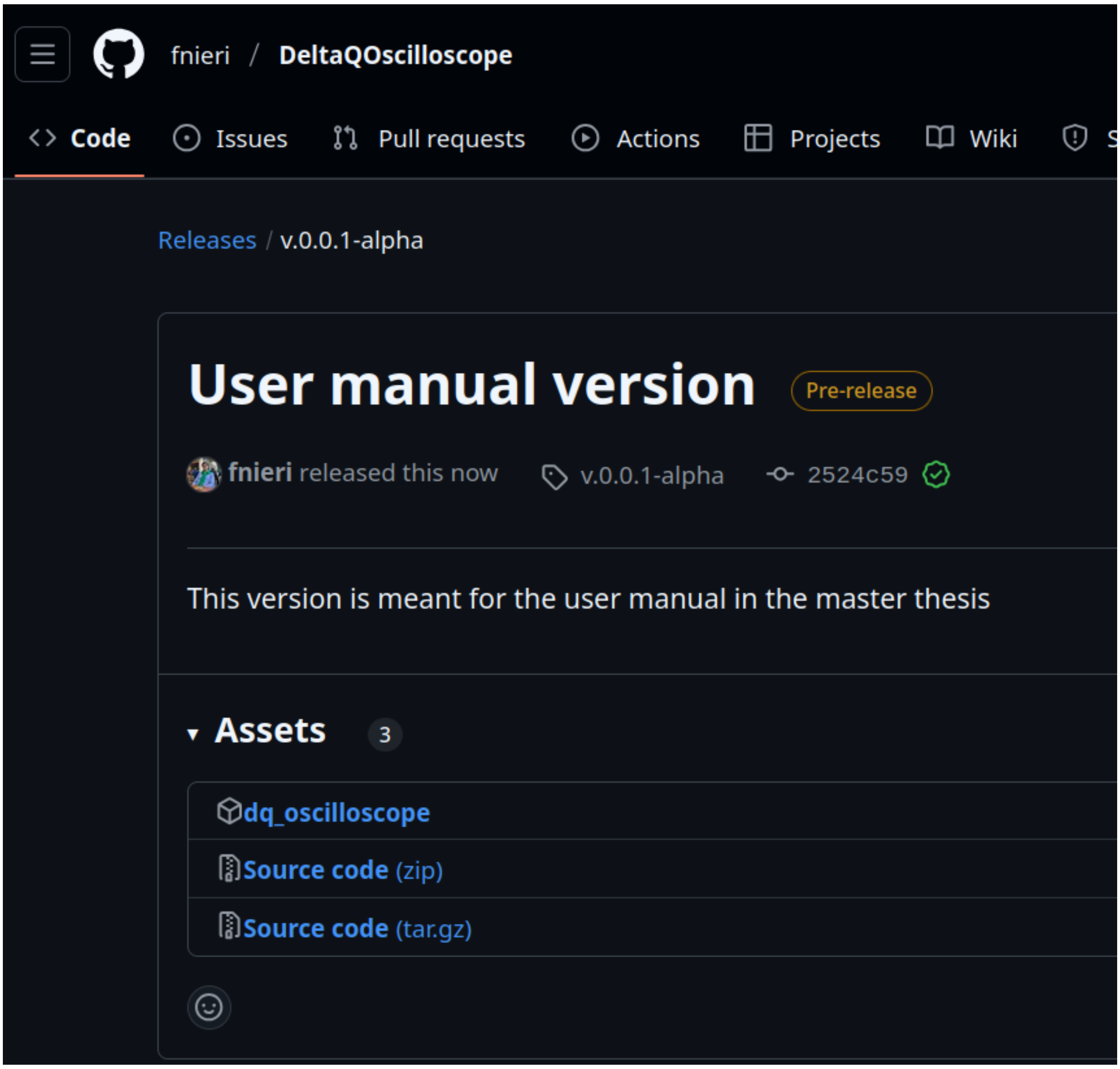

Figure C.2: Sample release

The dq_oscilloscope binary is the binary to download. By launching the binary on the CLI with:

```
./dq_oscilloscope
```

The oscilliscope will launch.

Appendix D

Oscilloscope: How to use

D.1 Establishing the adapter - oscilloscope connection

To connect to the oscilloscope to the server, you first need to start the oscilloscope server by setting the oscilloscope listening IP and port on the dashboard (Fig. D.1).

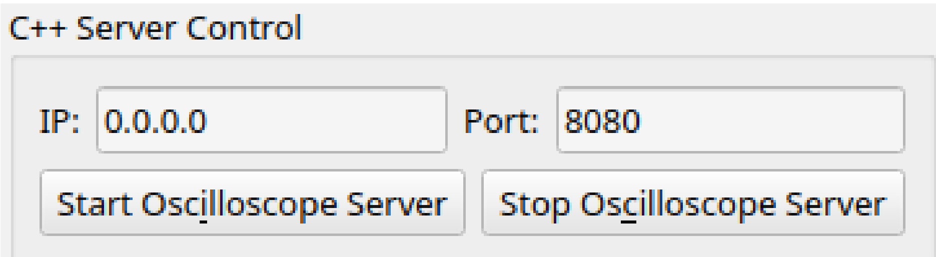

Figure D.1: Widget to set server endpoint in the oscilloscope

If the server cannot start, a popup will appear with the error.

Once this is done, the adapter can establish a connection to the oscilloscope to send outcome instances (Fig. D.2). The adapter can now connect to the oscilloscope server.

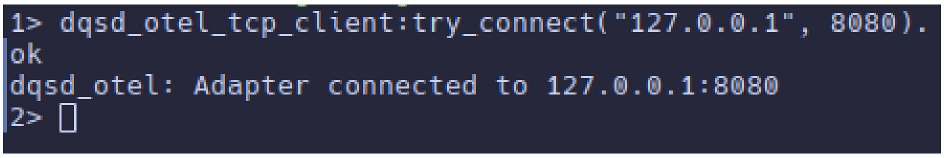

Figure D.2: Establish connection from adapter to oscilloscope.

We now need to start the listener on the adapter (Fig. D.3).

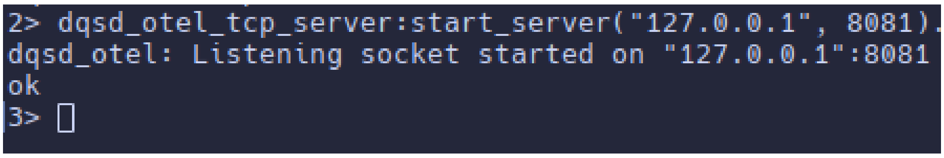

Figure D.3: Adapter starting listener for commands and parameters from the oscilloscope
.

If an error may arrive during the start-up of the listener, it will be printed out.

Now, we can connect the oscilloscope to the adapter by setting the listener endpoint (Fig. D.4).

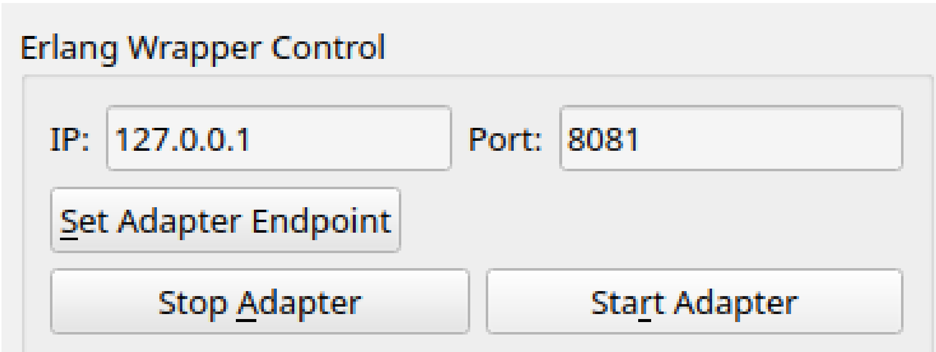

Figure D.4: Set Erlang listener endpoint in oscilloscope.

If the server cannot connect, an error will pop up. Once connected, the server can start and stop the adapter sending of the spans by clicking the two buttons below.

D.2 Sidebar: Outcome diagram and plots

D.2.1 Creating the system/outcome diagrams

You need to provide a textual description of your outcome diagram following the syntax which was defined previously (Section 4.3). You can create your system by clicking on the **Create or edit system** button (Fig. D.5). If the parser successfully parsed the text, your system will be created, and you can start setting the parameters for your probes.

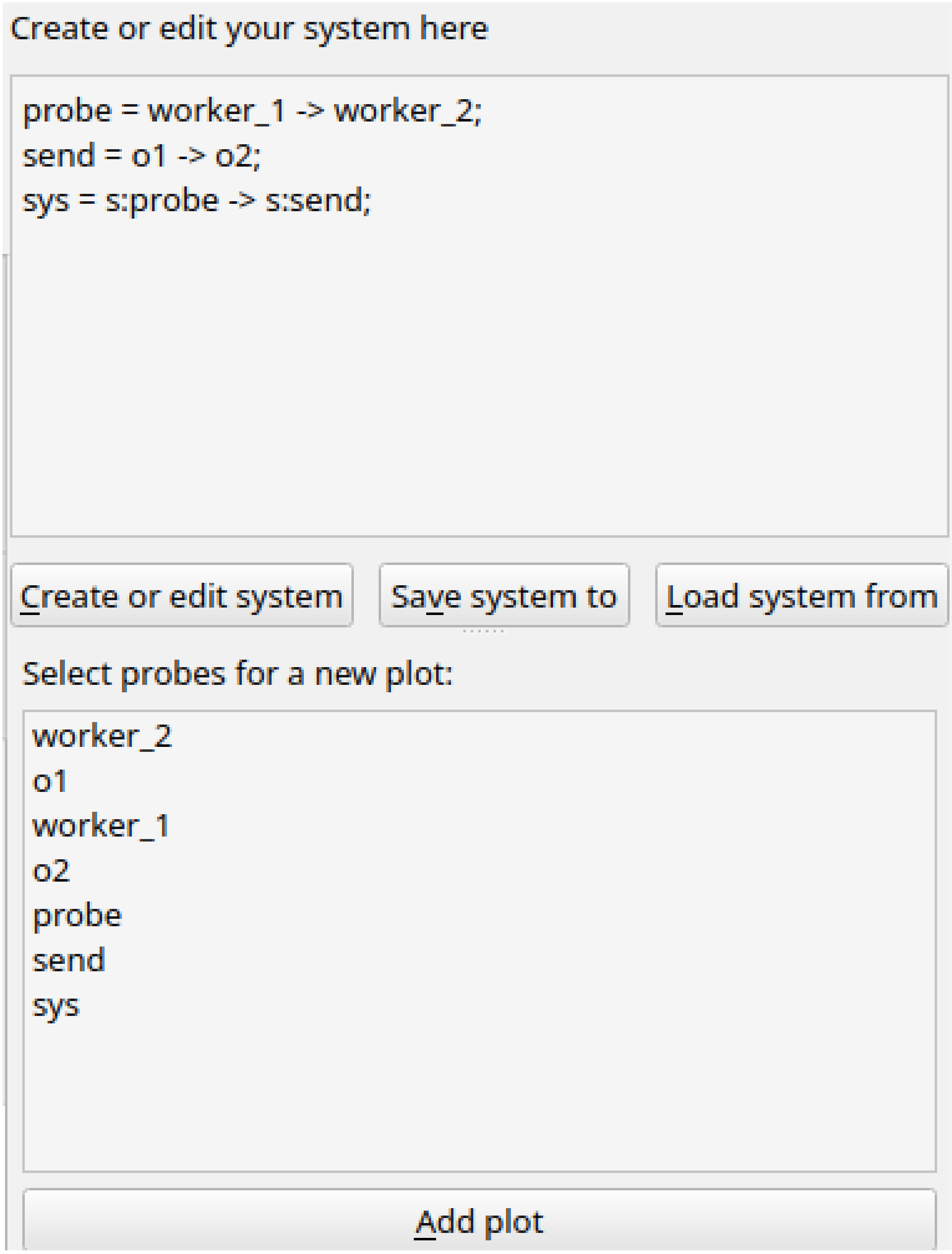

Figure D.5: After selecting create system, if the system definition is correct, the inserted probes will be available to be plotted.

If the parser does not correctly parse the text, it will show a popup indicating the line where the error was produced and what it was expecting (Fig. D.6). If there is a cycle, it will specify where it found the cycle.

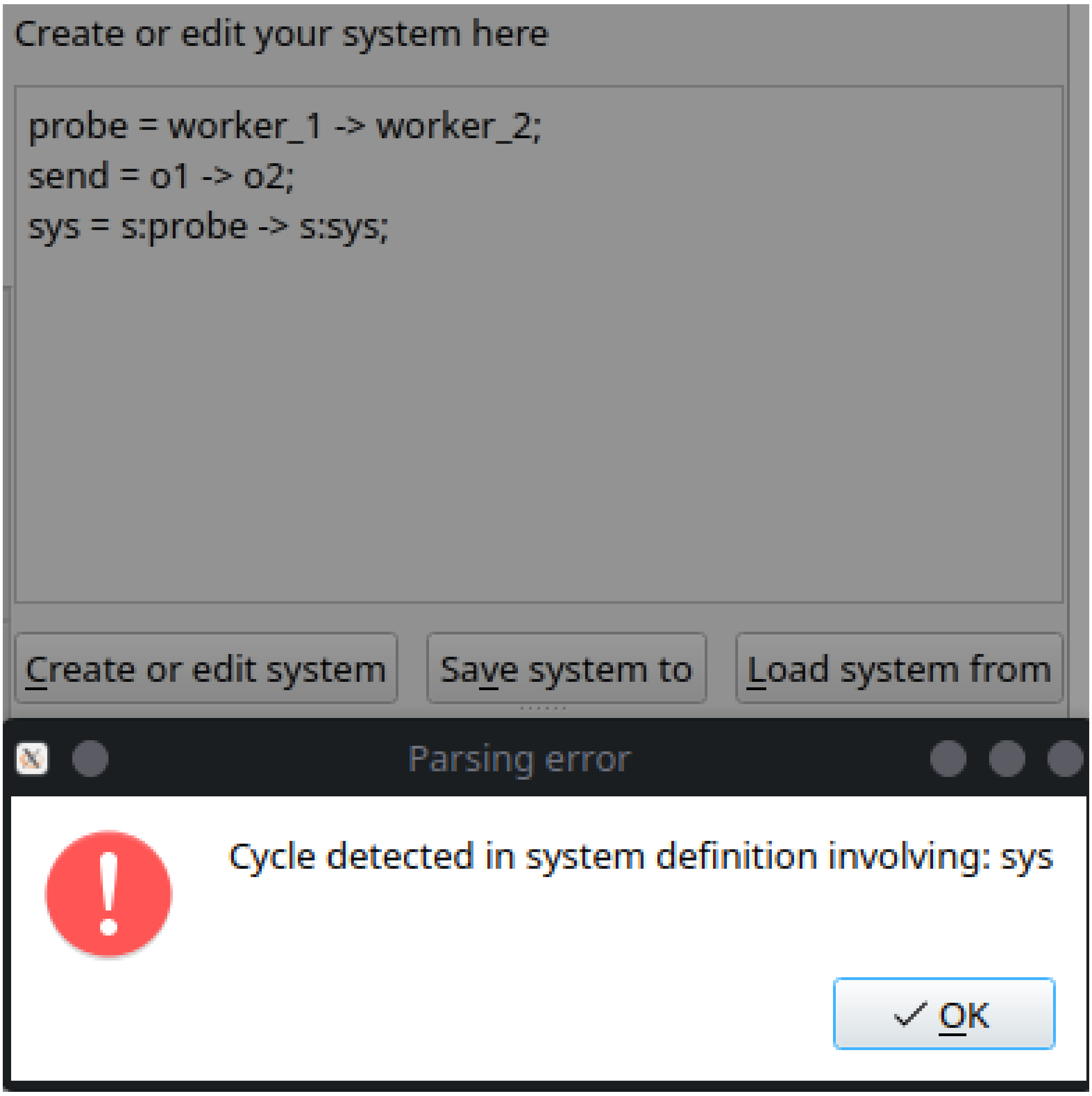

Figure D.6: Creation of a system with a cycle

D.2.2 Saving the system definition

The button **Save system to** gives you the possibility to save the textual definition of the outcome diagram to a file. You can save the file to any extension, but preferably the file will have the **.dq** extension (Fig. D.7).

D.2.3 Loading the system definition

The button **Load system to** gives you the possibility to load the textual definition of the outcome diagram you may have previously saved to a file (Fig. D.7). As for the extension, you can load any file extension.

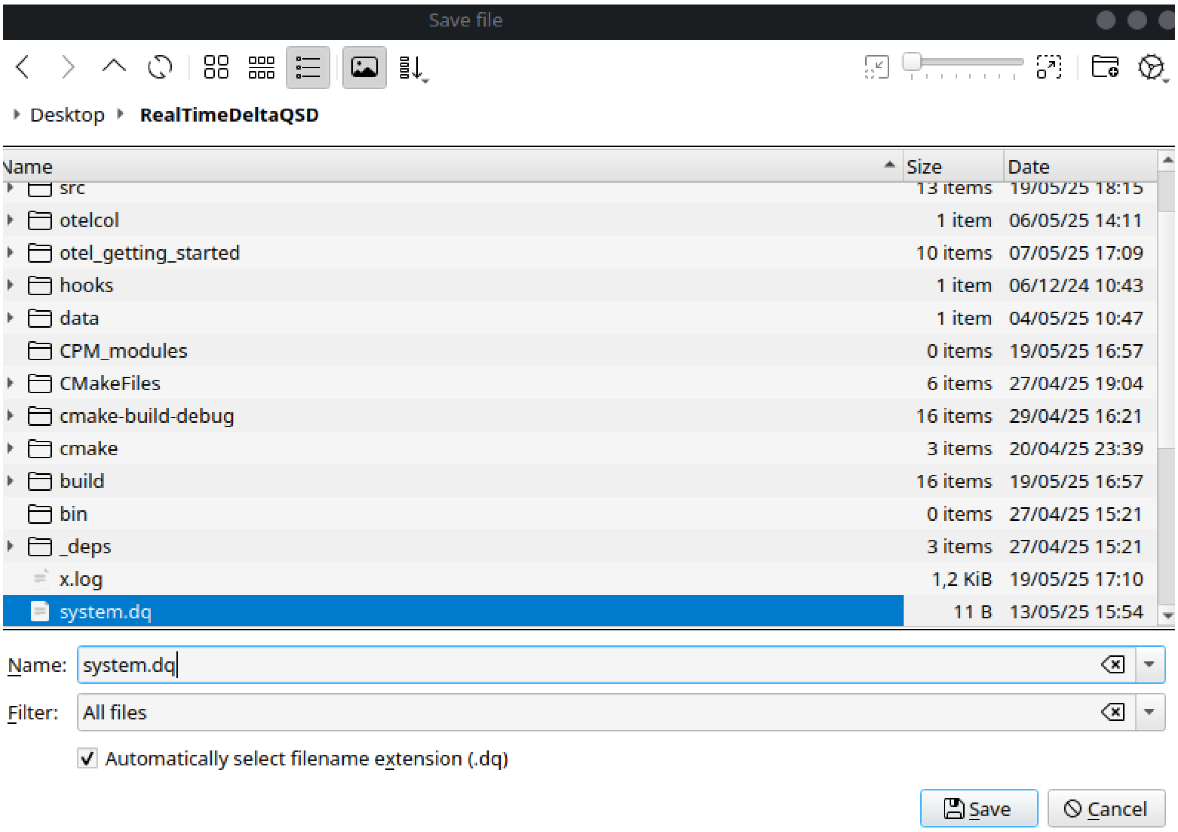

Figure D.7: Example popup where the user can select a file where the system definition is loaded.

D.2.4 Changing the sampling rate

The user can set the sampling rate via the slider in this tab (Fig. D.8):

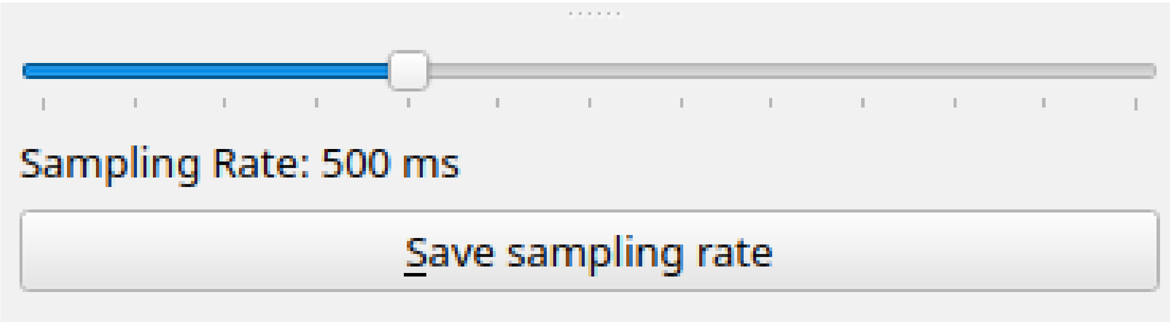

Figure D.8: Slider widget.

D.2.5 Managing the plots

Once you have your system defined you can start adding plots of the ΔQs of the probes you inserted in your system.

Adding a plot

Multiple probes can be added at once in your plot, you can select the probes you want to add to a new plot by selecting them in the **“Add a new plot”** section (Fig. D.9). Then by clicking the **“Add plot”** button, the selected probes will be added to the plot (Fig. D.10).

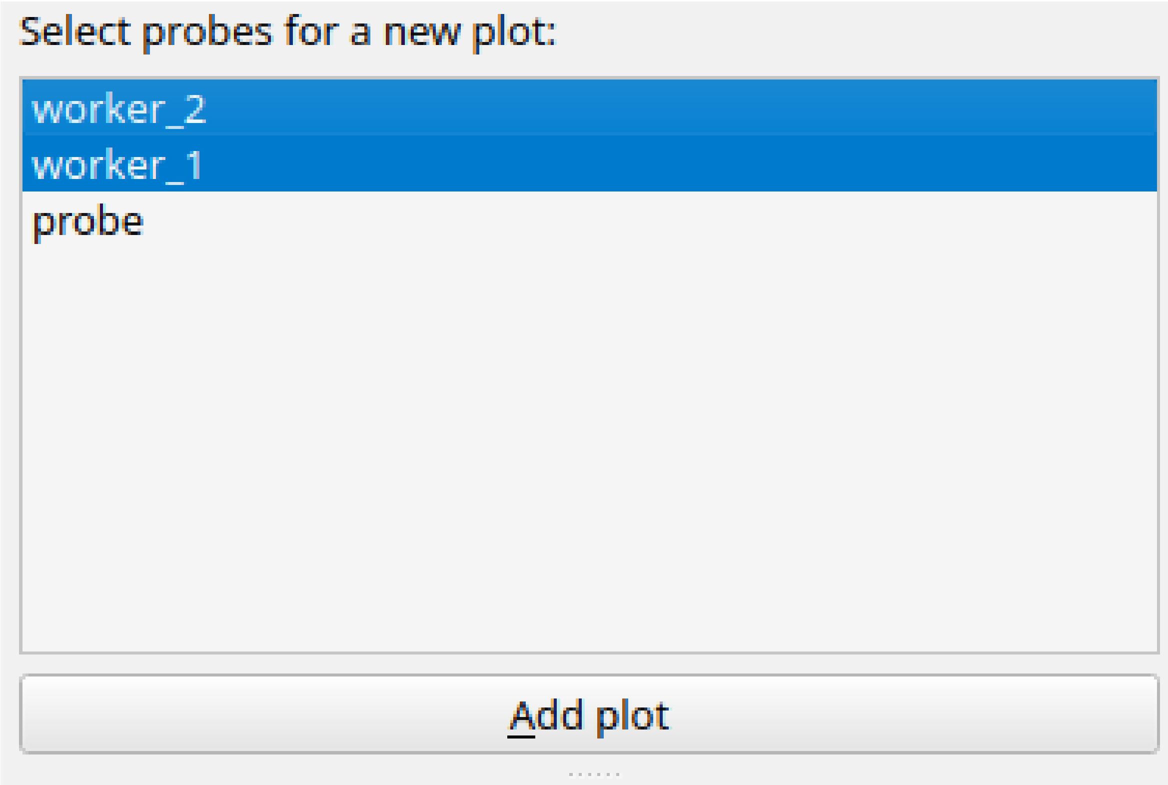

Figure D.9: Selecting the probes to add to a plot

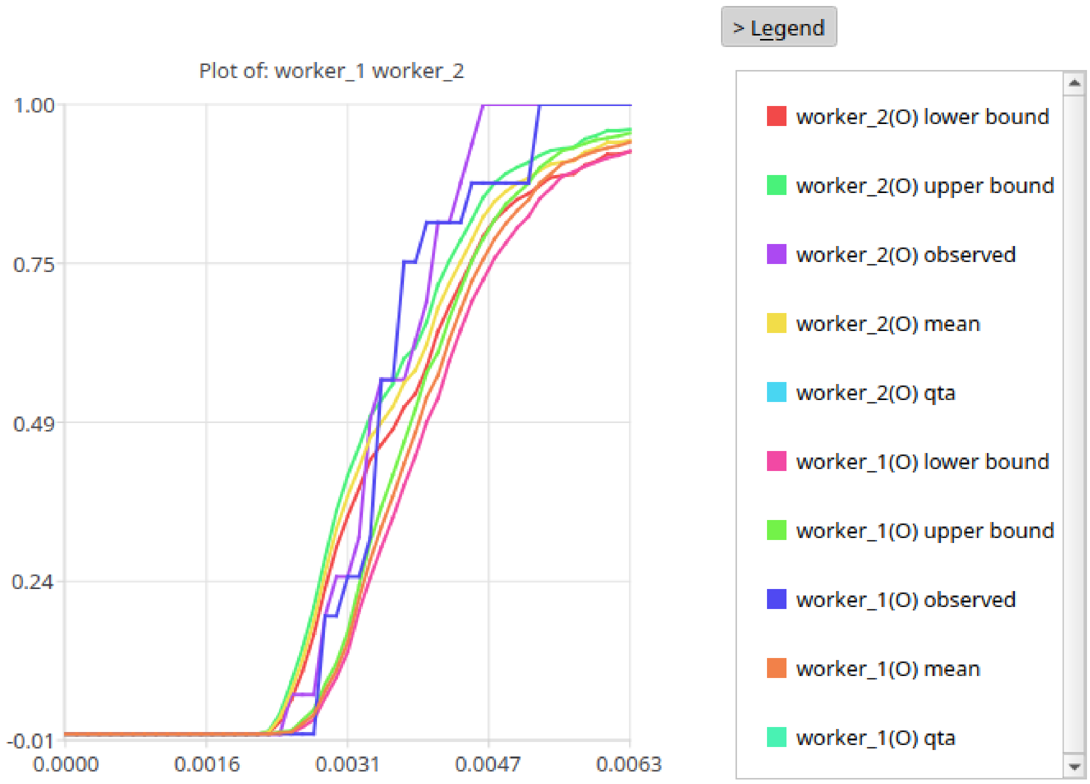

Figure D.10: After adding the probe's to a plot, if the adapter is already sending spans, the plot will show the real time ΔQs.

Editing a plot

You can remove the probes you have added to the plot by first clicking to it, this sets the plot as the **selected plot**. Once you have clicked the plot, a section will pop up beneath the rest of the controls on the sidebar.

In the section there are two subsections, one which shows the selected components which form the plot (those you have added previously), and the available components. You can select the probes you want to remove in the **"selected probes"** zone, by clicking **"Remove selected probes"** the components will be removed from the plot. Inversely, in the **"Available probes"** section, you can select the plots you want to add, and by clicking **"Add selected probes"** you can add the selected components to the selected plot (Fig. D.11).

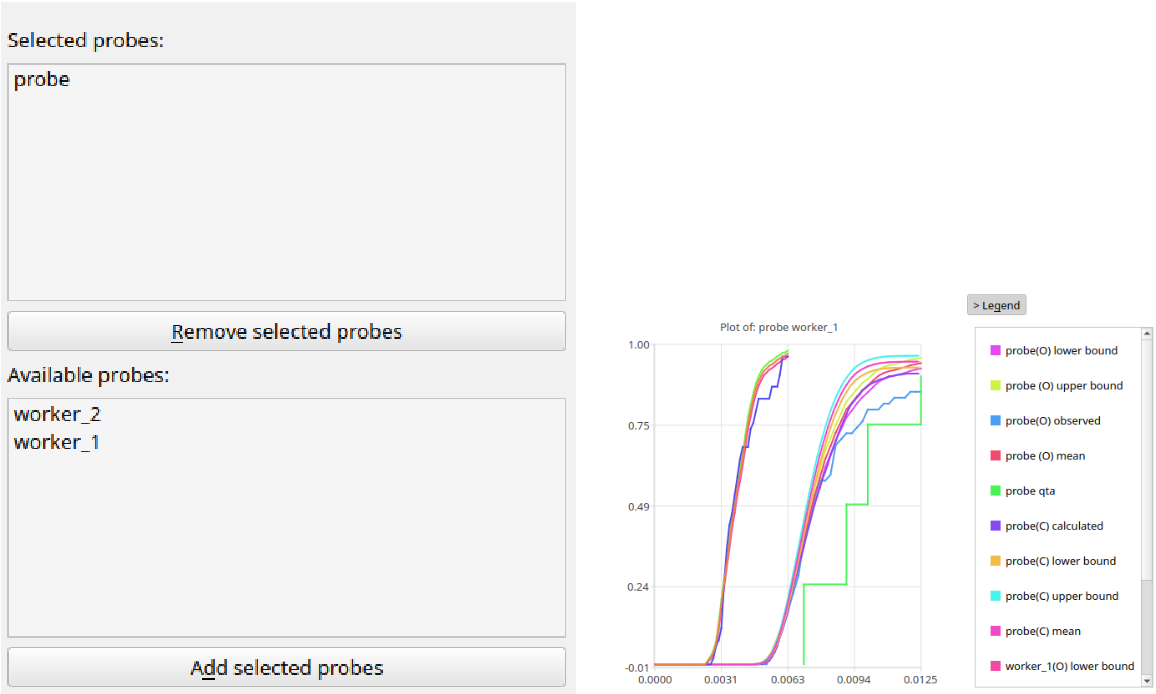

Figure D.11: To modify a plot, click on the probe you want to remove or add. Here, for example, worker_1's probe was added by selecting it below and clicking "Add selected probes".

D.2.6 Removing a plot

By left-clicking the plot, a popup appears, you can click **"Remove plot"**, this removes it from the selected plot (Fig. D.11).

D.3 Sidebar: Probes settings

Now that you have create your outcome diagram, you can modify the probes settings to set the $dMax$ you want and the QTA you desire.

D.3.1 Setting a QTA

You can set a QTA in the **"Set a QTA"** section, there you are presented with the possibility to:

- Select the probe for which you want to set the QTA.
- Set the QTA at the three percentiles (25%, 50%, 75%), the text to the left indicates which percentile is which. You need to set the delay in seconds.
- The minimum amount of successful events you can allow, which is bigger than 75%.

Of course, for the delay of the QTA at three percentiles, the delay at the percentile must be higher or equal than the delay at the previous percentile and higher than 0.

By pressing *"Save QTA settings"* you will save the QTA for the defined probe. Fig. D.12 shows how to do this.

D.3.2 Setting the parameters of a probe

You can set the parameters for a probe in its section. To the left you can select the probe you want to set the parameter for. We provide a slider (which goes from -10 to 10) for the n parameters, to the right of the slider, you can select the number of bins for the probe. The maximum delay calculated will be shown below.

Once you press **"Save delay"**, a message to the erlang adapter will be sent, which will set the maximum delay you have set in the adapter. Fig. D.12 shows how to do this.

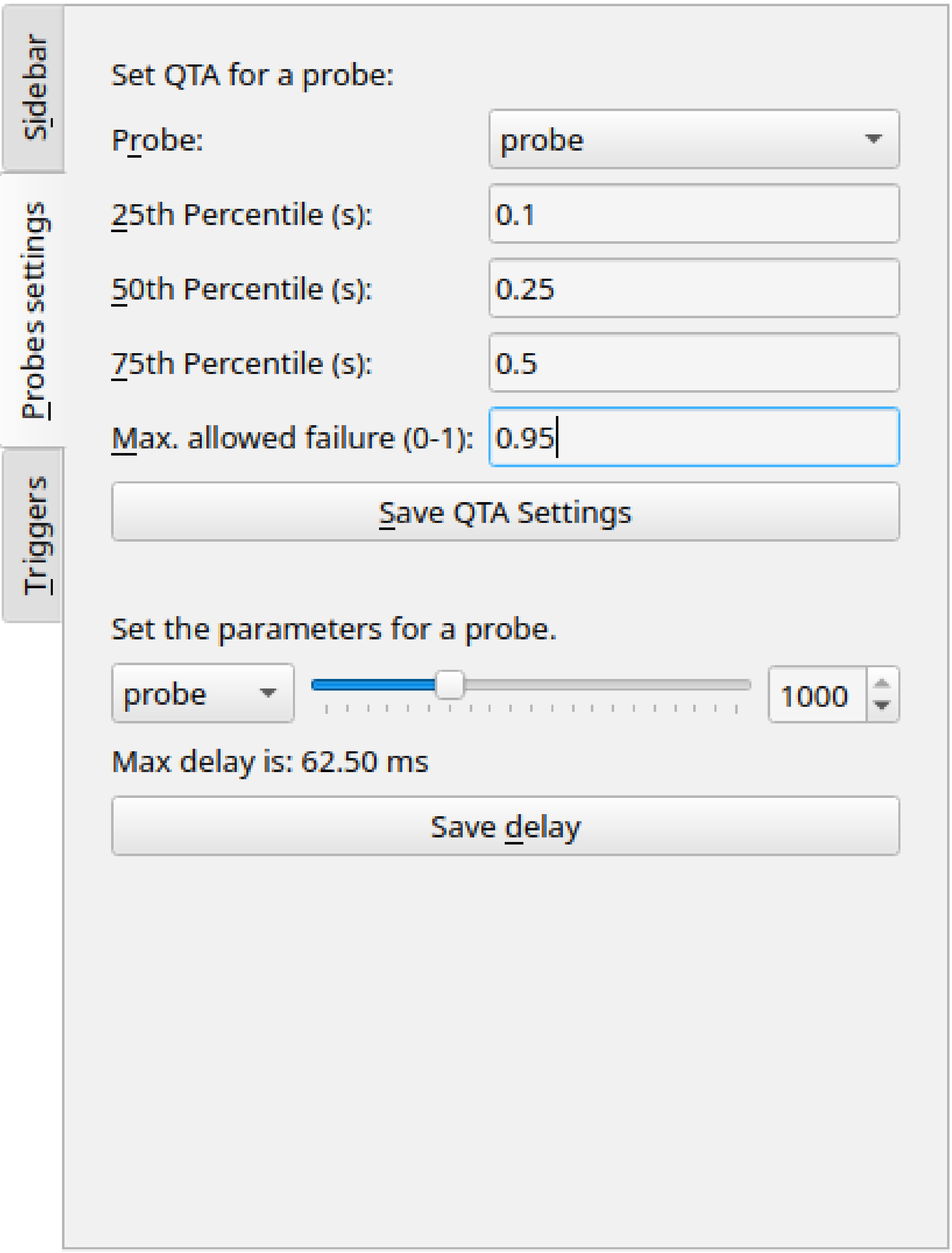

Figure D.12: Probes settings tab. Above: QTA settings. Below: Probe parameters settings.

D.4 Triggers

In the triggers section you can define which triggers to apply for a given probe, once selected, they will be automatically activated.

Once a trigger is triggered, the oscilloscope will keep recording the system for a few seconds. Once stopped, under the "snapshots" section, you can click the snapshots to view them in a separate section, there, you can observe the ΔQs of all the probes before and after the trigger was triggered. You can discard the snapshot by left clicking on it and clicking *"Delete snapshot"*.

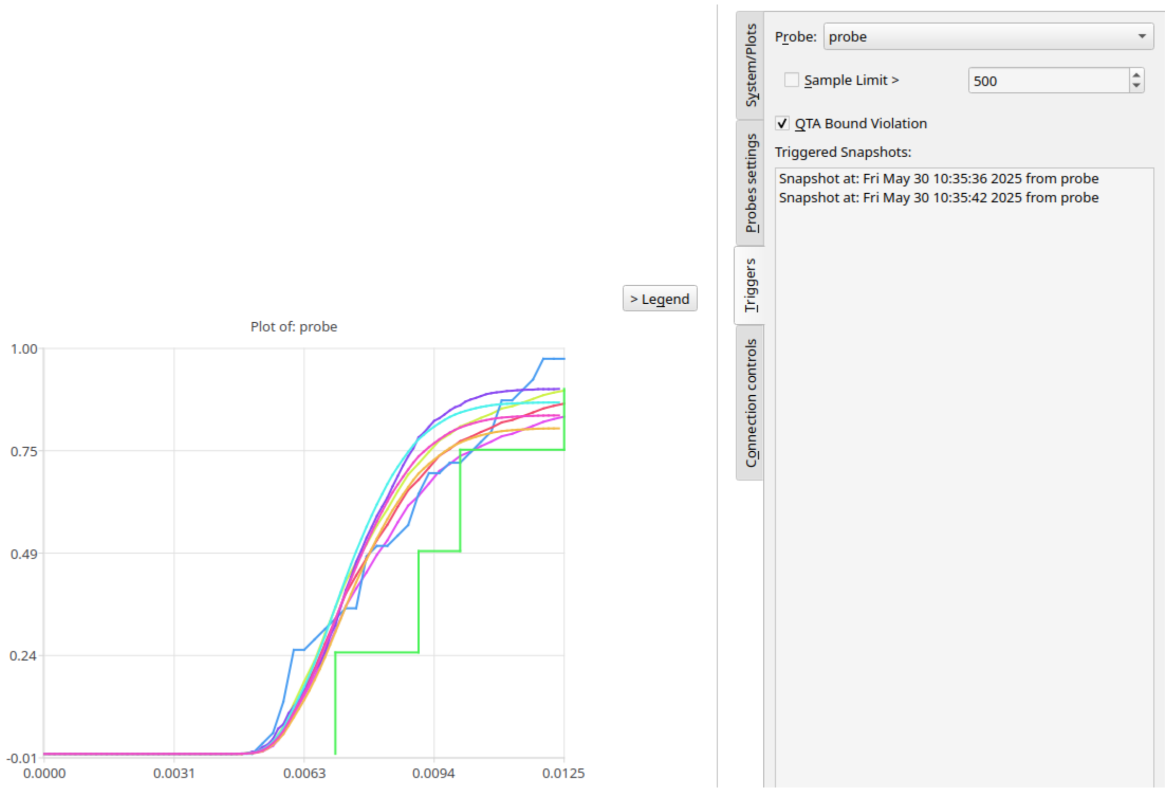

Figure D.13: Examples of the QTA being set for a probe "probe". Once a trigger is fired, the resulting snapshot from the fired trigger will be available under "Triggered snapshots".

D.5 Snapshots

The snapshots can be observed to look at the state of the system, before, during and after the trigger has been fired.

By clicking on the snapshot in the "triggered snapshots" section, a new window will popup where you can observe the system. A slider allows you to move backwards and forwards in time, observing the state of a probe in time.

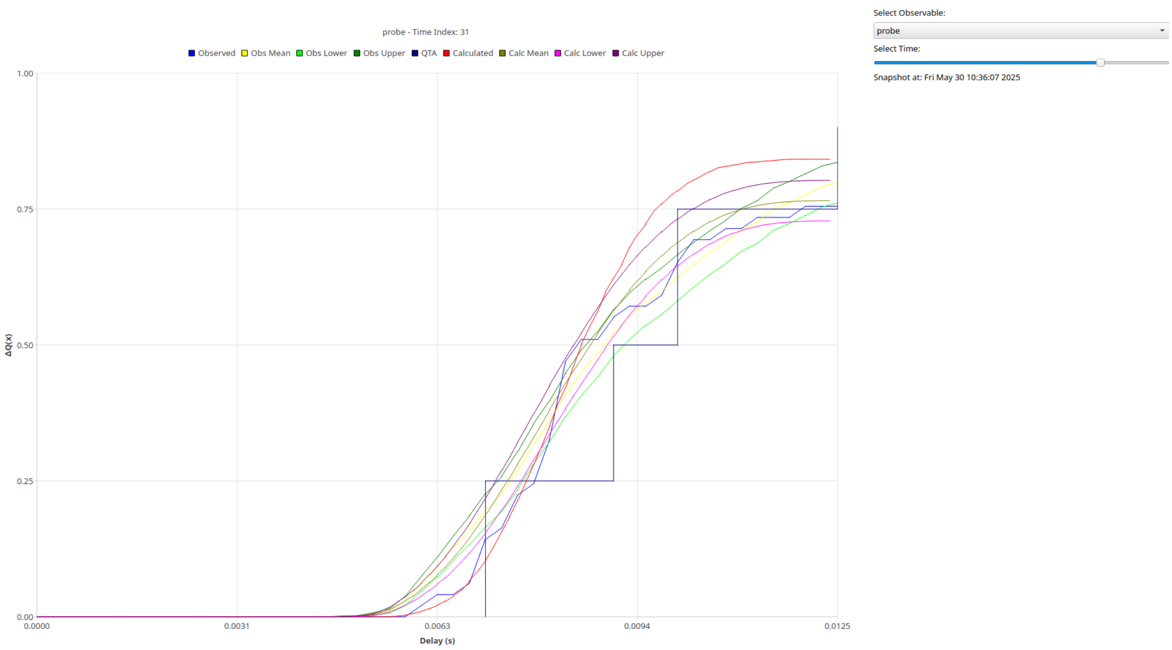

Figure D.14: Example of a snapshot, to the left, the graph at time t. To the right, the time of the graph on the left and a slider to advance backwards and forwards. As of right now, you can only select a probe at a time.

D.6 Instrumenting the Erlang application

D.6.1 Including the adapter

The Erlang project you need to instrument needs to include the adapter in its dependencies, to do that, you need to include it in your dependencies.

```
    %your_app.app.src
{application, otel_getting_started, [
    ...
    {applications, [
        kernel,
        stdlib,
        ...
    dqsd_otel
    ]},
    ...
    ]}.

    {deps, [
    {opentelemetry, "~> 1.3"},
    {opentelemetry_api, "~> 1.2"},
    {opentelemetry_exporter, "~> 1.0"},
    {dqsd_otel, {git, "https://github.com/fnieri/dqsd_otel.git", {tag,
 ↪  "the_latest_version_on_git"}}}
```

```
]}.
```

Once you have the dependencies set up you can begin creating outcome instances for the oscilloscope. (**Note:** If the project was to change name, you can still find the project in https://github.com/fnieri/).

D.6.2 Starting spans

To start spans you need to call:

```
{ProbeCtx, Pid} = dqsd_otel:start_span(<<"probe">>, #{attributes =>
↪   [{attr, <<"my_attr_5o5s10">>}]})
or without attributes
{ProbeCtx, Pid} = dqsd_otel:start_span(<<"probe">>)
```

This will give you the OpenTelemetry context of the probe and the Pid of the process to call upon end. It is left up to you to decide how to carry both in the execution. The function calls OpenTelemetry `?start_span` macro, effectively replacing it.

D.6.3 With spans

To start with spans you need to call:

```
dqsd_otel:with_span(<<"worker_2">>, fun() -> ok, #{attributes =>
↪   [{attr, <<"my_attr">>}]})
or without attributes
dqsd_otel:with_span(<<"worker_2">>, fun() -> ok end)
```

The second attribute is the function you want to evaluate.

D.6.4 Ending spans

To end spans you started with `dqsd_otel:start_span`, you need to call:

```
dqsd_otel:end_span(ProbeCtx, ProbePid)
```

This will end the span on the OpenTelemetry side and end the outcome instance if it hasn't timed out. The function calls OpenTelemetry `?end_span` macro, effectively replacing it.

D.6.5 Failing outcome instances

To fail **custom** spans you need to call:

```
dqsd_otel:fail_span(WorkerPid)
```

Contrary to the other methods, this does not end OpenTelemetry spans, it is let up to you to decide how to handle failure in spans.

Part III

Source Code Appendix

Appendix E

Grammar

This is the grammar for the ANTLR4 parser.

```
grammar DQGrammar;

PROBE_ID: 's';
BEHAVIOR_TYPE: 'f' | 'a' | 'p';
NUMBER: [0-9]+('.'[0-9]+)?;
IDENTIFIER: [a-zA-Z_][a-zA-Z0-9_]*;
WS: [ \t\r\n]+ -> skip;

// Parser Rules
start: definition* system? EOF;

definition: IDENTIFIER '=' component_chain ';';
system: 'system' '=' component_chain ';'?;

component_chain
    : component ('->' component)*
;

component
    : behaviorComponent
    | probeComponent
    | outcome
;

behaviorComponent
    : BEHAVIOR_TYPE ':' IDENTIFIER ('[' probability_list ']')? '('
   component_list ')'
;

probeComponent
    : PROBE_ID ':' IDENTIFIER
;

probability_list: NUMBER (',' NUMBER)+;
component_list: component_chain (',' component_chain)+;

outcome: IDENTIFIER;
```

Appendix F

C++ Source Files

F.1 Root folder

F.1.1 main.cpp

This is the main file of the project.

```cpp

#include "Application.h"
#include "dashboard/MainWindow.h"
#include "diagram/System.h"
#include "server/Server.h"
#include <QApplication>
#include <cstring>

#include <QApplication>
#include <QPalette>
#include <QStyleFactory>
#include <signal.h>
void setLightMode(QApplication &app)
{
    // Taken from https://forum.qt.io/topic/104406/styling-the-
   qmdiarea-using-the-the-fusion-style-and-qpalette-dark-theme
    // Use the Fusion style (consistent across platforms)
    app.setStyle(QStyleFactory::create("Fusion"));

    QPalette lightPalette;

    lightPalette.setColor(QPalette::Window, QColor(255, 255, 255));
    lightPalette.setColor(QPalette::WindowText, Qt::black);
    lightPalette.setColor(QPalette::Base, QColor(245, 245, 245));
    lightPalette.setColor(QPalette::AlternateBase, QColor(240, 240,
   240));
    lightPalette.setColor(QPalette::ToolTipBase, Qt::white);
    lightPalette.setColor(QPalette::ToolTipText, Qt::black);
    lightPalette.setColor(QPalette::Text, Qt::black);
    lightPalette.setColor(QPalette::Button, QColor(230, 230, 230));
    lightPalette.setColor(QPalette::ButtonText, Qt::black);
    lightPalette.setColor(QPalette::BrightText, Qt::red);
```

```
    lightPalette.setColor(QPalette::Link, QColor(0, 122, 204));

    lightPalette.setColor(QPalette::Highlight, QColor(0, 122, 204));
    lightPalette.setColor(QPalette::HighlightedText, Qt::white);

    app.setPalette(lightPalette);
}

int main(int argc, char *argv[])
{

    Server server(8080);
    signal(SIGPIPE, SIG_IGN); // Ignore SIGPIPE so when Erlang closes
   socket it will not crash
    Application &application = Application::getInstance();
    application.setServer(&server);
    System system = System();
    application.setSystem(system);
    QApplication app(argc, argv);
    setLightMode(app);
    MainWindow window;
    window.show();

    int result = app.exec();
    server.stop();
    return result;
}
```

F.1.2 Application.cpp

This is a singleton that handles the global state of the application.

```
#include "Application.h"

Application::Application()
{
}
Application &Application::getInstance()
{
    static Application instance;
    return instance;
}

std::shared_ptr<System> Application::getSystem()
{
    return system;
}

void Application::addObserver(std::function<void()> callback)
{
    observers.push_back(callback);
}

void Application::notifyObservers()
{
```

```
    for (auto &observer : observers) {
        observer();
    }
}

void Application::setServer(Server *s)
{
    server = s;
}

void Application::sendDelayChange(std::string &name, double newDelay)
{
    server->sendToErlang("set_timeout;" + name + ';' + std::to_string(
    newDelay));
}

void Application::setStubRunning(bool running)
{
    if (running)
        server->sendToErlang("start_stub");
    else
        server->sendToErlang("stop_stub");
}

SystemDiff Application::diffWith(System &newSystem)
{
    SystemDiff diff;

    auto oldSystem = getSystem();
    if (!oldSystem) {
        // No system yet: everything is new
        for (const auto &[name, _] : newSystem.getProbes())
            diff.addedProbes.push_back(name);
        for (const auto &[name, _] : newSystem.getOperators())
            diff.addedOperators.push_back(name);
        for (const auto &[name, _] : newSystem.getOutcomes())
            diff.addedOutcomes.push_back(name);
        return diff;
    }

    auto &oldProbes = oldSystem->getProbes();
    auto &newProbes = newSystem.getProbes();

    for (const auto &[name, probe] : newProbes) {
        if (!oldProbes.count(name)) {
            diff.addedProbes.push_back(name);
        } else if (componentsDiffer(oldProbes.at(name), probe)) {
            diff.changedProbes.push_back(name);
        }
    }
    for (const auto &[name, _] : oldProbes) {
        if (!newProbes.count(name))
            diff.removedProbes.push_back(name);
    }

    auto &oldOps = oldSystem->getOperators();
```

```
    auto &newOps = newSystem.getOperators();

    for (const auto &[name, op] : newOps) {
        if (!oldOps.count(name)) {
            diff.addedOperators.push_back(name);
        } else if (componentsDiffer(oldOps.at(name), op)) {
            diff.changedOperators.push_back(name);
        }
    }
    for (const auto &[name, _] : oldOps) {
        if (!newOps.count(name))
            diff.removedOperators.push_back(name);
    }

    auto &oldOut = oldSystem->getOutcomes();
    auto &newOut = newSystem.getOutcomes();

    for (const auto &[name, out] : newOut) {
        if (!oldOut.count(name)) {
            diff.addedOutcomes.push_back(name);
        } else if (componentsDiffer(oldOut.at(name), out)) {
            diff.changedOutcomes.push_back(name);
        }
    }
    for (const auto &[name, _] : oldOut) {
        if (!newOut.count(name))
            diff.removedOutcomes.push_back(name);
    }

    return diff;
}

bool Application::componentsDiffer(const std::shared_ptr<Observable> &
    a, const std::shared_ptr<Observable> &b)
{
    if (!a || !b || a->getName() != b->getName())
        return true;

    // For Probes, check causalLinks. For Operators, check type,
    probabilities, children.
    if (auto pa = std::dynamic_pointer_cast<Probe>(a)) {
        auto pb = std::dynamic_pointer_cast<Probe>(b);
        if (!pb)
            return true;

        auto ca = pa->getCausalLinks();
        auto cb = pb->getCausalLinks();

        if (ca.size() != cb.size())
            return true;
        for (size_t i = 0; i < ca.size(); ++i) {
            if (ca[i]->getName() != cb[i]->getName())
                return true;
        }
    }

```

```
    if (auto oa = std::dynamic_pointer_cast<Operator>(a)) {
        auto ob = std::dynamic_pointer_cast<Operator>(b);
        if (!ob)
            return true;

        if (oa->getType() != ob->getType())
            return true;

        auto ca = oa->getCausalLinks();
        auto cb = ob->getCausalLinks();

        if (ca.size() != cb.size())
            return true;
        for (size_t i = 0; i < ca.size(); ++i) {
            if (ca[i].size() == cb[i].size()) {
                for (size_t j = 0; j < ca[i].size(); ++j) {
                    if (ca[i][j]->getName() != cb[i][j]->getName())
                        return true;
                }

            } else {
                return true;
            }
        }
        auto probsA = oa->getProbabilities();
        auto probsB = ob->getProbabilities();
        if (probsA != probsB)
            return true;
    }

    return false;
}

void Application::setSystem(System newSystem)
{
    if (!system) {
        system = std::make_shared<System>(newSystem);
        notifyObservers();
        return;
    }
    SystemDiff diff = diffWith(newSystem);

    // Apply removals
    for (const auto &name : diff.removedProbes) {
        system->getProbes().erase(name);
        system->getObservables().erase(name);
    }
    for (const auto &name : diff.removedOperators) {
        system->getOperators().erase(name);
        system->getObservables().erase(name);
    }

    for (const auto &name : diff.removedOutcomes) {
        system->getOutcomes().erase(name);
        system->getObservables().erase(name);
    }
```

```

    // Apply changes and additions
    for (const auto &name : diff.changedProbes) {
        system->getProbes()[name]->setCausalLinks(newSystem.getProbes().at(name)->getCausalLinks());
        system->getObservables()[name] = newSystem.getProbes().at(name);
    }
    for (const auto &name : diff.addedProbes) {
        system->getProbes()[name] = newSystem.getProbes().at(name);
        system->getObservables()[name] = newSystem.getProbes().at(name);
    }
    for (const auto &name : diff.changedOperators) {
        system->getOperators()[name] = newSystem.getOperators().at(name);
        system->getObservables()[name] = newSystem.getOperators().at(name);
    }
    for (const auto &name : diff.addedOperators) {
        system->getOperators()[name] = newSystem.getOperators().at(name);
        system->getObservables()[name] = newSystem.getOperators().at(name);
    }

    for (const auto &name : diff.changedOutcomes)
        system->getOutcomes()[name] = newSystem.getOutcomes().at(name);
    for (const auto &name : diff.addedOutcomes) {
        system->getOutcomes()[name] = newSystem.getOutcomes().at(name);
        system->getObservables()[name] = newSystem.getOutcomes().at(name);
    }
    std::string defText = newSystem.getSystemDefinitionText();
    system->setSystemDefinitionText(defText);

    notifyObservers();
}

bool Application::startCppServer(const std::string &&ip, int port)
{
    return server->startServer(ip, port);
}

void Application::stopCppServer()
{
    server->stopServer();
}
void Application::setErlangEndpoint(const std::string &&ip, int port)
{
    server->setErlangEndpoint(ip, port);
}
```

F.2 Dashboard

This folder contains the widgets that compose the dashboard.

F.2.1 ColorRegistry.cpp

This class stores all the colors for all the QtSeries, moreover, it generates HSV colors based on the algorithm taken from `https://martin.ankerl.com/2009/12/09/how-to-create-random-colors-programmatically/`.

```cpp
#include "ColorRegistry.h"
#include <cmath>

std::unordered_map<std::string, QColor> ColorRegistry::colorMap;

QColor ColorRegistry::getColorFor(const std::string &name)
{
    if (colorMap.count(name))
        return colorMap[name];

    int index = colorMap.size();
    QColor color = generateDistinctColor(index);
    colorMap[name] = color;
    return color;
}

// Taken from
// https://martin.ankerl.com/2009/12/09/how-to-create-random-colors-programmatically/
QColor ColorRegistry::generateDistinctColor(int index)
{
    const double golden_ratio_conjugate = 137.508; // degrees
    double hue = std::fmod(index * golden_ratio_conjugate, 360.0);
    QColor color;
    color.setHsvF(hue / 360.0, 0.7, 0.95); // Saturation and Value tuned for brightness
    return color;
}
```

F.2.2 CustomLegendEntry.cpp

This class represents an entry in the plot legend.

```cpp
#include "CustomLegendEntry.h"
#include <QHBoxLayout>
CustomLegendEntry::CustomLegendEntry(const QString &name, const QColor &color, QWidget *parent)
    : QWidget(parent)
{
    layout = new QHBoxLayout(this);
    colorBox = new QLabel;
    colorBox->setFixedSize(12, 12);
    colorBox->setStyleSheet(QString("background-color: %1").arg(color.name()));
```

```
    nameLabel = new QLabel(name);
    layout->addWidget(colorBox);
    layout->addWidget(nameLabel);
    layout->addStretch();
}
```

F.2.3 CustomLegendPanel.cpp

This widget represents the legend panel for a plot, with entries corresponding to each series.

```
#include "CustomLegendPanel.h"
#include "CustomLegendEntry.h"

CustomLegendPanel::CustomLegendPanel(QWidget *parent)
    : QWidget(parent)
{
    // Initialize scroll area and configure its behavior
    scrollArea = new QScrollArea(this);
    scrollArea->setWidgetResizable(true);
    scrollArea->setVerticalScrollBarPolicy(Qt::ScrollBarAsNeeded);
    scrollArea->setHorizontalScrollBarPolicy(Qt::ScrollBarAlwaysOff);

    // Create content widget for the scroll area
    scrollContent = new QWidget();
    scrollArea->setWidget(scrollContent);

    // Set up layout for legend entries (aligned to top)
    legendLayout = new QVBoxLayout(scrollContent);
    legendLayout->setAlignment(Qt::AlignTop);

    // Set up main layout and add scroll area
    mainLayout = new QVBoxLayout(this);
    mainLayout->addWidget(scrollArea);
    setLayout(mainLayout);
}

/**
 * @brief Adds a new entry to the legend panel.
 * @param name The name for the entry.
 * @param color The color for the entry.
 */
void CustomLegendPanel::addEntry(const QString &name, const QColor &
    color)
{
    auto entry = new CustomLegendEntry(name, color, this);
    legendLayout->addWidget(entry);
    legendEntries[name] = entry;
}

/**
 * @brief Removes a specific entry from the legend by name.
 * @param name The name of the entry to remove.
 */
void CustomLegendPanel::removeEntry(const QString &name)
```

```
{
    if (legendEntries.count(name)) {
        QWidget *entry = legendEntries[name];
        legendLayout->removeWidget(entry);
        entry->deleteLater();
        legendEntries.erase(name);
    }
}

/**
 * @brief Clears all entries from the legend panel.
 */
void CustomLegendPanel::clear()
{
    for (auto it = legendEntries.begin(); it != legendEntries.end();)
   {
        QWidget *entry = it->second;
        legendLayout->removeWidget(entry);
        entry->deleteLater();
        it = legendEntries.erase(it);
    }
}
```

F.2.4 DQPlotController.cpp

This class is the controller of DeltaQPlot, based on the MVC design pattern.

```

#include "DQPlotController.h"
#include "../Application.h"
#include <QMetaObject>
#include <QVector>
#include <QtConcurrent>
#include <algorithm>
#include <cstdlib>
#include <qcontainerfwd.h>
#include <qlineseries.h>
using namespace std::chrono;
DQPlotController::DQPlotController(DeltaQPlot *plot, const std::vector
   <std::string> &selectedItems)
    : plot(plot)
{
    auto system = Application::getInstance().getSystem();
    for (const auto &name : selectedItems) {
        addComponent(name, (system->hasOutcome(name)));
    }
    setTitle();
}

DQPlotController::~DQPlotController()
{
    outcomes.clear();
    probes.clear();
    operators.clear();
}
```

```cpp

bool DQPlotController::isEmptyAfterReset()
{
    auto system = Application::getInstance().getSystem();

    std::lock_guard<std::mutex> lock(resetMutex);

    for (auto it = outcomes.begin(); it != outcomes.end();) {
        if (!system->hasOutcome(it->first)) {
            removeComponent(it->first);
            it = outcomes.begin();
        } else {
            ++it;
        }
    }

    for (auto it = probes.begin(); it != probes.end();) {
        if (!system->hasProbe(it->first)) {
            removeComponent(it->first);
            it = probes.begin();
        } else {
            ++it;
        }
    }
    for (auto it = operators.begin(); it != operators.end();) {
        if (!system->hasOperator(it->first)) {
            removeComponent(it->first);
            it = operators.begin();
        } else {
            ++it;
        }
    }
    return (outcomes.empty() && probes.empty() && operators.empty());
}

bool DQPlotController::containsComponent(std::string name)
{
    return ((outcomes.find(name) != outcomes.end()) || (probes.find(name) != probes.end()) || (operators.find(name) != operators.end()));
}

void DQPlotController::setTitle()
{
    std::vector<std::string> existingItems = getComponents();
    std::string title = "Plot of: ";
    for (const auto &name : existingItems) {
        title += name + " ";
    }
    plot->setTitle(QString::fromStdString(title));
}

void DQPlotController::editPlot(const std::vector<std::string> &selectedItems)
{
    std::vector<std::string> existingItems = getComponents();
```

```

    for (const auto &name : existingItems) {
        if (std::find(selectedItems.begin(), selectedItems.end(), name
    ) == selectedItems.end()) {
            removeComponent(name);
        }
    }
    auto system = Application::getInstance().getSystem();

    for (const auto &name : selectedItems) {
        if (!containsComponent(name)) {
            addComponent(name, system->hasOutcome(name));
        }
    }
    setTitle();
}

void DQPlotController::addComponent(const std::string &name, bool
    isOutcome)
{
    auto system = Application::getInstance().getSystem();
    if (isOutcome) {
        addOutcomeSeries(name);
    } else {
        addExpressionSeries(name, system->hasProbe(name));
    }
}

std::vector<std::string> DQPlotController::getComponents()
{
    std::vector<std::string> components;
    components.reserve(probes.size() + outcomes.size() + operators.
    size());

    for (const auto &kv : probes) {
        components.push_back(kv.first);
    }
    for (const auto &kv : outcomes) {
        components.push_back(kv.first);
    }
    for (const auto &kv : operators) {
        components.push_back(kv.first);
    }
    return components;
}

QLineSeries *DQPlotController::createAndAddLineSeries(const std::
    string &legendName)
{
    auto series = new QLineSeries();
    plot->addSeries(series, legendName);
    return series;
}

void DQPlotController::addOutcomeSeries(const std::string &name)
{
```

```
    auto system = Application::getInstance().getSystem();

    auto lowerBoundSeries = createAndAddLineSeries(name + "(O) lower
   bound");
    auto upperBoundSeries = createAndAddLineSeries(name + "(O) upper
   bound");
    auto outcomeSeries = createAndAddLineSeries(name + "(O) observed")
   ;
    auto meanSeries = createAndAddLineSeries(name + "(O) mean");
    auto qtaSeries = createAndAddLineSeries(name + "(O) qta");

    OutcomeSeries series
        = {.outcomeS = outcomeSeries, .lowerBoundS = lowerBoundSeries,
    .upperBoundS = upperBoundSeries, .meanS = meanSeries, .qtaS =
   qtaSeries};

    outcomes[name] = {series, system->getOutcome(name)};
}

void DQPlotController::addExpressionSeries(const std::string &name,
    bool isProbe)
{
    auto system = Application::getInstance().getSystem();

    auto obsLowerBoundSeries = createAndAddLineSeries(name + "(O)
   lower bound");
    auto obsUpperBoundSeries = createAndAddLineSeries(name + " (O)
   upper bound");
    auto obsSeries = createAndAddLineSeries(name + "(O) observed");
    auto obsMeanSeries = createAndAddLineSeries(name + " (O) mean");

    auto qtaSeries = createAndAddLineSeries(name + " qta");

    auto calcSeries = createAndAddLineSeries(name + "(C) calculated");
    auto calcLowerBoundSeries = createAndAddLineSeries(name + "(C)
   lower bound");
    auto calcUpperBoundSeries = createAndAddLineSeries(name + "(C)
   upper bound");
    auto calcMeanSeries = createAndAddLineSeries(name + "(C) mean");

    ExpressionSeries series = {
        .obsS = obsSeries,
        .obsLowerBoundS = obsLowerBoundSeries,
        .obsUpperBoundS = obsUpperBoundSeries,
        .obsMeanS = obsMeanSeries,
        .calcS = calcSeries,
        .calcLowerBoundS = calcLowerBoundSeries,
        .calcUpperBoundS = calcUpperBoundSeries,
        .calcMeanS = calcMeanSeries,
        .qtaS = qtaSeries,
    };
    if (isProbe)
        probes[name] = {series, system->getProbe(name)};
    else
        operators[name] = {series, system->getOperator(name)};
}
```

```
void DQPlotController::removeOutcomeSeries(const std::string &name)
{
    OutcomeSeries series = outcomes[name].first;

    plot->removeSeries(series.outcomeS);
    delete series.outcomeS;
    series.outcomeS = NULL;

    plot->removeSeries(series.lowerBoundS);
    delete series.lowerBoundS;
    series.lowerBoundS = NULL;

    plot->removeSeries(series.upperBoundS);
    delete series.upperBoundS;
    series.upperBoundS = NULL;

    plot->removeSeries(series.meanS);
    delete series.meanS;
    series.meanS = NULL;

    plot->removeSeries(series.qtaS);
    delete series.qtaS;
    series.qtaS = NULL;

    outcomes.erase(name);
}

void DQPlotController::removeExpressionSeries(const std::string &name,
    bool isProbe)
{
    ExpressionSeries series;
    if (isProbe)
        series = probes[name].first;
    else
        series = operators[name].first;

    plot->removeSeries(series.obsLowerBoundS);
    delete series.obsLowerBoundS;
    series.obsLowerBoundS = NULL;

    plot->removeSeries(series.obsUpperBoundS);
    delete series.obsUpperBoundS;
    series.obsUpperBoundS = NULL;

    plot->removeSeries(series.obsS);
    delete series.obsS;
    series.obsS = NULL;

    plot->removeSeries(series.obsMeanS);
    delete series.obsMeanS;
    series.obsMeanS = NULL;

    plot->removeSeries(series.qtaS);
    delete series.qtaS;
    series.qtaS = NULL;
```

```

    plot->removeSeries(series.calcS);
    delete series.calcS;
    series.calcS = NULL;

    plot->removeSeries(series.calcLowerBoundS);
    delete series.calcLowerBoundS;
    series.calcLowerBoundS = NULL;

    plot->removeSeries(series.calcUpperBoundS);
    delete series.calcUpperBoundS;
    series.calcUpperBoundS = NULL;

    plot->removeSeries(series.calcMeanS);
    delete series.calcMeanS;
    series.calcMeanS = NULL;

    if (isProbe)
        probes.erase(name);
    else
        operators.erase(name);
}

void DQPlotController::removeComponent(const std::string &name)
{
    if (outcomes.count(name)) {
        removeOutcomeSeries(name);
    }
    removeExpressionSeries(name, probes.count(name));
}

void DQPlotController::update(uint64_t timeLowerBound, uint64_t
    timeUpperBound)
{
    std::lock_guard<std::mutex> lock(updateMutex);
    double outcomeMax = 0;

    for (auto &[name, seriesOutcome] : outcomes) {
        double outcomeRange = updateOutcome(seriesOutcome.first,
    seriesOutcome.second, timeLowerBound, timeUpperBound);
        if (outcomeRange > outcomeMax) {
            outcomeMax = outcomeRange;
        }
    }

    double probeMax = 0;
    for (auto &[name, seriesProbe] : probes) {
        if (seriesProbe.second) {
            double probeRange = updateProbe(probes[name].first, probes
    [name].second, timeLowerBound, timeUpperBound);
            if (probeRange > probeMax) {
                probeMax = probeRange;
            }
        }
    }

```

```
    double operatorMax = 0;
    for (auto &[name, seriesOp] : operators) {
        if (seriesOp.second) {
            double opRange = updateOperator(operators[name].first, operators[name].second, timeLowerBound, timeUpperBound);
            if (opRange > operatorMax) {
                operatorMax = opRange;
            }
        }
    }
    plot->updateXRange(std::max({outcomeMax, probeMax, operatorMax}));
}

double DQPlotController::updateOutcome(OutcomeSeries &series, const std::shared_ptr<Outcome> &outcome, uint64_t timeLowerBound, uint64_t timeUpperBound)
{
    auto ret = QtConcurrent::run([=]() {
        double maxDelay = outcome->getMaxDelay();
        DeltaQRepr repr = outcome->getObservedDeltaQRepr(timeLowerBound, timeUpperBound);

        DeltaQ deltaQ = repr.deltaQ;
        std::vector<Bound> bounds = repr.bounds;
        int size = deltaQ.getBins();
        double binWidth = deltaQ.getBinWidth();

        QVector<QPointF> deltaQData;
        deltaQData.emplace_back(QPointF(0, 0));
        deltaQData.reserve(size);

        QVector<QPointF> lowerBoundData;
        lowerBoundData.emplace_back(QPointF(0, 0));
        lowerBoundData.reserve(size);

        QVector<QPointF> upperBoundData;
        upperBoundData.emplace_back(QPointF(0, 0));
        upperBoundData.reserve(size);

        QVector<QPointF> meanData;
        meanData.emplace_back(QPointF(0, 0));
        meanData.reserve(size);

        QVector<QPointF> qtaData;
        auto qta = outcome->getQTA();
        for (int i = 0; i < size; ++i) {
            double x = binWidth * (i + 1);

            deltaQData.emplace_back(QPointF(x, deltaQ.cdfAt(i)));
            lowerBoundData.emplace_back(QPointF(x, bounds[i].lowerBound));
            upperBoundData.emplace_back(QPointF(x, bounds[i].upperBound));
            meanData.emplace_back(QPointF(x, bounds[i].mean));
        }

```

```
        if (qta.defined) {
            qtaData.reserve(8);
            qtaData.emplace_back(qta.perc_25, 0);
            qtaData.emplace_back(qta.perc_25, 0.25);
            qtaData.emplace_back(qta.perc_50, 0.25);
            qtaData.emplace_back(qta.perc_50, 0.5);
            qtaData.emplace_back(qta.perc_75, 0.5);
            qtaData.emplace_back(qta.perc_75, 0.75);
            qtaData.emplace_back(maxDelay, 0.75);
            qtaData.emplace_back(maxDelay, qta.cdfMax);
        }
        QMetaObject::invokeMethod(
            plot,
            [=]() {
                //         auto guiStart = high_resolution_clock::now()
    ;

                plot->setUpdatesEnabled(false);

                plot->updateSeries(series.outcomeS, deltaQData);
                plot->updateSeries(series.lowerBoundS, lowerBoundData)
    ;
                plot->updateSeries(series.upperBoundS, upperBoundData)
    ;
                plot->updateSeries(series.meanS, meanData);
                plot->updateSeries(series.qtaS, qtaData);

                plot->setUpdatesEnabled(true);

                //     auto guiEnd = high_resolution_clock::now();
                //       qDebug() << "GUI update took" << duration_cast
    <microseconds>(guiEnd - guiStart).count() << "";
            },
            Qt::QueuedConnection);
    });
    ret.waitForFinished();
    return outcome->getMaxDelay();
}

double DQPlotController::updateOperator(ExpressionSeries &series, std
    ::shared_ptr<Operator> &op, uint64_t timeLowerBound, uint64_t
    timeUpperBound)
{
    updateExpression(series, op->getObservedDeltaQRepr(timeLowerBound,
     timeUpperBound), op->getCalculatedDeltaQRepr(timeLowerBound,
    timeUpperBound),
        op->getQTA(), op->getMaxDelay());

    return op->getMaxDelay();
}

double DQPlotController::updateProbe(ExpressionSeries &series, std::
    shared_ptr<Probe> &probe, uint64_t timeLowerBound, uint64_t
    timeUpperBound)
{
    updateExpression(series, probe->getObservedDeltaQRepr(
```

```
timeLowerBound, timeUpperBound), probe->getCalculatedDeltaQRepr(
timeLowerBound, timeUpperBound),
        probe->getQTA(), probe->getMaxDelay());
    return probe->getMaxDelay();
}

void DQPlotController::updateExpression(ExpressionSeries &series,
    DeltaQRepr &&obsRepr, DeltaQRepr &&calcRepr, QTA &&qta, double
    maxDelay)
{
    auto ret = QtConcurrent::run([=]() {
        auto computeStart = high_resolution_clock::now();

        DeltaQ obsDeltaQ = obsRepr.deltaQ;
        std::vector<Bound> obsBounds = obsRepr.bounds;
        DeltaQ calcDeltaQ = calcRepr.deltaQ;
        std::vector<Bound> calcBounds = calcRepr.bounds;
        int observedBins = obsDeltaQ.getBins();
        double observedBinWidth = obsDeltaQ.getBinWidth();

        // --- Prepare data ---
        QVector<QPointF> obsDeltaQData;
        obsDeltaQData.emplace_back(QPointF(0, 0));
        obsDeltaQData.reserve(observedBins);

        QVector<QPointF> obsLowerBoundData;
        obsLowerBoundData.emplace_back(QPointF(0, 0));
        obsLowerBoundData.reserve(observedBins);

        QVector<QPointF> obsUpperBoundData;
        obsUpperBoundData.emplace_back(QPointF(0, 0));
        obsUpperBoundData.reserve(observedBins);

        QVector<QPointF> obsMeanData;
        obsMeanData.emplace_back(QPointF(0, 0));
        obsMeanData.reserve(observedBins);

        for (int i = 0; i < observedBins; ++i) {
            double x = observedBinWidth * (i + 1);
            obsDeltaQData.emplace_back(QPointF(x, obsDeltaQ.cdfAt(i)))
;
            obsLowerBoundData.emplace_back(QPointF(x, obsBounds[i].
lowerBound));
            obsUpperBoundData.emplace_back(QPointF(x, obsBounds[i].
upperBound));
            obsMeanData.emplace_back(QPointF(x, obsBounds[i].mean));
        }

        QVector<QPointF> calcDeltaQData;
        calcDeltaQData.emplace_back(QPointF(0, 0));
        calcDeltaQData.reserve(observedBins);

        QVector<QPointF> calcLowerBoundData;
        calcLowerBoundData.emplace_back(QPointF(0, 0));
        calcLowerBoundData.reserve(observedBins);

```

```
        QVector<QPointF> calcUpperBoundData;
        calcUpperBoundData.emplace_back(QPointF(0, 0));
        calcUpperBoundData.reserve(observedBins);

        QVector<QPointF> calcMeanData;
        calcMeanData.emplace_back(QPointF(0, 0));
        calcMeanData.reserve(observedBins);

        QVector<QPointF> qtaData;

        // Prepare calculatedDeltaQ data
        int calculatedBins = calcDeltaQ.getBins();
        double calculatedBinWidth = calcDeltaQ.getBinWidth();
        for (int i = 0; i < calculatedBins; ++i) {
            double x = calculatedBinWidth * (i + 1);
            calcDeltaQData.emplace_back(QPointF(x, calcDeltaQ.cdfAt(i)));
            calcLowerBoundData.emplace_back(QPointF(x, calcBounds[i].lowerBound));
            calcUpperBoundData.emplace_back(QPointF(x, calcBounds[i].upperBound));
            calcMeanData.emplace_back(QPointF(x, calcBounds[i].mean));
        }
        if (qta.defined) {
            qtaData.reserve(8);
            qtaData.emplace_back(qta.perc_25, 0);
            qtaData.emplace_back(qta.perc_25, 0.25);
            qtaData.emplace_back(qta.perc_50, 0.25);
            qtaData.emplace_back(qta.perc_50, 0.5);
            qtaData.emplace_back(qta.perc_75, 0.5);
            qtaData.emplace_back(qta.perc_75, 0.75);
            qtaData.emplace_back(maxDelay, 0.75);
            qtaData.emplace_back(maxDelay, qta.cdfMax);
        }
        auto computeEnd = high_resolution_clock::now();
        //   std::cout << "comp," << observedBins << "," << calculatedBins << "," << duration_cast<microseconds>(computeEnd - computeStart).count();

        // --- Push results back to GUI thread ---

        QMetaObject::invokeMethod(
            plot,
            [=]() {
                auto guiStart = high_resolution_clock::now();

                plot->setUpdatesEnabled(false);

                plot->updateSeries(series.obsS, obsDeltaQData);
                plot->updateSeries(series.obsLowerBoundS, obsLowerBoundData);
                plot->updateSeries(series.obsUpperBoundS, obsUpperBoundData);
                plot->updateSeries(series.obsMeanS, obsMeanData);

                plot->updateSeries(series.calcS, calcDeltaQData);
```

```
                plot->updateSeries(series.calcLowerBoundS, calcLowerBoundData);
                plot->updateSeries(series.calcUpperBoundS, calcUpperBoundData);
                plot->updateSeries(series.calcMeanS, calcMeanData);
                plot->updateSeries(series.qtaS, qtaData);

                plot->setUpdatesEnabled(true);

                auto guiEnd = high_resolution_clock::now();
                // std::cout << "gui," << observedBins << "," << calculatedBins << "," << duration_cast<microseconds>(guiEnd - guiStart).count();
            },
            Qt::QueuedConnection);
    });
    ret.waitForFinished();
}
```

F.2.5 DQPlotList.cpp

This widget handles the adding and removing of probes series from a plot.

```
#include "DQPlotList.h"
#include "../Application.h"
#include <QLabel>
#include <QListWidgetItem>
#include <qlistwidget.h>
DQPlotList::DQPlotList(DQPlotController *controller, QWidget *parent)
    : QWidget(parent)
    , controller(controller)
{
    QVBoxLayout *layout = new QVBoxLayout(this);

    // Selected items list
    QLabel *selectedLabel = new QLabel("Selected probes:");
    selectedList = new QListWidget(this);
    layout->addWidget(selectedLabel);
    layout->addWidget(selectedList);
    selectedList->setSelectionMode(QAbstractItemView::MultiSelection);
    removeButton = new QPushButton("Remove selected probes", this);
    layout->addWidget(removeButton);

    connect(removeButton, &QPushButton::clicked, this, &DQPlotList::onRemoveSelection);
    // Available items list
    QLabel *availableLabel = new QLabel("Available probes:");
    availableList = new QListWidget(this);
    layout->addWidget(availableLabel);
    layout->addWidget(availableList);
    availableList->setSelectionMode(QAbstractItemView::MultiSelection);
    // Confirm button
    addButton = new QPushButton("Add selected probes", this);
```

```
    connect(addButton, &QPushButton::clicked, this, &DQPlotList::
   onConfirmSelection);
    layout->addWidget(addButton);

    updateLists();
}

bool DQPlotList::isEmptyAfterReset()
{
    return controller->isEmptyAfterReset();
}


void DQPlotList::updateLists()
{
    availableList->clear();
    selectedList->clear();

    auto plotComponents = controller->getComponents();
    auto system = Application::getInstance().getSystem();
    // Add components selected in a DeltaQPlot
    for (auto &component : plotComponents) {
        new QListWidgetItem(QString::fromStdString(component),
   selectedList);
    }

    // Select all components that are available to be chosen to add
    auto allComponents = system->getAllComponentsName();

    auto pred
        = [&plotComponents](const std::string &key) -> bool { return
   std::find(plotComponents.begin(), plotComponents.end(), key) !=
   plotComponents.end(); };

    allComponents.erase(std::remove_if(allComponents.begin(),
   allComponents.end(), pred), allComponents.end());
    for (auto &component : allComponents) {
        new QListWidgetItem(QString::fromStdString(component),
   availableList);
    }
}

void DQPlotList::onConfirmSelection()
{
    QList<QListWidgetItem *> selected = availableList->selectedItems()
   ;
    auto system = Application::getInstance().getSystem();
    for (QListWidgetItem *item : selected) {
        controller->addComponent(item->text().toStdString(), system->
   hasOutcome(item->text().toStdString()));
    }
    updateLists();
}

void DQPlotList::onRemoveSelection()
{
```

```
    QList<QListWidgetItem *> selected = selectedList->selectedItems();

    for (QListWidgetItem *item : selected) {
        controller->removeComponent(item->text().toStdString());
    }
    updateLists();
}
```

F.2.6 DelaySettingsWidget.cpp

This widget represent the slider widget to modify the parameters $dMax$, Δt and N.

```
/**
 * @file DelaySettingsWidget.cpp
 * @brief Implementation of the DelaySettingsWidget class.
 */

#include "DelaySettingsWidget.h"
#include "../Application.h"
#include <qlabel.h>
#include <qpushbutton.h>

DelaySettingsWidget::DelaySettingsWidget(QWidget *parent)
    : QWidget(parent)
{
    mainLayout = new QVBoxLayout(this);
    settingsLayout = new QHBoxLayout(this);

    settingsLabel = new QLabel("Set the parameters for a probe.");
    mainLayout->addWidget(settingsLabel);

    observableComboBox = new QComboBox();
    settingsLayout->addWidget(observableComboBox);

    delaySlider = new QSlider(Qt::Horizontal);
    delaySlider->setRange(-10, 10);
    delaySlider->setTickInterval(1);
    delaySlider->setTickPosition(QSlider::TicksBelow);
    settingsLayout->addWidget(delaySlider);

    binSpinBox = new QSpinBox();
    binSpinBox->setRange(1, 1000);
    binSpinBox->setValue(10);
    settingsLayout->addWidget(binSpinBox);

    mainLayout->addLayout(settingsLayout);

    maxDelayLabel = new QLabel("Max delay is: ");
    mainLayout->addWidget(maxDelayLabel);

    saveDelayButton = new QPushButton("Save delay");
    mainLayout->addWidget(saveDelayButton);

    connect(delaySlider, &QSlider::valueChanged, this, &
   DelaySettingsWidget::updateMaxDelay);
```

```cpp
    connect(binSpinBox, QOverload<int>::of(&QSpinBox::valueChanged),
   this, &DelaySettingsWidget::updateMaxDelay);
    connect(saveDelayButton, &QPushButton::clicked, this, &
   DelaySettingsWidget::onSaveDelayClicked);
    connect(observableComboBox, &QComboBox::currentTextChanged, this,
   &DelaySettingsWidget::loadObservableSettings);

    Application::getInstance().addObserver([this]() { this->
   populateComboBox(); });
}

/**
 * @brief Populates the combo box with available observables from the
   system.
 */
void DelaySettingsWidget::populateComboBox()
{
    auto system = Application::getInstance().getSystem();
    if (!system)
        return;

    observableComboBox->clear();
    for (const auto &[name, obs] : system->getObservables()) {
        if (obs)
            observableComboBox->addItem(QString::fromStdString(name));
    }
}

/**
 * @brief Loads delay settings for the currently selected observable.
 */
void DelaySettingsWidget::loadObservableSettings()
{
    auto system = Application::getInstance().getSystem();
    if (!system)
        return;

    QString observableName = observableComboBox->currentText();
    if (observableName.isEmpty())
        return;

    auto observable = system->getObservable(observableName.toStdString
   ());

    auto exponent = observable->getDeltaTExp();
    auto bins = observable->getNBins();
    delaySlider->setValue(exponent);
    binSpinBox->setValue(bins);

    updateMaxDelay();
}

/**
 * @brief Computes the current maximum delay.
 * @return Maximum delay value based on bin count and delay exponent.
 */
```

```
double DelaySettingsWidget::getMaxDelayMs() const
{
    int exponent = delaySlider->value();
    int bins = binSpinBox->value();
    return 1.0 * std::pow(2.0, exponent) * bins;
}

/**
 * @brief Updates the label that shows the computed max delay value.
 */
void DelaySettingsWidget::updateMaxDelay()
{
    double delay = getMaxDelayMs();
    maxDelayLabel->setText(QString("Max delay is: %1 ms").arg(delay,
   0, 'f', 2));
}

/**
 * @brief Saves the current delay settings to the system and emits a
    change signal.
 */
void DelaySettingsWidget::onSaveDelayClicked()
{
    auto system = Application::getInstance().getSystem();
    if (!system)
        return;

    QString name = observableComboBox->currentText();
    if (name.isEmpty())
        return;

    int exponent = delaySlider->value();
    int bins = binSpinBox->value();
    std::string nameString = name.toStdString();
    system->setObservableParameters(nameString, exponent, bins);

    Q_EMIT delayParametersChanged();
}
```

F.2.7 DeltaQPlot.cpp

This widget is the widget containing a plot and its legend.

```

#include "DeltaQPlot.h"
#include "ColorRegistry.h"
#include "CustomLegendPanel.h"
#include "DQPlotList.h"
#include <QChartView>
#include <QDebug>
#include <QHBoxLayout>
#include <QLineSeries>
#include <QMouseEvent>
#include <QRandomGenerator>
#include <QToolButton>
```

```cpp
#include <QVBoxLayout>
DeltaQPlot::DeltaQPlot(const std::vector<std::string> &selectedItems,
    QWidget *parent)
     : QWidget(parent)
{

    // Create legend panel and toggle
    legendPanel = new CustomLegendPanel(this);
    QToolButton *toggleButton = new QToolButton(this);
    toggleButton->setText("> Legend");
    toggleButton->setCheckable(true);
    toggleButton->setChecked(true);
    toggleButton->setToolButtonStyle(Qt::ToolButtonTextBesideIcon);

    connect(toggleButton, &QToolButton::toggled, legendPanel, &QWidget
    ::setVisible);
    connect(toggleButton, &QToolButton::toggled, [toggleButton](bool
    checked) { toggleButton->setText(checked ? "^ Legend" : "> Legend"
    ); });

    // Create chart and view
    chart = new QChart();
    chartView = new QChartView(chart, this);
    chartView->setRenderHint(QPainter::Antialiasing);
    chart->legend()->setVisible(false);

    axisX = new QValueAxis();
    axisY = new QValueAxis();
    axisY->setRange(-0.01, 1.0);
    axisX->setRange(0, 0.05);
    chart->addAxis(axisX, Qt::AlignBottom);
    chart->addAxis(axisY, Qt::AlignLeft);

    controller = new DQPlotController(this, selectedItems);
    plotList = new DQPlotList(controller, this);
    // Right-side layout: toggle + legend
    QVBoxLayout *rightLayout = new QVBoxLayout();
    rightLayout->addWidget(toggleButton);
    rightLayout->addWidget(legendPanel);
    rightLayout->addStretch();

    // Main layout: chart + right side
    QHBoxLayout *mainLayout = new QHBoxLayout(this);
    mainLayout->addWidget(chartView, 1);
    mainLayout->addLayout(rightLayout);
    mainLayout->setContentsMargins(0, 0, 0, 0);
    mainLayout->setSpacing(5);
    setLayout(mainLayout);
    chartView->setRenderHint(QPainter::Antialiasing);
}

DeltaQPlot::~DeltaQPlot()
{
    delete controller;
    controller = nullptr;
    delete plotList;
```

```cpp
    plotList = nullptr;
}

bool DeltaQPlot::isEmptyAfterReset()
{
    if (!controller->isEmptyAfterReset()) {
        plotList->updateLists();
        return false;
    }
    return true;
}

void DeltaQPlot::setTitle(QString &&title)
{
    chart->setTitle(title);
}

void DeltaQPlot::addSeries(QLineSeries *series, const std::string &
    name)
{
    chart->addSeries(series);
    series->setName(QString::fromStdString(name));
    series->attachAxis(axisX);
    series->attachAxis(axisY);
    QColor color = ColorRegistry::getColorFor(name);
    legendPanel->addEntry(QString::fromStdString(name), color);
    series->setColor(color);
    series->setVisible(true);
}

void DeltaQPlot::update(uint64_t timeLowerBound, uint64_t
    timeUpperBound)
{
    controller->update(timeLowerBound, timeUpperBound);
}

void DeltaQPlot::removeSeries(QAbstractSeries *series)
{
    chart->removeSeries(series);
    auto name = series->name();
    legendPanel->removeEntry(name);
}

void DeltaQPlot::editPlot(const std::vector<std::string> &
    selectedItems)
{
    controller->editPlot(selectedItems);
}

void DeltaQPlot::updateSeries(QLineSeries *series, const QVector<
    QPointF> &data)
{
    series->replace(data);
}

void DeltaQPlot::updateXRange(double xRange)
```

```
{
    axisX->setRange(0, xRange);
}

std::vector<std::string> DeltaQPlot::getComponents()
{
    return controller->getComponents();
}

DQPlotList *DeltaQPlot::getPlotList()
{
    return plotList;
}

void DeltaQPlot::mousePressEvent(QMouseEvent *event)
{
    Q_EMIT plotSelected(this);
}
```

F.2.8 MainWindow.cpp

This widget is the main window of the application, it has a tab to the side where the widgets to control the oscilloscope are. To the left, the panel where all plots are shown.

```
#include "MainWindow.h"
#include "../Application.h"
#include "../maths/DeltaQOperations.h"
#include "DQPlotList.h"
#include "DeltaQPlot.h"
#include "NewPlotList.h"
#include "ObservableSettings.h"
#include "Sidebar.h"
#include <QMenu>
#include <QMessageBox>
#include <QThread>
#include <QVBoxLayout>
#define MAX_P_ROW 2
#define MAX_P_COL 2

MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
{
    // Set up central widget and main layout
    centralWidget = new QWidget(this);
    setCentralWidget(centralWidget);
    mainLayout = new QHBoxLayout(centralWidget);

    // Configure scroll area for plots
    scrollArea = new QScrollArea(this);
    scrollArea->setWidgetResizable(true);
    scrollArea->setHorizontalScrollBarPolicy(Qt::ScrollBarAsNeeded);
    scrollArea->setVerticalScrollBarPolicy(Qt::ScrollBarAsNeeded);
    scrollArea->setBaseSize(800, 800);

    // Set up plot container with grid layout
```

```
    plotContainer = new QWidget();
    plotLayout = new QGridLayout(plotContainer);
    scrollArea->setWidget(plotContainer);
    mainLayout->addWidget(scrollArea, 1);

    // Initialize side panels
    sidebar = new Sidebar(this);
    triggersTab = new TriggersTab(this);
    observableSettings = new ObservableSettings(this);
    stubWidget = new StubControlWidget(this);

    // Configure tabbed side panel
    sideTabWidget = new QTabWidget(this);
    sideTabWidget->addTab(sidebar, "System/Plots");
    sideTabWidget->addTab(observableSettings, "Probes settings");
    sideTabWidget->addTab(triggersTab, "Triggers");
    sideTabWidget->addTab(stubWidget, "Connection controls");
    sideTabWidget->setTabPosition(QTabWidget::West);
    // Connect sidebar signals
    connect(sidebar, &Sidebar::addPlotClicked, this, &MainWindow::
   onAddPlotClicked);

    // Set up side container layout
    sideContainer = new QWidget(this);
    sideLayout = new QVBoxLayout(sideContainer);
    sideLayout->addWidget(sideTabWidget);
    sideLayout->setStretch(1, 0); // Make tabs take up most space
    mainLayout->addWidget(sideContainer, 0);

    // Set up update timer in separate thread
    timerThread = new QThread(this);
    updateTimer = new QTimer();
    updateTimer->moveToThread(timerThread);
    connect(updateTimer, &QTimer::timeout, this, &MainWindow::
   updatePlots, Qt::QueuedConnection);
    connect(timerThread, &QThread::started, [this]() { updateTimer->
   start(200); });
    timerThread->start();

    // Register system reset observer
    Application::getInstance().addObserver([this] { this->reset(); });

    // Initialize time bounds
    auto now = std::chrono::system_clock::now();
    auto adjustedTime = now - std::chrono::milliseconds(200);
    timeLowerBound = std::chrono::duration_cast<std::chrono::
   nanoseconds>(adjustedTime.time_since_epoch()).count();

    // Connect sampling rate changes
    connect(sidebar, &Sidebar::onSamplingRateChanged, this, [this](int
    ms) {
        qDebug() << "MainWindow received sampling rate:" << ms;
        samplingRate = ms;
        QMetaObject::invokeMethod(updateTimer, [ms, this]() {
   updateTimer->setInterval(ms); }, Qt::QueuedConnection);
    });
```

```cpp
}

/**
 * @brief Cleans up empty plots during reset.
 */
void MainWindow::reset()
{
    std::lock_guard<std::mutex> lock(plotDelMutex);
    auto it = plotContainers.begin();
    while (it != plotContainers.end()) {
        DeltaQPlot *plot = it.key();
        QWidget *plotWidget = it.value();

        if (plot->isEmptyAfterReset()) {
            it = plotContainers.erase(it);
            delete plot;
            if (plotWidget) {
                delete plotWidget;
            }
            sidebar->hideCurrentPlot();
        } else {
            ++it;
        }
    }
}

/**
 * @brief Updates all plots with new data from the system.
 */
void MainWindow::updatePlots()
{
    // Update time bounds
    timeLowerBound += std::chrono::duration_cast<std::chrono::nanoseconds>(std::chrono::milliseconds(samplingRate)).count();
    uint64_t timeUpperBound = timeLowerBound + std::chrono::duration_cast<std::chrono::nanoseconds>(std::chrono::milliseconds(samplingRate)).count();

    auto system = Application::getInstance().getSystem();
    std::lock_guard<std::mutex> lock(plotDelMutex);

    // Update probes
    for (auto &[name, probe] : system->getProbes()) {
        if (probe) {
            probe->getObservedDeltaQ(timeLowerBound, timeUpperBound);
            probe->calculateCalculatedDeltaQ(timeLowerBound, timeUpperBound);
        }
    }

    // Update operators
    for (auto &[name, op] : system->getOperators()) {
        if (op) {
            op->getObservedDeltaQ(timeLowerBound, timeUpperBound);
            op->calculateCalculatedDeltaQ(timeLowerBound, timeUpperBound);
```

```
        }
    }

    // Update outcomes
    for (auto &[name, outcome] : system->getOutcomes()) {
        if (outcome) {
            outcome->getObservedDeltaQ(timeLowerBound, timeUpperBound)
;
        }
    }

    // Update all plots
    for (auto [plot, _] : plotContainers.asKeyValueRange()) {
        plot->update(timeLowerBound, timeUpperBound);
    }
}

/**
 * @brief Destructor cleans up timer thread and resources.
 */
MainWindow::~MainWindow()
{
    if (timerThread) {
        timerThread->quit();
        timerThread->wait();
        delete timerThread;
    }
}

/**
 * @brief Adds a new plot based on sidebar selection.
 */
void MainWindow::onAddPlotClicked()
{
    auto selectedItems = sidebar->getPlotList()->getSelectedItems();

    if (selectedItems.empty()) {
        QMessageBox::warning(this, "No Selection", "Please select
components before adding a plot.");
        return;
    }

    // Create new plot container
    auto *plotWidget = new QWidget(this);
    auto *plotWidgetLayout = new QVBoxLayout(plotWidget);
    auto *deltaQPlot = new DeltaQPlot(selectedItems, this);

    // Set up connections and tracking
    connect(deltaQPlot, &DeltaQPlot::plotSelected, this, &MainWindow::
onPlotSelected);
    plotContainers[deltaQPlot] = plotWidget;

    // Configure layout
    plotWidgetLayout->addWidget(deltaQPlot);
    plotWidget->setMaximumWidth(scrollArea->width() / MAX_P_ROW);
    plotWidget->setMaximumHeight(scrollArea->height() / 2);
```

```

    // Position in grid
    int plotCount = plotContainers.size();
    int row = (plotCount - 1) / MAX_P_ROW;
    int col = (plotCount - 1) % MAX_P_COL;
    plotLayout->addWidget(plotWidget, row, col);

    // Update UI state
    onPlotSelected(deltaQPlot);
    sidebar->clearOnAdd();
}

/**
 * @brief Handles window resize events to adjust plot sizes.
 * @param event The resize event.
 */
void MainWindow::resizeEvent(QResizeEvent *event)
{
    QMainWindow::resizeEvent(event);
    for (auto [plot, widget] : plotContainers.asKeyValueRange()) {
        widget->setMaximumWidth(scrollArea->width() / MAX_P_COL);
        widget->setMaximumHeight(scrollArea->height() / 2);
    }
}

/**
 * @brief Shows context menu for plot management.
 * @param event The context menu event.
 */
void MainWindow::contextMenuEvent(QContextMenuEvent *event)
{
    QWidget *child = childAt(event->pos());
    if (!child)
        return;

    // Find which plot was right-clicked
    DeltaQPlot *selectedPlot = nullptr;
    for (auto it = plotContainers.begin(); it != plotContainers.end();
    ++it) {
        if (it.value()->isAncestorOf(child)) {
            selectedPlot = it.key();
            break;
        }
    }

    if (!selectedPlot)
        return;

    // Create and show context menu
    QMenu contextMenu(this);
    QAction *removeAction = contextMenu.addAction("Remove Plot");

    QAction *selectedAction = contextMenu.exec(event->globalPos());

    if (selectedAction == removeAction) {
        onRemovePlot(selectedPlot);
```

```
    }
}

/**
 * @brief Removes a plot and cleans up resources.
 * @param plot The plot to remove.
 */
void MainWindow::onRemovePlot(DeltaQPlot *plot)
{
    QWidget *plotWidget = plotContainers.value(plot, nullptr);
    if (plotWidget) {
        plotLayout->removeWidget(plotWidget);
        plotWidget->deleteLater();
    }
    plotContainers.remove(plot);

    // Reorganize remaining plots
    int plotCount = 0;
    for (auto it = plotContainers.begin(); it != plotContainers.end();
    ++it) {
        int row = plotCount / MAX_P_ROW;
        int col = plotCount % MAX_P_COL;
        plotLayout->addWidget(it.value(), row, col);
        plotLayout->setRowStretch(row, 1);
        plotLayout->setColumnStretch(col, 1);
        plotCount++;
    }

    sidebar->hideCurrentPlot();
}

/**
 * @brief Updates sidebar when a plot is selected.
 * @param plot The newly selected plot.
 */
void MainWindow::onPlotSelected(DeltaQPlot *plot)
{
    if (!plot)
        return;
    DQPlotList *plotList = plot->getPlotList();
    if (!plotList) {
        return;
    }
    sidebar->setCurrentPlotList(plotList);
}
```

F.2.9 NewPlotList.cpp

This widget is the widget to add a new plot to the plots panel.

```
/**
 * @file NewPlotList.cpp
 * @brief Implementation of the NewPlotList class, which provides a UI
    list for selecting observables to create a new plot.
 */
```

```cpp

#include "NewPlotList.h"
#include "../Application.h"
#include <iostream>
#include <qlistwidget.h>

NewPlotList::NewPlotList(QWidget *parent)
    : QListWidget(parent)
{
    setSelectionMode(QAbstractItemView::MultiSelection);
    Application::getInstance().addObserver([this]() { this->reset();
   });
}

/**
 * @brief Clears and repopulates the list with updated observables.
 *
 */
void NewPlotList::reset()
{
    this->blockSignals(true);
    while (count() != 0)
        delete takeItem(0);
    this->blockSignals(false);
    addItems();
}

/**
 * @brief Adds all observables from the system as items in the list.
 */
void NewPlotList::addItems()
{
    auto system = Application::getInstance().getSystem();
    auto observables = system->getObservables();

    for (const auto &obs : observables) {
        if (obs.second) {
            QListWidgetItem *item = new QListWidgetItem(QString::
   fromStdString(obs.first), this);
            item->setFlags(Qt::ItemIsSelectable | Qt::ItemIsEnabled);
        }
    }
}

/**
 * @brief Returns a list of selected observable names.
 * @return A vector of strings corresponding to selected item labels.
 */
std::vector<std::string> NewPlotList::getSelectedItems()
{
    std::vector<std::string> selectedItems;
    QList<QListWidgetItem *> selected = this->selectedItems();

    for (QListWidgetItem *item : selected) {
        selectedItems.push_back(item->text().toStdString());
    }
```

```
    return selectedItems;
}

/**
 * @brief Deselects all currently selected items in the list.
 */
void NewPlotList::deselectAll()
{
    clearSelection();
}
```

F.2.10 ObservableSettings.cpp

This widget is a tab that contains the settings for the probes (Setting a QTA, setting $dMax$).

```
#include "ObservableSettings.h"
#include <qlabel.h>
#include <qwidget.h>

ObservableSettings::ObservableSettings(QWidget *parent)
    : QWidget(parent)
{
    layout = new QVBoxLayout(this);
    layout->setAlignment(Qt::AlignTop);
    layout->setSpacing(10);
    layout->setContentsMargins(10, 10, 10, 10);

    qtaInputWidget = new QTAInputWidget(this);
    layout->addWidget(qtaInputWidget);

    delaySettingsWidget = new DelaySettingsWidget(this);
    layout->addWidget(delaySettingsWidget);
    connect(delaySettingsWidget, &DelaySettingsWidget::
   delayParametersChanged, qtaInputWidget, &QTAInputWidget::
   loadObservableSettings);
}
```

F.2.11 SamplingRateWidget.cpp

This widget allows the sampling rate to be changed via a slider.

```
#include "SamplingRateWidget.h"

SamplingRateWidget::SamplingRateWidget(QWidget *parent)
    : QWidget(parent)
{
    // Available sampling rate options in milliseconds
    samplingRates = {100, 200, 300, 400, 500, 600, 700, 800, 900,
   1000, 2000, 5000, 10000};

    // Configure rate selection slider
    slider = new QSlider(Qt::Horizontal);
    slider->setMinimum(0);
```

```
    slider->setMaximum(samplingRates.size() - 1);
    slider->setValue(1);
    slider->setTickInterval(1);
    slider->setTickPosition(QSlider::TicksBelow);

    valueLabel = new QLabel(QString("Sampling Rate: %1 ms").arg(
   samplingRates[0]));

    saveButton = new QPushButton("Save sampling rate");

    // Set up layout
    auto *layout = new QVBoxLayout();
    layout->addWidget(slider);
    layout->addWidget(valueLabel);
    layout->addWidget(saveButton);
    setLayout(layout);

    // Connect signals
    connect(slider, &QSlider::valueChanged, this, &SamplingRateWidget
   ::onSliderValueChanged);
    connect(saveButton, &QPushButton::clicked, this, &
   SamplingRateWidget::onSaveClicked);
}

/**
 * @brief Updates the displayed rate when slider changes.
 * @param value The current slider index (0-based).
 *
 * Converts slider index to actual milliseconds and updates the
   display label.
 */
void SamplingRateWidget::onSliderValueChanged(int value)
{
    int ms = samplingRates[value];
    valueLabel->setText(QString("Sampling Rate: %1 ms").arg(ms));
}

/**
 * @brief Emits the selected sampling rate when save is clicked.
 *
 * Gets the current slider value, converts it to milliseconds,
 * and emits the onSamplingRateChanged signal.
 */
void SamplingRateWidget::onSaveClicked()
{
    int ms = samplingRates[slider->value()];
    Q_EMIT onSamplingRateChanged(ms);
}
```

F.2.12 QTAInputWidget.cpp

This widget allows the QTA to be set for a probe

```
#include "QTAInputWidget.h"
#include "../Application.h"
```

```cpp
#include <QDoubleValidator>
#include <QMessageBox>
#include <QPushButton>

QTAInputWidget::QTAInputWidget(QWidget *parent)
    : QWidget(parent)
{
    auto layout = new QFormLayout(this);

    qtaLabel = new QLabel(this);
    qtaLabel->setText("Set QTA for a probe:");
    layout->addRow(qtaLabel);
    observableComboBox = new QComboBox(this);
    layout->addRow("Probe:", observableComboBox);

    perc25Edit = new QLineEdit(this);
    perc25Edit->setPlaceholderText("Seconds (s)");

    perc50Edit = new QLineEdit(this);
    perc50Edit->setPlaceholderText("Seconds (s)");

    perc75Edit = new QLineEdit(this);
    perc75Edit->setPlaceholderText("Seconds (s)");

    cdfMaxEdit = new QLineEdit(this);
    cdfMaxEdit->setPlaceholderText("Value between 0 and 1");

    layout->addRow("25th Percentile (s):", perc25Edit);
    layout->addRow("50th Percentile (s):", perc50Edit);
    layout->addRow("75th Percentile (s):", perc75Edit);
    layout->addRow("Max. allowed failure (0-1):", cdfMaxEdit);

    connect(observableComboBox, &QComboBox::currentTextChanged, this,
   &QTAInputWidget::loadObservableSettings);

    Application::getInstance().addObserver([this]() { this->
   populateComboBox(); });
    saveButton = new QPushButton("Save QTA Settings", this);
    layout->addRow(saveButton);
    connect(saveButton, &QPushButton::clicked, this, &QTAInputWidget::
   onSaveButtonClicked);

    setLayout(layout);

    Application::getInstance().addObserver([this]() { this->
   populateComboBox(); });
}

void QTAInputWidget::populateComboBox()
{
    auto system = Application::getInstance().getSystem();
    if (!system)
        return;

    observableComboBox->clear();
    for (const auto &[name, _] : system->getProbes()) {
```

```
            observableComboBox->addItem(QString::fromStdString(name));
        }
        for (const auto &[name, _] : system->getOutcomes()) {
            observableComboBox->addItem(QString::fromStdString(name));
        }
}

void QTAInputWidget::loadObservableSettings()
{
        auto system = Application::getInstance().getSystem();
        if (!system)
            return;
        std::string observableName = observableComboBox->currentText().
    toStdString();
        auto observable = system->getObservable(observableName);
        if (observable) {
            auto qta = observable->getQTA();
            perc25Edit->setText(QString::number(qta.perc_25, 'f', 6)); //
    6 decimal places
            perc50Edit->setText(QString::number(qta.perc_50, 'f', 6));
            perc75Edit->setText(QString::number(qta.perc_75, 'f', 6));
            cdfMaxEdit->setText(QString::number(qta.cdfMax, 'f', 6));
        }
}

void QTAInputWidget::onSaveButtonClicked()
{
        auto system = Application::getInstance().getSystem();
        if (!system)
            return;

        std::string observableName = observableComboBox->currentText().
    toStdString();
        auto observable = system->getObservable(observableName);
        if (!observable)
            return;

        try {
            QTA newQTA = QTA::create(getPerc25(), getPerc50(), getPerc75()
    , getCdfMax(), true);
            observable->setQTA(newQTA);
        } catch (std::exception &e) {
            QMessageBox::warning(this, "Error", e.what());
        }
}

double QTAInputWidget::getPerc25() const
{
        return perc25Edit->text().toDouble();
}
double QTAInputWidget::getPerc50() const
{
        return perc50Edit->text().toDouble();
}
double QTAInputWidget::getPerc75() const
{
```

```
    return perc75Edit->text().toDouble();
}
double QTAInputWidget::getCdfMax() const
{
    return cdfMaxEdit->text().toDouble();
}

QString QTAInputWidget::getSelectedObservable() const
{
    return observableComboBox->currentText();
}
```

F.2.13 Sidebar.cpp

This widget is a tab where the user can handle the system, add/remove plots and change the sampling rate.

```
#include "Sidebar.h"

#include "NewPlotList.h"
#include "SamplingRateWidget.h"
#include "SystemCreationWidget.h"
#include <QBoxLayout>
#include <QFileDialog>
#include <QLabel>
#include <QMessageBox>
#include <iostream>
#include <qboxlayout.h>
#include <qlabel.h>
#include <qlogging.h>
#include <qnamespace.h>
#include <qpushbutton.h>
#include <qsplitter.h>
#include <qtextedit.h>

Sidebar::Sidebar(QWidget *parent)
    : QWidget(parent)
{
    // Main layout setup
    layout = new QVBoxLayout(this);
    layout->setContentsMargins(0, 0, 0, 0);

    mainSplitter = new QSplitter(Qt::Vertical, this);

    // System creation section
    systemCreationWidget = new SystemCreationWidget(this);
    mainSplitter->addWidget(systemCreationWidget);

    // New plot section
    newPlotListWidget = new QWidget(this);
    newPlotListLayout = new QVBoxLayout(newPlotListWidget);
    newPlotListLayout->setContentsMargins(5, 5, 5, 5);

    newPlotLabel = new QLabel("Select probes for a new plot:", this);
```

```
    newPlotLabel->setSizePolicy(QSizePolicy::Preferred, QSizePolicy::Fixed);
    addNewPlotButton = new QPushButton("Add plot");
    newPlotList = new NewPlotList(this);
    connect(addNewPlotButton, &QPushButton::clicked, this, &Sidebar::onAddPlotClicked);

    newPlotListLayout->addWidget(newPlotLabel);
    newPlotListLayout->addWidget(newPlotList);
    newPlotListLayout->addWidget(addNewPlotButton);

    mainSplitter->addWidget(newPlotListWidget);

    // Sampling rate section
    samplingRateWidget = new SamplingRateWidget(this);
    mainSplitter->addWidget(samplingRateWidget);

    connect(samplingRateWidget, &SamplingRateWidget::onSamplingRateChanged, this, &Sidebar::handleSamplingRateChanged);

    // Current plot section (initially hidden)
    currentPlotWidget = new QWidget(this);
    currentPlotLayout = new QVBoxLayout(currentPlotWidget);
    currentPlotLabel = new QLabel("Modify current plot:", this);
    currentPlotLabel->setSizePolicy(QSizePolicy::Preferred, QSizePolicy::Fixed);
    currentPlotLabel->hide();

    currentPlotLayout->addWidget(currentPlotLabel);
    mainSplitter->addWidget(currentPlotWidget);

    layout->addWidget(mainSplitter);
}

/**
 * @brief Sets the current plot list widget to display.
 * @param plotList The DQPlotList widget to show in the current plot section.
 */
void Sidebar::setCurrentPlotList(DQPlotList *plotList)
{
    if (currentPlotList == plotList) {
        return;
    }

    if (currentPlotList) {
        layout->removeWidget(currentPlotList);
        currentPlotList->hide();
    }

    if (plotList) {
        currentPlotList = plotList;
        layout->addWidget(currentPlotList);
        currentPlotList->show();
        currentPlotLabel->show();
    }
```

```
}

/**
 * @brief Handles sampling rate change events from the
   SamplingRateWidget.
 * @param ms The new sampling rate in milliseconds.
 */
void Sidebar::handleSamplingRateChanged(int ms)
{
    Q_EMIT onSamplingRateChanged(ms);
}

/**
 * @brief Hides the current plot management section.
 */
void Sidebar::hideCurrentPlot()
{
    if (currentPlotList) {
        layout->removeWidget(currentPlotList);
        currentPlotList = nullptr;
    }
    currentPlotLabel->hide();
}

/**
 * @brief Clears new plot selection after plot creation.
 */
void Sidebar::clearOnAdd()
{
    newPlotList->clearSelection();
}

/**
 * @brief Handles the "Add plot" button click event.
 * Validates selection and emits addPlotClicked() signal.
 */
void Sidebar::onAddPlotClicked()
{
    auto selectedItems = newPlotList->getSelectedItems();

    if (selectedItems.empty()) {
        QMessageBox::warning(this, "No Selection", "Please select
   probes before adding a plot.");
        return;
    }

    Q_EMIT addPlotClicked();
}
```

F.2.14 SnapshotViewerWindow.cpp

This is a window to observe a snapshot from the triggers tab.

```
/**
 * @file SnapshotViewerWindow.cpp
```

```
 * @brief Implementation of the SnapshotViewerWindow class.
 */

#include "SnapshotViewerWindow.h"

#include <QComboBox>
#include <QHBoxLayout>
#include <QLabel>
#include <QSlider>
#include <QVBoxLayout>
#include <QtCharts/QChartView>
#include <QtCharts/QLineSeries>
#include <QtCharts/QValueAxis>
#include <QtConcurrent>
#include <qnamespace.h>

/**
 * @brief Constructs a new SnapshotViewerWindow.
 */
SnapshotViewerWindow::SnapshotViewerWindow(std::vector<Snapshot> & snapshotList, QWidget *parent)
    : QWidget(parent)
    , observableSelector(new QComboBox(this))
    , timeSlider(new QSlider(Qt::Horizontal, this))
    , timeLabel(new QLabel(this))
    , chartView(new QChartView(new QChart(), this))
{
    auto *mainLayout = new QHBoxLayout(this);

    // Chart area
    chartView->setRenderHint(QPainter::Antialiasing);
    mainLayout->addWidget(chartView, 3); // 3/4 of space

    // Controls
    auto *controlLayout = new QVBoxLayout(this);
    controlLayout->addWidget(new QLabel("Select Observable:"));
    controlLayout->addWidget(observableSelector);

    controlLayout->addWidget(new QLabel("Select Time:"));
    controlLayout->addWidget(timeSlider);
    controlLayout->addWidget(timeLabel);
    controlLayout->addStretch();
    mainLayout->addLayout(controlLayout, 1); // 1/4 of space

    connect(observableSelector, &QComboBox::currentTextChanged, this, &SnapshotViewerWindow::onObservableChanged);
    connect(timeSlider, &QSlider::valueChanged, this, &SnapshotViewerWindow::onTimeSliderChanged);

    setSnapshots(snapshotList);
}

/**
 * @brief Sets the snapshot data and updates the UI.
 */
void SnapshotViewerWindow::setSnapshots(std::vector<Snapshot> &
```

```
    snapshotList)
{
    snapshots.clear();
    observableSelector->clear();

    for (auto &s : snapshotList) {
        snapshots[s.getName()] = s;
        observableSelector->addItem(QString::fromStdString(s.getName()
    ));
    }

    if (!snapshots.empty()) {
        observableSelector->setCurrentIndex(0);
        onObservableChanged(observableSelector->currentText());
    }
}

/**
 * @brief Handles the change of observable by updating the slider
    range and triggering a plot update.
 */
void SnapshotViewerWindow::onObservableChanged(const QString &name)
{
    currentObservable = name.toStdString();

    const auto &snapshot = snapshots.at(currentObservable);
    int count = static_cast<int>(snapshot.getObservedSize());
    timeSlider->setRange(0, std::max(0, count - 1));
    timeSlider->setValue(0);
    onTimeSliderChanged(0);
}

/**
 * @brief Handles changes in the time slider by updating the chart.
 */
void SnapshotViewerWindow::onTimeSliderChanged(int value)
{
    if (snapshots.empty() || !snapshots.count(currentObservable))
        return;

    const auto &snapshot = snapshots.at(currentObservable);

    if (snapshot.getObservedSize() == 0 || value >= static_cast<int>(
    snapshot.getObservedSize()))
        return;

    const auto obs = snapshot.getObservedDeltaQs()[value];
    const auto calc = snapshot.getCalculatedSize() > value ? std::
    optional<DeltaQRepr>(snapshot.getCalculatedDeltaQs()[value]) : std
    ::nullopt;
    const auto qta = snapshot.getQTAs()[value];
    auto time = obs.time;

    qint64 msTime = time / 1000000;
    QDateTime timestamp = QDateTime::fromMSecsSinceEpoch(msTime);
    timeLabel->setText(QString("Snapshot at: %1").arg(timestamp.
```

```
    toString()));

     auto ret = QtConcurrent::run([=]() {
         int bins = obs.deltaQ.getBins();
         double binWidth = obs.deltaQ.getBinWidth();

         QVector<QPointF> obsMean, obsLower, obsUpper, obsCdf, qtaData;
         obsCdf.reserve(bins + 1);
         obsMean.reserve(bins + 1);
         obsLower.reserve(bins + 1);
         obsUpper.reserve(bins + 1);

         obsCdf.append(QPointF(0, 0));
         obsMean.append(QPointF(0, 0));
         obsLower.append(QPointF(0, 0));
         obsUpper.append(QPointF(0, 0));

         for (int i = 0; i < bins; ++i) {
             double x = binWidth * (i + 1);
             obsCdf.append(QPointF(x, obs.deltaQ.cdfAt(i)));
             obsLower.append(QPointF(x, obs.bounds[i].lowerBound));
             obsUpper.append(QPointF(x, obs.bounds[i].upperBound));
             obsMean.append(QPointF(x, obs.bounds[i].mean));
         }

         if (qta.defined) {
             double maxDelay = bins * binWidth;
             qtaData = {
                 {qta.perc_25, 0         },
                 {qta.perc_25, 0.25      },
                 {qta.perc_50, 0.25      },
                 {qta.perc_50, 0.5       },
                 {qta.perc_75, 0.5       },
                 {qta.perc_75, 0.75      },
                 {maxDelay,    0.75      },
                 {maxDelay,    qta.cdfMax}
             };
         }

         QVector<QPointF> calcCdf, calcMean, calcLower, calcUpper;
         if (calc) {
             int cbins = calc->deltaQ.getBins();
             double cbinWidth = calc->deltaQ.getBinWidth();
             calcCdf.reserve(cbins + 1);
             calcMean.reserve(cbins + 1);
             calcLower.reserve(cbins + 1);
             calcUpper.reserve(cbins + 1);

             calcCdf.append(QPointF(0, 0));
             calcMean.append(QPointF(0, 0));
             calcLower.append(QPointF(0, 0));
             calcUpper.append(QPointF(0, 0));

             for (int i = 0; i < cbins; ++i) {
                 double x = cbinWidth * (i + 1);
                 calcCdf.append(QPointF(x, calc->deltaQ.cdfAt(i)));
```

```
                calcLower.append(QPointF(x, calc->bounds[i].lowerBound));
                calcUpper.append(QPointF(x, calc->bounds[i].upperBound));
                calcMean.append(QPointF(x, calc->bounds[i].mean));
            }
        }

        QMetaObject::invokeMethod(this, [=]() {
            QChart *chart = chartView->chart();

            // Clear previous content
            chart->removeAllSeries();
            const auto axes = chart->axes();
            for (QAbstractAxis *axis : axes) {
                chart->removeAxis(axis);
                axis->deleteLater(); // Safe axis cleanup
            }

            chart->setTitle(QString::fromStdString(currentObservable) + QString(" - Time Index: %1").arg(value));

            auto addSeries = [&](const QVector<QPointF> &data, const QString &name, const QColor &color) {
                QLineSeries *series = new QLineSeries();
                series->setName(name);
                series->append(data);
                series->setColor(color);
                chart->addSeries(series);
                return series;
            };

            addSeries(obsCdf, "Observed", Qt::blue);
            addSeries(obsMean, "Obs Mean", Qt::yellow);
            addSeries(obsLower, "Obs Lower", Qt::green);
            addSeries(obsUpper, "Obs Upper", Qt::darkGreen);
            addSeries(qtaData, "QTA", Qt::darkBlue);

            if (!calcCdf.isEmpty()) {
                addSeries(calcCdf, "Calculated", Qt::red);
                addSeries(calcMean, "Calc Mean", Qt::darkYellow);
                addSeries(calcLower, "Calc Lower", Qt::magenta);
                addSeries(calcUpper, "Calc Upper", Qt::darkMagenta);
            }

            auto *axisX = new QValueAxis();
            axisX->setTitleText("Delay (s)");
            chart->addAxis(axisX, Qt::AlignBottom);

            auto *axisY = new QValueAxis();
            axisY->setTitleText("ΔQ(x)");
            axisY->setRange(0, 1);
            chart->addAxis(axisY, Qt::AlignLeft);

            for (auto *series : chart->series()) {
                series->attachAxis(axisX);
```

```
                series->attachAxis(axisY);
            }
        });
    });
}
```

F.2.15 StubControlWidget.cpp

This widget allows to open the server on the IP and Port defined by the user and to connect to the adapter on the IP and port specified by the user.

```
#include "StubControlWidget.h"

StubControlWidget::StubControlWidget(QWidget *parent)
    : QWidget(parent)
{
    mainLayout = new QVBoxLayout(this);

    // Erlang Control Group
    QGroupBox *erlangGroup = new QGroupBox("Erlang Wrapper Control",
   this);
    QVBoxLayout *erlangMainLayout = new QVBoxLayout(erlangGroup);

    // First row: IP and Port
    QHBoxLayout *erlangIpPortLayout = new QHBoxLayout();
    erlangIpPortLayout->addWidget(new QLabel("IP:"));
    erlangReceiverIpEdit = new QLineEdit("127.0.0.1", this);
    erlangIpPortLayout->addWidget(erlangReceiverIpEdit);
    erlangIpPortLayout->addWidget(new QLabel("Port:"));
    erlangReceiverPortEdit = new QLineEdit("8081", this);
    erlangIpPortLayout->addWidget(erlangReceiverPortEdit);
    erlangMainLayout->addLayout(erlangIpPortLayout);

    // Second row: Set Endpoint Button
    setErlangEndpointButton = new QPushButton("Set Adapter Endpoint",
   this);
    erlangMainLayout->addWidget(setErlangEndpointButton, 0, Qt::
   AlignLeft);

    // Third row: Start/Stop Wrapper
    QHBoxLayout *erlangButtonsLayout = new QHBoxLayout();
    stopErlangButton = new QPushButton("Stop Adapter", this);
    startErlangButton = new QPushButton("Start Adapter", this);
    erlangButtonsLayout->addWidget(stopErlangButton);
    erlangButtonsLayout->addWidget(startErlangButton);
    erlangMainLayout->addLayout(erlangButtonsLayout);

    // Server Control Group
    QGroupBox *serverGroup = new QGroupBox("C++ Server Control", this)
   ;
    QVBoxLayout *serverMainLayout = new QVBoxLayout(serverGroup);

    // First row: IP and Port
    QHBoxLayout *serverIpPortLayout = new QHBoxLayout();
    serverIpPortLayout->addWidget(new QLabel("IP:"));
```

```
    serverIpEdit = new QLineEdit("0.0.0.0", this);
    serverIpPortLayout->addWidget(serverIpEdit);
    serverIpPortLayout->addWidget(new QLabel("Port:"));
    serverPortEdit = new QLineEdit("8080", this);
    serverIpPortLayout->addWidget(serverPortEdit);
    serverMainLayout->addLayout(serverIpPortLayout);

    // Second row: Start/Stop Server
    QHBoxLayout *serverButtonsLayout = new QHBoxLayout();
    startServerButton = new QPushButton("Start Oscilloscope Server",
   this);
    stopServerButton = new QPushButton("Stop Oscilloscope Server",
   this);
    serverButtonsLayout->addWidget(startServerButton);
    serverButtonsLayout->addWidget(stopServerButton);
    serverMainLayout->addLayout(serverButtonsLayout);

    // Add to main layout
    mainLayout->addWidget(erlangGroup);
    mainLayout->addWidget(serverGroup);
    mainLayout->addStretch();
    setLayout(mainLayout);
    setSizePolicy(QSizePolicy::Preferred, QSizePolicy::Maximum);

    // Connect signals
    connect(startErlangButton, &QPushButton::clicked, this, &
   StubControlWidget::onStartErlangClicked);
    connect(stopErlangButton, &QPushButton::clicked, this, &
   StubControlWidget::onStopErlangClicked);
    connect(startServerButton, &QPushButton::clicked, this, &
   StubControlWidget::onStartServerClicked);
    connect(stopServerButton, &QPushButton::clicked, this, &
   StubControlWidget::onStopServerClicked);
    connect(setErlangEndpointButton, &QPushButton::clicked, this, &
   StubControlWidget::onSetErlangEndpointClicked);
}

void StubControlWidget::onStartErlangClicked()
{
    Application::getInstance().setStubRunning(true);
}

void StubControlWidget::onStopErlangClicked()
{
    Application::getInstance().setStubRunning(false);
}

void StubControlWidget::onStartServerClicked()
{
    QString ip = serverIpEdit->text();
    int port = serverPortEdit->text().toInt();

    if (Application::getInstance().startCppServer(ip.toStdString(),
   port)) {
        startServerButton->setEnabled(false);
        stopServerButton->setEnabled(true);
```

```
    }
}

void StubControlWidget::onStopServerClicked()
{
    Application::getInstance().stopCppServer();
    startServerButton->setEnabled(true);
    stopServerButton->setEnabled(false);
}

void StubControlWidget::onSetErlangEndpointClicked()
{
    QString ip = erlangReceiverIpEdit->text();
    int port = erlangReceiverPortEdit->text().toInt();

    Application::getInstance().setErlangEndpoint(ip.toStdString(),
   port);
}
```

F.2.16 SystemCreationWidget.cpp

This widget allows the creation/update of a system, loading an already existing one or saving one.

```
#include "SystemCreationWidget.h"
#include "../Application.h"
#include "../parser/SystemParserInterface.h"
#include <QFileDialog>
#include <QMessageBox>
#include <fstream>

/**
 * @brief Constructs the SystemCreationWidget and sets up the UI.
 * @param parent Parent QWidget.
 */
SystemCreationWidget::SystemCreationWidget(QWidget *parent)
    : QWidget(parent)
{
    mainLayout = new QVBoxLayout(this);
    mainLayout->setAlignment(Qt::AlignTop);
    mainLayout->setSpacing(10);
    mainLayout->setContentsMargins(0, 0, 0, 0);

    systemLabel = new QLabel("Create or edit your system here");
    systemTextEdit = new QTextEdit();

    buttonLayout = new QHBoxLayout();
    updateSystemButton = new QPushButton("Create or edit system");
    saveSystemButton = new QPushButton("Save system to");
    loadSystemButton = new QPushButton("Load system from");

    buttonLayout->addWidget(updateSystemButton);
    buttonLayout->addWidget(saveSystemButton);
    buttonLayout->addWidget(loadSystemButton);

```

```
    mainLayout->addWidget(systemLabel);
    mainLayout->addWidget(systemTextEdit);
    mainLayout->addLayout(buttonLayout);

    connect(updateSystemButton, &QPushButton::clicked, this, &
   SystemCreationWidget::onUpdateSystem);
    connect(saveSystemButton, &QPushButton::clicked, this, &
   SystemCreationWidget::saveSystemTo);
    connect(loadSystemButton, &QPushButton::clicked, this, &
   SystemCreationWidget::loadSystem);
}

/**
 * @brief Gets the text currently in the system text editor.
 * @return System text as a std::string.
 */
std::string SystemCreationWidget::getSystemText() const
{
    return systemTextEdit->toPlainText().toStdString();
}

/**
 * @brief Sets the content of the system text editor.
 * @param text The new system text.
 */
void SystemCreationWidget::setSystemText(const std::string &text)
{
    systemTextEdit->setText(QString::fromStdString(text));
}

/**
 * @brief Parses the system text and updates the application system if
     valid.
 */
void SystemCreationWidget::onUpdateSystem()
{
    std::string text = getSystemText();

    try {
        auto system = SystemParserInterface::parseString(text);
        if (system.has_value()) {
            Application::getInstance().setSystem(system.value());
            system->setSystemDefinitionText(text);
            Q_EMIT systemUpdated();
        }
    } catch (const std::exception &e) {
        QMessageBox::critical(this, "Parsing error", e.what());
    }
}

/**
 * @brief Opens a dialog to save the current system to a file.
 */
void SystemCreationWidget::saveSystemTo()
{
    QFileDialog dialog(this);
```

```
    dialog.setFileMode(QFileDialog::AnyFile);
    QString filename = dialog.getSaveFileName(this, "Save file", "", "All files (* *.dq)");

    if (!filename.isEmpty()) {
        std::string systemText = getSystemText();
        auto system = SystemParserInterface::parseString(systemText);

        if (system.has_value()) {
            std::ofstream outFile(filename.toStdString());
            if (outFile.is_open()) {
                outFile << systemText;
                outFile.close();
                QMessageBox::information(this, "Success", "File saved successfully.");
                Q_EMIT systemSaved();
            } else {
                QMessageBox::critical(this, "Error", "Could not open file for writing.");
            }
        } else {
            QMessageBox::warning(this, "Error", "System parsing failed. File not saved.");
        }
    }
}

/**
 * @brief Opens a dialog to load a system from a file, parses it, and updates the editor.
 */
void SystemCreationWidget::loadSystem()
{
    QFileDialog dialog(this);
    std::string filename = dialog.getOpenFileName(this, "Select file", " ", "All files (* *.dq)").toStdString();

    auto system = SystemParserInterface::parseFile(filename);
    if (system.has_value()) {
        Application::getInstance().setSystem(system.value());
        std::ifstream file(filename);
        std::string str;
        std::string fileContents;
        while (std::getline(file, str)) {
            fileContents += str;
            fileContents.push_back('\n');
        }

        system->setSystemDefinitionText(fileContents);
        setSystemText(fileContents);
        Q_EMIT systemLoaded();
    }
}
```

F.2.17 TriggersTab.cpp

This tab holds the widgets to set/remove triggers and view fired ones.

```
#include "TriggersTab.h"
#include "SnapshotViewerWindow.h"
#include <QDateTime>
#include <QLabel>
#include <QString>
#include <QTimer>
#include <cstdlib>
#include <sstream>
#include <string>
#include <unordered_set>

TriggersTab::TriggersTab(QWidget *parent)
    : QWidget(parent)
{
    // Main layout setup
    mainLayout = new QVBoxLayout(this);
    formLayout = new QFormLayout();

    // Observable selection dropdown
    observableComboBox = new QComboBox(this);
    connect(observableComboBox, &QComboBox::currentTextChanged, this,
   &TriggersTab::onObservableChanged);
    formLayout->addRow("Probe:", observableComboBox);

    // Sample limit controls
    sampleLimitCheckBox = new QCheckBox("Sample Limit >", this);
    sampleLimitSpinBox = new QSpinBox(this);
    sampleLimitSpinBox->setRange(1, 100000);
    sampleLimitSpinBox->setValue(sampleLimitThreshold);

    sampleLimitLayout = new QHBoxLayout();
    sampleLimitLayout->addWidget(sampleLimitCheckBox);
    sampleLimitLayout->addWidget(sampleLimitSpinBox);

    sampleLimitWidget = new QWidget(this);
    sampleLimitWidget->setLayout(sampleLimitLayout);
    formLayout->addRow(sampleLimitWidget);

    // QTA bounds violation checkbox
    qtaBoundsCheckBox = new QCheckBox("QTA Bound Violation", this);
    formLayout->addRow(qtaBoundsCheckBox);

    // Connect signals
    connect(sampleLimitCheckBox, &QCheckBox::checkStateChanged, this,
   &TriggersTab::onTriggerChanged);
    connect(sampleLimitSpinBox, QOverload<int>::of(&QSpinBox::
   valueChanged), this, &TriggersTab::onTriggerChanged);
    connect(qtaBoundsCheckBox, &QCheckBox::checkStateChanged, this, &
   TriggersTab::onTriggerChanged);

    mainLayout->addLayout(formLayout);

```

```
    // Triggered events list
    triggeredList = new QListWidget(this);
    triggeredList->setSelectionMode(QAbstractItemView::SingleSelection
   );
    triggeredList->setMinimumHeight(150);
    connect(triggeredList, &QListWidget::itemClicked, this, &
   TriggersTab::onTriggeredItemClicked);

    mainLayout->addWidget(new QLabel("Triggered Snapshots:"));
    mainLayout->addWidget(triggeredList);

    // Set up system observer
    Application::getInstance().addObserver([this]() { this->
   populateObservables(); });

    populateObservables();
}

TriggersTab::~TriggersTab()
{
    delete mainLayout;
}

/**
 * @brief Populates the observable dropdown with available probes and
   outcomes.
 */
void TriggersTab::populateObservables()
{
    try {
        auto system = Application::getInstance().getSystem();
        observableComboBox->clear();
        for (const auto &[name, obs] : system->getObservables()) {
            if (obs) {
                observableComboBox->addItem(QString::fromStdString(
   name));
            }
        }
    } catch (std::exception &) {
        return;
    }
}

/**
 * @brief Handles observable selection changes.
 * @param name The newly selected observable name (unused, signal
   requires parameter).
 */
void TriggersTab::onObservableChanged(const QString &)
{
    updateCheckboxStates();
}

/**
 * @brief Updates triggers when conditions change.
 */
```

```
void TriggersTab::onTriggerChanged()
{
    try {
        auto observable = getCurrentObservable();
        std::string name = observable->getName();

        // Update sample limit trigger
        if (sampleLimitCheckBox->isChecked()) {
            observable->addTrigger(
                TriggerType::SampleLimit,
                TriggerDefs::Conditions::SampleLimit(sampleLimitSpinBox->value()),
                [this, name](const DeltaQ &, const QTA &, std::uint64_t time) {
                    this->captureSnapshots(time, name);
                },
                true,
                sampleLimitSpinBox->value());
        } else {
            observable->removeTrigger(TriggerType::SampleLimit);
        }

        // Update QTA bounds trigger
        if (qtaBoundsCheckBox->isChecked()) {
            observable->addTrigger(
                TriggerType::QTAViolation,
                TriggerDefs::Conditions::QTABounds(),
                [this, name](const DeltaQ &, const QTA &, std::uint64_t time) {
                    this->captureSnapshots(time, name);
                },
                true,
                std::nullopt);
        } else {
            observable->removeTrigger(TriggerType::QTAViolation);
        }
    } catch (std::exception &) {
        return;
    }
}

/**
 * @brief Captures snapshots when triggers are activated.
 * @param time The timestamp of the trigger event.
 * @param name The name of the observable that triggered.
 */
void TriggersTab::captureSnapshots(std::uint64_t time, const std::string &name)
{
    auto system = Application::getInstance().getSystem();
    if (!system->isRecording()) {
        system->setRecording(true);
        QMetaObject::invokeMethod(
            this,
            [this, name, system, time]() {
                auto *timer = new QTimer(this);
```

```
                timer->setSingleShot(true);

                connect(timer, &QTimer::timeout, this, [=]() {
                    system->getObservablesSnapshotAt(time);

                    // Format timestamp for display
                    qint64 msTime = time / 1000000;
                    QDateTime timestamp = QDateTime::fromMSecsSinceEpoch(msTime);

                    // Create display string
                    QString timestampStr = timestamp.toString();
                    std::ostringstream oss;
                    oss << "Snapshot at: " << timestampStr.toStdString() << " from " << name;
                    QString snapshotString = QString::fromStdString(oss.str());

                    // Add to triggered list
                    auto *item = new QListWidgetItem(snapshotString);
                    item->setData(Qt::UserRole, static_cast<qulonglong>(time));
                    triggeredList->addItem(item);
                    timer->deleteLater();

                    system->setRecording(false);
                });

                timer->start(5000);
            },
            Qt::QueuedConnection);
    }
}

/**
 * @brief Updates checkbox states based on current triggers.
 */
void TriggersTab::updateCheckboxStates()
{
    try {
        auto observable = getCurrentObservable();
        auto &manager = observable->getTriggerManager();

        // Get active trigger types
        auto all = manager.getAllTriggers();
        std::unordered_set<TriggerType> activeTypes;
        for (const auto &trigger : all) {
            if (trigger.enabled)
                activeTypes.insert(trigger.type);
        }

        // Update UI to match active triggers
        sampleLimitCheckBox->setChecked(activeTypes.count(TriggerType::SampleLimit));
        qtaBoundsCheckBox->setChecked(activeTypes.count(TriggerType::QTAViolation));
```

```cpp

        // Update sample limit value if trigger exists
        for (const auto &t : all) {
            if (t.type == TriggerType::SampleLimit && t.
    sampleLimitValue) {
                sampleLimitSpinBox->setValue(*t.sampleLimitValue);
            }
        }
    } catch (std::exception &) {
        sampleLimitCheckBox->setChecked(false);
        qtaBoundsCheckBox->setChecked(false);
    }
}

/**
 * @brief Gets the currently selected observable.
 * @return current observable.
 * @throws std::runtime_error if system or observable doesn't exist.
 */
std::shared_ptr<Observable> TriggersTab::getCurrentObservable()
{
    auto system = Application::getInstance().getSystem();
    if (!system)
        throw std::runtime_error("System does not exist");

    std::string name = observableComboBox->currentText().toStdString()
    ;
    auto observable = system->getObservable(name);
    if (!observable)
        throw std::runtime_error("Observable does not exist");

    return observable;
}

/**
 * @brief Handles triggered item clicks to show snapshots.
 * @param item The clicked list item containing snapshot data.
 */
void TriggersTab::onTriggeredItemClicked(QListWidgetItem *item)
{
    if (!item)
        return;

    // Retrieve timestamp from item data
    qulonglong timestamp = item->data(Qt::UserRole).toULongLong();
    if (timestamp == 0)
        return;

    // Find and display corresponding snapshot
    auto system = Application::getInstance().getSystem();
    const auto &snapshots = system->getAllSnapshots();
    auto it = snapshots.find(timestamp);
    if (it != snapshots.end()) {
        const auto &snapshotList = it->second;

        auto *viewer = new SnapshotViewerWindow(const_cast<std::vector
```

```
    <Snapshot> &>(snapshotList));
            viewer->setAttribute(Qt::WA_DeleteOnClose);
            viewer->resize(800, 600);
            viewer->show();
    }
}
```

F.3 Outcome diagram

The "diagram" folder contains everything related to outcome diagrams. Due to time related issues, there are some issues with the names. We will explain what each class represents, but it differs from the definitions which are explained in the thesis.

F.3.1 Observable.cpp

The observable class represents a generic "observable" element of the outcome diagram, it is the base class for probes, outcome and operators. In this class one can calculate the observed ΔQ, store the outcome instances (samples), set the parameters, set a QTA, add/remove triggers and get a snapshot. It is what we described throughout the whole paper as a probe.

```
#include "Observable.h"
#include <algorithm>
#include <cmath>
#include <cstdint>
#include <iostream>

#define MAX_DQ 30
Observable::Observable(const std::string &name)
    : observedInterval(50)
    , name(name)
{
    observableSnapshot.setName(name);
}

void Observable::addSample(const Sample &sample)
{
    samples.emplace_back(sample);
    sorted = false;
}

std::vector<Sample> Observable::getSamplesInRange(std::uint64_t
    lowerTime, std::uint64_t upperTime)
{
    std::lock_guard<std::mutex> lock(samplesMutex);
    if (!sorted) {
        std::sort(samples.begin(), samples.end(), [](const Sample &a,
    const Sample &b) { return a.endTime < b.endTime; });
        sorted = true;
    }

    std::vector<Sample> selectedSamples;
```

```

    auto lower = std::lower_bound(samples.begin(), samples.end(), lowerTime, [](const Sample &s, long long time) { return s.endTime < time; });

    auto upper = std::upper_bound(samples.begin(), samples.end(), upperTime, [](long long time, const Sample &s) { return time < s.endTime; });

    for (auto it = lower; it != upper; ++it) {
        selectedSamples.emplace_back(*it);
    }
    samples.erase(std::remove_if(samples.begin(), samples.end(), [upperTime](const Sample &s) { return s.endTime < upperTime; }), samples.end());

    return selectedSamples;
}

DeltaQ Observable::calculateObservedDeltaQ(std::uint64_t timeLowerBound, std::uint64_t timeUpperBound)
{
    auto samplesInRange = getSamplesInRange(timeLowerBound, timeUpperBound);
    if (samplesInRange.empty()) {
        DeltaQ deltaQ = DeltaQ();
        updateSnapshot(timeLowerBound, deltaQ);
        return deltaQ;
    }

    DeltaQ deltaQ {getBinWidth(), samplesInRange, nBins};
    updateSnapshot(timeLowerBound, deltaQ);

    triggerManager.evaluate(deltaQ, qta, timeLowerBound);
    return deltaQ;
}

void Observable::updateSnapshot(uint64_t timeLowerBound, DeltaQ &deltaQ)
{
    if (!(deltaQ == DeltaQ())) {
        observedInterval.addDeltaQ(deltaQ);
        confidenceIntervalHistory.push_back(deltaQ);

        if (confidenceIntervalHistory.size() > MAX_DQ) {
            observedInterval.removeDeltaQ(confidenceIntervalHistory.front());
            confidenceIntervalHistory.pop_front();
        }
    }
    observableSnapshot.addObservedDeltaQ(timeLowerBound, deltaQ, observedInterval.getBounds());
    observableSnapshot.addQTA(timeLowerBound, qta);
    if (!recording && (confidenceIntervalHistory.size() > MAX_DQ)) {
        observableSnapshot.resizeTo(MAX_DQ);
    }
```

```
}

DeltaQ Observable::getObservedDeltaQ(uint64_t timeLowerBound, uint64_t timeUpperBound)
{
    std::lock_guard<std::mutex> lock(observedMutex);
    auto deltaQRepr = observableSnapshot.getObservedDeltaQAtTime(timeLowerBound);
    if (!deltaQRepr.has_value()) {
        calculateObservedDeltaQ(timeLowerBound, timeUpperBound);
        deltaQRepr = observableSnapshot.getObservedDeltaQAtTime(timeLowerBound);
    }
    return deltaQRepr.value().deltaQ;
}

DeltaQRepr Observable::getObservedDeltaQRepr(uint64_t timeLowerBound, uint64_t timeUpperBound)
{
    std::lock_guard<std::mutex> lock(observedMutex);
    auto deltaQRepr = observableSnapshot.getObservedDeltaQAtTime(timeLowerBound);
    if (!deltaQRepr.has_value()) {
        calculateObservedDeltaQ(timeLowerBound, timeUpperBound);
        deltaQRepr = observableSnapshot.getObservedDeltaQAtTime(timeLowerBound);
    }
    return deltaQRepr.value();
}

void Observable::setQTA(const QTA &newQTA)
{
    if (newQTA.perc_25 > maxDelay || newQTA.perc_50 > maxDelay || newQTA.perc_75 > maxDelay)
        throw std::invalid_argument("Percentages should not be bigger than maximum delay " + std::to_string(maxDelay));
    qta = newQTA;
}

void Observable::addTrigger(TriggerType type, TriggerDefs::Condition condition, TriggerDefs::Action action, bool enabled, std::optional<int> sampleLimitVal)
{
    auto &tm = triggerManager;
    tm.addTrigger(type, condition, action, enabled);
    auto &trigger = tm.getTriggersByType(type).back();
    trigger.sampleLimitValue = sampleLimitVal;
}

void Observable::removeTrigger(TriggerType type)
{
    triggerManager.removeTriggersByType(type);
}

void Observable::setRecording(bool isRecording)
{
```

```
    if (recording && !isRecording) {
        recording = isRecording;
        observableSnapshot.resizeTo(30); // FIXME magic numb
    }
    recording = isRecording;
}

Snapshot Observable::getSnapshot()
{
    return observableSnapshot;
}

double Observable::setNewParameters(int newExp, int newNBins)
{
    std::lock_guard<decltype(paramMutex)> lock(paramMutex);

    bool binsChanged = (newNBins != nBins);
    bool expChanged = (newExp != deltaTExp);

    nBins = newNBins;
    deltaTExp = newExp;

    maxDelay = DELTA_T_BASE * std::pow(2, deltaTExp) * nBins;

    if (qta.perc_25 > maxDelay || qta.perc_50 > maxDelay || qta.
   perc_75 > maxDelay) {
        qta = QTA::create(0, 0, 0, qta.cdfMax, false);
    }

    if (binsChanged || expChanged) {
        observedInterval.setNumBins(nBins);
        confidenceIntervalHistory.clear();
    }

    return maxDelay;
}

std::string Observable::getName() const &
{
    return name;
}
```

F.3.2 Operator.cpp

This class represent a generic operator, it can be either an FTF, ATF or PC. It allows calculating the "calculated ΔQ".

```
#include "Operator.h"

#include <numeric>

#include "../maths/DeltaQOperations.h"
#include "Observable.h"
#include "OperatorType.h"
#include <cmath>
```

```cpp
#include <limits>
#define OP_EPSILON std::numeric_limits<double>::epsilon()

Operator::Operator(const std::string &name, OperatorType type)
    : Observable(name)
    , type(type)
    , calculatedInterval(50)
{
}

Operator::~Operator() = default;

void Operator::setProbabilities(const std::vector<double> &probs)
{
    if (type != OperatorType::PRB) {
        throw std::invalid_argument("Only probabilistic operators
    accept probabilities");
    }

    double result = std::reduce(probs.begin(), probs.end());

    if (std::fabs(1 - result) > OP_EPSILON)
        throw std::logic_error("Result should approximate to 1");

    probabilities = probs;
}

DeltaQ Operator::calculateCalculatedDeltaQ(uint64_t timeLowerBound,
    uint64_t timeUpperBound)
{
    if (observableSnapshot.getCalculatedDeltaQAtTime(timeLowerBound).
    has_value()) {
        return observableSnapshot.getCalculatedDeltaQAtTime(
    timeLowerBound).value().deltaQ;
    }
    std::vector<DeltaQ> deltaQs;

    // Get DeltaQ for all children
    for (auto &childrenLinks : causalLinks) {
        std::vector<DeltaQ> childrenDeltaQs;

        // For each children in causal link, get their DeltaQ
        for (auto &component : childrenLinks) {
            childrenDeltaQs.push_back(component->getObservedDeltaQ(
    timeLowerBound, timeUpperBound));
        }
        // Get the convolution of the components in a children
        deltaQs.push_back(convolveN(childrenDeltaQs));
    }
    DeltaQ result;

    // Choose appropriate operation
    if (type == OperatorType::FTF)
        result = firstToFinish(deltaQs);
    else if (type == OperatorType::PRB)
        result = probabilisticChoice(probabilities, deltaQs);
```

```
    else
        result = allToFinish(deltaQs);

    int calculatedBins = result.getBins();

    if (calculatedBins != calculatedInterval.getBins()) {
        calculatedInterval.setNumBins(calculatedBins);
        calculatedDeltaQHistory.clear();
    }

    calculatedInterval.addDeltaQ(result);
    calculatedDeltaQHistory.push_back(result);

    if (calculatedDeltaQHistory.size() > MAX_DQ) {
        calculatedInterval.removeDeltaQ(calculatedDeltaQHistory.front());
        calculatedDeltaQHistory.pop_front();
    }

    observableSnapshot.addCalculatedDeltaQ(timeLowerBound, result, calculatedInterval.getBounds());

    if (!recording && calculatedDeltaQHistory.size() > MAX_DQ) {
        observableSnapshot.removeOldestCalculatedDeltaQ();
    }
    return result;
}

DeltaQRepr Operator::getCalculatedDeltaQRepr(uint64_t timeLowerBound, uint64_t timeUpperBound)
{
    std::lock_guard<std::mutex> lock(calcMutex);
    auto deltaQRepr = observableSnapshot.getCalculatedDeltaQAtTime(timeLowerBound);
    if (!deltaQRepr.has_value()) {
        calculateCalculatedDeltaQ(timeLowerBound, timeUpperBound);
        deltaQRepr = observableSnapshot.getCalculatedDeltaQAtTime(timeLowerBound);
    }
    return deltaQRepr.value();
}
```

F.3.3 Outcome.cpp

This class represents a simple outcome.

```
#include "Outcome.h"

Outcome::Outcome(const std::string &name)
    : Observable(name)
{
}

Outcome::~Outcome() = default;
```

F.3.4 Probe.cpp

This class represents what we described as "sub-outcome diagram". As the operator class, it allows calculating the "calculated ΔQ".

```cpp
#include "Probe.h"
#include "../maths/ConfidenceInterval.h"
#include "../maths/DeltaQOperations.h"
#include <iostream>
#include <memory>
#include <mutex>
Probe::Probe(const std::string &name)
     : Observable(name)
     , calculatedInterval(50)
{
}

Probe::Probe(const std::string &name, std::vector<std::shared_ptr<
    Observable>> causalLinks)
     : Observable(name)
     , causalLinks(causalLinks)
     , calculatedInterval(50)
{
}

Probe::~Probe() = default;

DeltaQ Probe::calculateCalculatedDeltaQ(uint64_t timeLowerBound,
    uint64_t timeUpperBound)
{
     if (observableSnapshot.getCalculatedDeltaQAtTime(timeLowerBound).
    has_value()) {
          return observableSnapshot.getCalculatedDeltaQAtTime(
    timeLowerBound).value().deltaQ;
     }
     std::vector<DeltaQ> deltaQs;
     for (const auto &component : causalLinks) {

          deltaQs.push_back(component->getObservedDeltaQ(timeLowerBound,
     timeUpperBound));
     }

     DeltaQ result = convolveN(deltaQs);
     int calculatedBins = result.getBins();

     if (calculatedBins != calculatedInterval.getBins()) {
          calculatedInterval.setNumBins(calculatedBins);
          calculatedDeltaQHistory.clear();
     }

     calculatedInterval.addDeltaQ(result);
     calculatedDeltaQHistory.push_back(result);

     if (calculatedDeltaQHistory.size() > MAX_DQ) {
          calculatedInterval.removeDeltaQ(calculatedDeltaQHistory.front
    ());
```

```cpp
        calculatedDeltaQHistory.pop_front();
    }

    observableSnapshot.addCalculatedDeltaQ(timeLowerBound, result,
   calculatedInterval.getBounds());

    if (!recording && calculatedDeltaQHistory.size() > MAX_DQ) {
        observableSnapshot.removeOldestCalculatedDeltaQ();
    }
    return result;
}
std::vector<Bound> Probe::getBounds() const
{
    return observedInterval.getBounds();
}

std::vector<Bound> Probe::getObservedBounds() const
{
    return observedInterval.getBounds();
}

std::vector<Bound> Probe::getCalculatedBounds() const
{
    return calculatedInterval.getBounds();
}

DeltaQRepr Probe::getCalculatedDeltaQRepr(uint64_t timeLowerBound,
    uint64_t timeUpperBound)
{
    std::lock_guard<std::mutex> lock(calcMutex);
    auto deltaQRepr = observableSnapshot.getCalculatedDeltaQAtTime(
   timeLowerBound);
    if (!deltaQRepr.has_value()) {
        calculateCalculatedDeltaQ(timeLowerBound, timeUpperBound);
        deltaQRepr = observableSnapshot.getCalculatedDeltaQAtTime(
   timeLowerBound);
    }
    return deltaQRepr.value();
}
```

F.3.5 System.cpp

This class represents the system, the whole outcome diagram. It coordinates the various parts of the outcome diagram.

```cpp
/**
 * @author Francesco Nieri
 * @date 26/10/2024
 * Class representing a DeltaQ system
 */

#include "System.h"
#include "../Application.h"
#include <utility>

```

```
void System::setOutcomes(std::unordered_map<std::string, std::
    shared_ptr<Outcome>> outcomesMap)
{
    outcomes = outcomesMap;
    for (auto &[name, outcome] : outcomes) {
        observables[name] = outcome;
    }
}

void System::setProbes(std::unordered_map<std::string, std::shared_ptr
    <Probe>> probesMap)
{
    probes = probesMap;
    for (auto &[name, probe] : probes) {
        observables[name] = probe;
    }
}

void System::setOperators(std::unordered_map<std::string, std::
    shared_ptr<Operator>> operatorsMap)
{
    operators = operatorsMap;
    for (auto &[name, op] : operators) {
        observables[name] = op;
    }
}
std::shared_ptr<Outcome> System::getOutcome(const std::string &name)
{
    return outcomes[name];
}

void System::setObservableParameters(std::string &componentName, int
    exponent, int numBins)
{
    if (auto it = observables.find(componentName); it != observables.
    end()) {
        double maxDelay = it->second->setNewParameters(exponent,
    numBins);
        Application::getInstance().sendDelayChange(componentName,
    maxDelay * 1000);
    }
}

void System::addSample(std::string &componentName, Sample &sample)
{
    if (auto it = observables.find(componentName); it != observables.
    end()) {
        it->second->addSample(sample);
    }
}

DeltaQ System::calculateDeltaQ()
{
    return DeltaQ();
}

```

```
[[nodiscard]] std::unordered_map<std::string, std::shared_ptr<Outcome
    >> &System::getOutcomes()
{
    return outcomes;
}

[[nodiscard]] std::unordered_map<std::string, std::shared_ptr<Operator
    >> &System::getOperators()
{
    return operators;
}

[[nodiscard]] std::unordered_map<std::string, std::shared_ptr<Probe>>
    &System::getProbes()
{
    return probes;
}

[[nodiscard]] std::unordered_map<std::string, std::shared_ptr<
    Observable>> &System::getObservables()
{
    return observables;
}

bool System::hasOutcome(const std::string &name)
{
    return outcomes.find(name) != outcomes.end();
}

bool System::hasProbe(const std::string &name)
{
    return probes.find(name) != probes.end();
}

bool System::hasOperator(const std::string &name)
{
    return operators.find(name) != operators.end();
}

std::shared_ptr<Operator> System::getOperator(const std::string &name)
{
    return operators[name];
}

std::shared_ptr<Probe> System::getProbe(const std::string &name)
{
    return probes[name];
}

std::vector<std::string> System::getAllComponentsName()
{
    std::vector<std::string> names;
    names.reserve(observables.size());

    for (auto &[name, obs] : observables) {
        if (obs)
```

```
            names.emplace_back(name);
    }

    return names;
}

void System::setSystemDefinitionText(std::string &text)
{
    systemDefinitionText = text;
}

std::string System::getSystemDefinitionText()
{
    return systemDefinitionText;
}

std::shared_ptr<Observable> System::getObservable(const std::string &name)
{
    return observables[name];
}

void System::setRecording(bool isRecording)
{
    if (recordingTrigger && isRecording) {
        return;
    }

    recordingTrigger = isRecording;
    for (auto &obs : observables) {
        if (obs.second) {
            obs.second->setRecording(isRecording);
        }
    }
}

bool System::isRecording() const
{
    return recordingTrigger;
}

void System::getObservablesSnapshotAt(std::uint64_t time)
{
    std::vector<Snapshot> result;
    for (auto &[name, observable] : observables) {
        if (observable)
            result.push_back(observable->getSnapshot());
    }

    snapshots[time] = std::move(result); // store the snapshots by timestamp
}

std::map<std::uint64_t, std::vector<Snapshot>> System::getAllSnapshots()
{
```

```
    return snapshots;
}
```

F.4 ΔQ (maths)

The "maths" folder represents all the classes related to ΔQ, where mathematical operations are being done (hence the "maths" name).

F.4.1 ConfidenceInterval.cpp

This class represents the confidence bounds described earlier.

```
#include "ConfidenceInterval.h"
#include <iostream>
#include <math.h>
#include <stdexcept>
#include <vector>

ConfidenceInterval::ConfidenceInterval()
    : numBins(0)
{
}

ConfidenceInterval::ConfidenceInterval(int numBins)

    : numBins(numBins)
    , cdfSum(numBins, 0.0)
    , cdfSumSquares(numBins, 0.0)
    , cdfSampleCounts(numBins, 0)
    , bounds(numBins)
{
}
void ConfidenceInterval::addDeltaQ(const DeltaQ &deltaQ)
{
    const auto &cdf = deltaQ.getCdfValues();
    if (deltaQ == DeltaQ()) {
        return;
    }

    if (cdf.size() != numBins) {
        std::cerr << "CDF size mismatch in addDeltaQ, have" << deltaQ.
    getBins() << " expected" << numBins << "\n";
        return;
    }
    for (size_t i = 0; i < cdf.size(); ++i) {
        const double cdfValue = cdf[i];

        cdfSum[i] += cdfValue;
        cdfSumSquares[i] += cdfValue * cdfValue;
        cdfSampleCounts[i] += 1;
    }

    updateConfidenceInterval();
```

```
}

void ConfidenceInterval::removeDeltaQ(const DeltaQ &deltaQ)
{
    if (deltaQ == DeltaQ()) {
        return;
    }

    const auto &cdf = deltaQ.getCdfValues();
    if (cdf.size() != numBins) {
        return; // Returning a previous DeltaQ which had different bins
    }

    for (size_t i = 0; i < cdf.size(); ++i) {
        const double cdfValue = cdf[i];

        cdfSum[i] -= cdfValue;
        cdfSumSquares[i] -= cdfValue * cdfValue;

        if (cdfSampleCounts[i] == 0) {
            return; // Removing more than needed
        }

        cdfSampleCounts[i] -= 1;
    }

    updateConfidenceInterval();
}

void ConfidenceInterval::updateConfidenceInterval()
{
    for (size_t i = 0; i < bounds.size(); ++i) {
        unsigned int n = cdfSampleCounts[i];
        if (cdfSampleCounts[i] == 0) {
            bounds[i].lowerBound = 0.0;
            bounds[i].upperBound = 0.0;
            continue;
        }

        double mean = cdfSum[i] / n;
        double meanSquare = cdfSumSquares[i] / n;
        double variance = meanSquare - (mean * mean);
        double stddev = std::sqrt(std::max(variance, 0.0));
        double marginOfError = z * stddev / std::sqrt(n);
        bounds[i].lowerBound = std::max(0.0, mean - marginOfError);
        bounds[i].upperBound = std::min(1.0, mean + marginOfError);
        bounds[i].mean = mean;
    }
}

void ConfidenceInterval::reset()
{
    bounds = std::vector<Bound>();
    bounds.resize(numBins);
    cdfSum = std::vector<double>();
```

```
    cdfSum.resize(numBins);
    cdfSumSquares = std::vector<double>();
    cdfSumSquares.resize(numBins);
    cdfSampleCounts = std::vector<unsigned int>();
    cdfSampleCounts.resize(numBins);
}

void ConfidenceInterval::setNumBins(int newNumBins)
{
    numBins = newNumBins;
    reset();
}

std::vector<Bound> ConfidenceInterval::getBounds() const
{
    return bounds;
}

unsigned int ConfidenceInterval::getBins()
{
    return numBins;
}
```

F.4.2 DeltaQ.cpp

This class represents a ΔQ. It supports calculating a ΔQ given multiple samples, calculating the quartiles of a ΔQ, it supports various arithmetical transformations.

```


#include "DeltaQ.h"

#include <algorithm>
#include <cmath>
#include <functional>
#include <iomanip>
#include <iostream>

#include <chrono>
#include <iostream>
DeltaQ::DeltaQ(const double binWidth)
    : binWidth(binWidth)
    , bins(0)
    , qta()
{
}

DeltaQ::DeltaQ(const double binWidth, const std::vector<double> &
   values, const bool isPdf)
    : binWidth(binWidth)
    , bins(values.size()) // Values is binned data
    , qta()
{
    if (isPdf) {
        pdfValues = values;
```

```
        calculateCDF();
    } else {
        cdfValues = values;
        calculatePDF();
    }
}
DeltaQ::DeltaQ(double binWidth, std::vector<Sample> &samples)
    : binWidth(binWidth)
    , bins {50}
    , qta()
{
    calculateDeltaQ(samples);
}

DeltaQ::DeltaQ(double binWidth, std::vector<Sample> &samples, int bins
   )
    : binWidth(binWidth)
    , bins(bins)
    , qta()
{
    calculateDeltaQ(samples);
}

void DeltaQ::calculateDeltaQ(std::vector<Sample> &outcomeSamples)
{

    double size = binWidth * bins;
    if (outcomeSamples.empty() || binWidth <= 0) {
        bins = 0;
        return;
    }
    std::vector<double> histogram(bins, 0.0);
    totalSamples = outcomeSamples.size();
    long long successfulSamples = 0;

    std::sort(outcomeSamples.begin(), outcomeSamples.end(), [](const
   Sample &a, const Sample &b) { return a.elapsedTime < b.elapsedTime
   ; });

    for (const auto &sample : outcomeSamples) {
        if (sample.status != Status::SUCCESS) {
            continue; // Exclude failed samples from histogram but
   count them
        }

        double elapsed = sample.elapsedTime;

        if (elapsed < 0 || std::isnan(elapsed) || std::isinf(elapsed))
    {
            std::cerr << "Warning: Invalid sample value: " << elapsed
   << std::endl;
            continue;
        }
        double invBinWidth = 1.0 / binWidth;
        int bin = static_cast<int>(elapsed * invBinWidth);
        if (bin < 0) {
```

```cpp
            std::cerr << "Warning: Negative bin value: " << bin << std::endl;
            continue;
        }
        if (bin >= bins) {
            if (elapsed > size) {
                continue;
            }
            bin = bins - 1;
        }

        successfulSamples++;
        histogram[bin] += 1.0;
    }
    // Calculate PDF
    for (double &val : histogram) {
        val /= totalSamples;
    }
    pdfValues = std::move(histogram);

    calculateCDF();
    calculateQuartiles(outcomeSamples);
}

void DeltaQ::calculateQuartiles(std::vector<Sample> &outcomeSamples)
{
    if (outcomeSamples.empty()) {
        return;
    }
    const size_t n = outcomeSamples.size();

    auto getElapsedAt = [&](size_t index) -> double { return outcomeSamples[index].elapsedTime; };

    auto getPercentile = [&](double p) -> double {
        double pos = p * (n - 1);
        auto idx = static_cast<size_t>(pos);
        double frac = pos - idx;

        if (idx + 1 < n) {
            // Linear interpolation between samples[idx] and samples[idx + 1]
            return getElapsedAt(idx) * (1.0 - frac) + getElapsedAt(idx + 1) * frac;
        }
        return getElapsedAt(idx);
    };
    qta = QTA::create(getPercentile(0.25), getPercentile(0.50), getPercentile(0.75), ((cdfAt(bins - 1)) > 1) ? 1 : cdfAt(bins - 1));
}

void DeltaQ::calculateCDF()
{
    cdfValues.clear();
    cdfValues.reserve(bins);
```

```
    double cumulativeSum = 0;
    for (const double &pdfValue : pdfValues) {
        cumulativeSum += pdfValue;
        cdfValues.push_back(cumulativeSum);
    }
}

void DeltaQ::calculatePDF()
{
    pdfValues.clear();

    double previous = 0;
    for (const double &cdfValue : cdfValues) {
        pdfValues.push_back(cdfValue - previous);
        previous = cdfValue;
    }
}

QTA DeltaQ::getQTA() const
{
    return qta;
}

const std::vector<double> &DeltaQ::getPdfValues() const
{
    return pdfValues;
}

const std::vector<double> &DeltaQ::getCdfValues() const
{
    return cdfValues;
}

const unsigned int DeltaQ::getTotalSamples() const
{
    return totalSamples;
}

void DeltaQ::setBinWidth(double newWidth)
{
    binWidth = newWidth;
}

double DeltaQ::getBinWidth() const
{
    return binWidth;
}

int DeltaQ::getBins() const
{
    return bins;
}

double DeltaQ::pdfAt(int x) const
{
    if (bins != 0) {
```

```
        if (x >= bins) {
            return 0.0;
        }
        return pdfValues.at(x);
    }
    return 0;
}

double DeltaQ::cdfAt(int x) const
{
    if (bins != 0) {
        if (x >= bins) {
            return cdfValues.at(bins - 1);
        }
        return cdfValues.at(x);
    }
    return 0;
}

bool DeltaQ::operator<(const DeltaQ &other) const
{
    return this->bins < other.bins;
}

bool DeltaQ::operator>(const DeltaQ &other) const
{
    return this->bins > other.bins;
}

bool DeltaQ::operator==(const DeltaQ &deltaQ) const
{
    return pdfValues == deltaQ.getPdfValues();
}

DeltaQ operator*(const DeltaQ &deltaQ, double constant)
{
    DeltaQ result(deltaQ.binWidth);
    result.bins = deltaQ.bins;

    result.pdfValues = deltaQ.pdfValues;
    std::transform(result.pdfValues.begin(), result.pdfValues.end(),
   result.pdfValues.begin(), [constant](double value) { return value
   * constant; });

    result.cdfValues = deltaQ.cdfValues;
    std::transform(result.cdfValues.begin(), result.cdfValues.end(),
   result.cdfValues.begin(), [constant](double value) { return value
   * constant; });

    return result;
}

template <typename BinaryOperation>
DeltaQ applyBinaryOperation(const DeltaQ &lhs, const DeltaQ &rhs,
    BinaryOperation op)
{
```

```
    const DeltaQ &highestDeltaQ = (lhs > rhs) ? lhs : rhs;
    const DeltaQ &otherDeltaQ = (lhs > rhs) ? rhs : lhs;

    std::vector<double> resultingCdf;
    resultingCdf.reserve(highestDeltaQ.getBins());

    for (size_t i = 0; i < highestDeltaQ.getBins(); i++) {
        double result = op(highestDeltaQ.cdfAt(i), otherDeltaQ.cdfAt(i
));
        resultingCdf.push_back(result);
    }

    return {highestDeltaQ.getBinWidth(), resultingCdf, false};
}

DeltaQ operator*(double constant, const DeltaQ &deltaQ)
{
    return deltaQ * constant;
}

DeltaQ operator+(const DeltaQ &lhs, const DeltaQ &rhs)
{
    return applyBinaryOperation(lhs, rhs, std::plus<>());
}

DeltaQ operator-(const DeltaQ &lhs, const DeltaQ &rhs)
{
    return applyBinaryOperation(lhs, rhs, std::minus<>());
}

DeltaQ operator*(const DeltaQ &lhs, const DeltaQ &rhs)
{
    return applyBinaryOperation(lhs, rhs, std::multiplies<>());
}

std::string DeltaQ::toString() const
{
    std::ostringstream oss;
    oss << "<";

    // Iterate through CDF values to construct the string
    for (size_t i = 0; i < pdfValues.size(); ++i) {
        const double bin = (i + 1) * binWidth;
        oss << "(" << std::fixed << std::setprecision(7) << bin << ",
" << std::setprecision(7) << pdfValues[i] << ")";
        if (i < pdfValues.size() - 1) {
            oss << ", ";
        }
    }

    oss << ">";
    return oss.str();
}
```

F.4.3 DeltaQOperations.cpp

This file contains the definition of all the operations that can be done on a ΔQ or on ΔQs, specified in the implementation chapter. Convolution (naive, FFT), FTF, ATF, PC operators, rebinning.

```
#include "DeltaQOperations.h"
#include "DeltaQ.h"
#include <fftw3.h>
#include <iostream>
#include <math.h>
#include <mutex>
#include <vector>
DeltaQ rebin(const DeltaQ &source, double targetBinWidth)
{
    double originalBinWidth = source.getBinWidth();

    if (std::abs(originalBinWidth - targetBinWidth) < 1e-9) {
        return source; // Already same bin width
    }

    if (targetBinWidth < originalBinWidth) {
        throw std::invalid_argument("Target bin width must be greater
   than or equal to original.");
    }

    int factor = static_cast<int>(std::round(targetBinWidth /
   originalBinWidth));
    const auto &originalPdf = source.getPdfValues();

    int newNumBins = static_cast<int>(std::ceil(originalPdf.size() /
   static_cast<double>(factor)));
    std::vector<double> newPdf(newNumBins, 0.0);

    for (size_t i = 0; i < originalPdf.size(); ++i) {
        newPdf[i / factor] += originalPdf[i];
    }

    return DeltaQ(targetBinWidth, newPdf, true);
}

DeltaQ convolve(const DeltaQ &lhs, const DeltaQ &rhs)
{
    double commonBinWidth = std::max(lhs.getBinWidth(), rhs.
   getBinWidth());

    if (lhs == DeltaQ()) {
        return rhs;
    }
    if (rhs == DeltaQ()) {
        return lhs;
    }

    DeltaQ lhsRebinned = rebin(lhs, commonBinWidth);
    DeltaQ rhsRebinned = rebin(rhs, commonBinWidth);

```

```
    const auto &lhsPdf = lhsRebinned.getPdfValues();
    const auto &rhsPdf = rhsRebinned.getPdfValues();

    int resultSize = lhsPdf.size() + rhsPdf.size() - 1;
    std::vector<double> resultPdf(resultSize, 0.0);

    for (size_t i = 0; i < lhsPdf.size(); ++i) {
        for (size_t j = 0; j < rhsPdf.size(); ++j) {
            resultPdf[i + j] += lhsPdf[i] * rhsPdf[j];
        }
    }

    return DeltaQ(commonBinWidth, resultPdf, true);
}

// Inspired by https://github.com/jeremyfix/FFTConvolution/blob/master
    /Convolution/src/convolution_fftw.h
DeltaQ convolveFFT(const DeltaQ &lhs, const DeltaQ &rhs)
{
    if (lhs == DeltaQ()) {
        return rhs;
    }
    if (rhs == DeltaQ()) {
        return lhs;
    }

    // Find a common bin width and rebin accordingly
    double commonBinWidth = std::max(lhs.getBinWidth(), rhs.
    getBinWidth());

    DeltaQ lhsRebinned = rebin(lhs, commonBinWidth);
    DeltaQ rhsRebinned = rebin(rhs, commonBinWidth);

    const auto &lhsPdf = lhsRebinned.getPdfValues();
    const auto &rhsPdf = rhsRebinned.getPdfValues();

    // Find the power of 2 nearest to the convolution size
    size_t lhsSize = lhsPdf.size();
    size_t rhsSize = rhsPdf.size();
    size_t convSize = lhsSize + rhsSize - 1;
    size_t fftSize = 1;
    while (fftSize < convSize)
        fftSize <<= 1;

    // Pad pdf with zeroes until end of PDF
    std::vector<double> lhsPadded(fftSize, 0.0);
    std::vector<double> rhsPadded(fftSize, 0.0);
    std::copy(lhsPdf.begin(), lhsPdf.end(), lhsPadded.begin());
    std::copy(rhsPdf.begin(), rhsPdf.end(), rhsPadded.begin());
    static std::mutex fftw_mutex;
    {
        std::lock_guard<std::mutex> lock(fftw_mutex);
        fftw_complex *lhsFreq = (fftw_complex *)fftw_malloc(sizeof(
    fftw_complex) * (fftSize / 2 + 1));
        fftw_complex *rhsFreq = (fftw_complex *)fftw_malloc(sizeof(
    fftw_complex) * (fftSize / 2 + 1));
```

```
        double *lhsTime = lhsPadded.data();
        double *rhsTime = rhsPadded.data();

        // Transform real input to complex output
        fftw_plan planLhs = fftw_plan_dft_r2c_1d(fftSize, lhsTime, lhsFreq, FFTW_ESTIMATE);
        fftw_plan planRhs = fftw_plan_dft_r2c_1d(fftSize, rhsTime, rhsFreq, FFTW_ESTIMATE);
        fftw_execute(planLhs);
        fftw_execute(planRhs);

        // Do complex multiplication, r: ac - bd i : ad + bc
        fftw_complex *resultFreq = (fftw_complex *)fftw_malloc(sizeof(fftw_complex) * (fftSize / 2 + 1));
        for (size_t i = 0; i < fftSize / 2 + 1; ++i) {
            double a = lhsFreq[i][0], b = lhsFreq[i][1];
            double c = rhsFreq[i][0], d = rhsFreq[i][1];
            resultFreq[i][0] = a * c - b * d;
            resultFreq[i][1] = a * d + b * c;
        }

        // Invert from complex plane to real plane
        std::vector<double> resultTime(fftSize);
        fftw_plan planInv = fftw_plan_dft_c2r_1d(fftSize, resultFreq, resultTime.data(), FFTW_ESTIMATE);
        fftw_execute(planInv);

        for (auto &val : resultTime)
            val /= fftSize;

        resultTime.resize(convSize);

        fftw_destroy_plan(planLhs);
        fftw_destroy_plan(planRhs);
        fftw_destroy_plan(planInv);
        fftw_free(lhsFreq);
        fftw_free(rhsFreq);
        fftw_free(resultFreq);
        return {lhsRebinned.getBinWidth(), resultTime, true};
    }
}

DeltaQ convolveN(const std::vector<DeltaQ> &deltaQs)
{
    if (deltaQs.empty())
        return DeltaQ();

    DeltaQ result = deltaQs[0];
    for (size_t i = 1; i < deltaQs.size(); ++i) {
        result = convolveFFT(result, deltaQs[i]);
    }
    return result;
}

DeltaQ probabilisticChoice(const std::vector<double> &probabilities, const std::vector<DeltaQ> &deltaQs)
```

```
{
    std::vector<DeltaQ> nonEmpty;
    std::vector<double> effectiveProbs;
    double commonBinWidth = 0.0;

    for (size_t i = 0; i < deltaQs.size(); ++i) {
        if (deltaQs[i] == DeltaQ()) {
            continue;
        }
        nonEmpty.push_back(deltaQs[i]);
        effectiveProbs.push_back(probabilities[i]);
        commonBinWidth = std::max(commonBinWidth, deltaQs[i].
   getBinWidth());
    }

    if (nonEmpty.empty()) {
        return DeltaQ();
    }

    for (auto &dq : nonEmpty) {
        dq = rebin(dq, commonBinWidth);
    }

    std::vector<DeltaQ> scaledDeltaQs;
    for (size_t i = 0; i < nonEmpty.size(); ++i) {
        scaledDeltaQs.push_back(nonEmpty[i] * effectiveProbs[i]);
    }

    DeltaQ result = scaledDeltaQs[0];
    for (size_t i = 1; i < scaledDeltaQs.size(); ++i) {
        result = result + scaledDeltaQs[i];
    }

    return result;
}

DeltaQ firstToFinish(const std::vector<DeltaQ> &deltaQs)
{
    std::vector<DeltaQ> nonEmpty;
    double commonBinWidth = 0.0;

    for (const auto &dq : deltaQs) {
        if (dq == DeltaQ()) {
            continue;
        }
        nonEmpty.push_back(dq);
        commonBinWidth = std::max(commonBinWidth, dq.getBinWidth());
    }

    if (nonEmpty.empty()) {
        return DeltaQ();
    }

    for (auto &dq : nonEmpty) {
        dq = rebin(dq, commonBinWidth);
    }
```

```cpp

    const int largestSize = chooseLongestDeltaQSize(nonEmpty);
    std::vector<double> resultingCdf;

    for (int i = 0; i < largestSize; ++i) {
        double sumAtI = 0;
        double productAtI = 1;
        for (const auto &dq : nonEmpty) {
            const double cdfAtI = dq.cdfAt(i);
            sumAtI += cdfAtI;
            productAtI *= cdfAtI;
        }
        resultingCdf.push_back(sumAtI - productAtI);
    }
    return {commonBinWidth, resultingCdf, false};
}

DeltaQ allToFinish(const std::vector<DeltaQ> &deltaQs)
{
    std::vector<DeltaQ> nonEmpty;
    double commonBinWidth = 0.0;

    for (const auto &dq : deltaQs) {
        if (dq == DeltaQ()) {
            continue;
        }
        nonEmpty.push_back(dq);
        commonBinWidth = std::max(commonBinWidth, dq.getBinWidth());
    }

    if (nonEmpty.empty()) {
        return DeltaQ();
    }

    for (auto &dq : nonEmpty) {
        dq = rebin(dq, commonBinWidth);
    }

    DeltaQ result = nonEmpty[0];
    for (size_t i = 1; i < nonEmpty.size(); ++i) {
        result = result * nonEmpty[i];
    }

    return result;
}

int chooseLongestDeltaQSize(const std::vector<DeltaQ> &deltaQs)
{
    int highestSize = 0;
    for (const DeltaQ &deltaQ : deltaQs) {
        if (deltaQ.getBins() > highestSize) {
            highestSize = deltaQ.getBins();
        }
    }
    return highestSize;
}
```

F.4.4 Snapshot.cpp

This class represents a single snapshot of a probe. It contains the QTA, observable ΔQ and calculated ΔQ at time t.

```cpp
#include "Snapshot.h"

void Snapshot::addObservedDeltaQ(std::uint64_t time, const DeltaQ &
    deltaQ, const std::vector<Bound> &bounds)
{
    DeltaQRepr repr = {time, deltaQ, bounds};
    observedDeltaQs[time] = repr;
}

void Snapshot::removeOldestObservedDeltaQ()
{
    if (getObservedSize() == 0)
        return;
    observedDeltaQs.erase(observedDeltaQs.begin());
}

void Snapshot::addCalculatedDeltaQ(std::uint64_t time, const DeltaQ &
    deltaQ, const std::vector<Bound> &bounds)
{
    DeltaQRepr repr = {time, deltaQ, bounds};
    calculatedDeltaQs[time] = repr;
}

DeltaQ Snapshot::getOldestCalculatedDeltaQ() const
{
    return calculatedDeltaQs.begin()->second.deltaQ;
}

DeltaQ Snapshot::getOldestObservedDeltaQ() const
{
    return observedDeltaQs.begin()->second.deltaQ;
}

void Snapshot::removeOldestCalculatedDeltaQ()
{
    if (getCalculatedSize() == 0)
        return;

    calculatedDeltaQs.erase(calculatedDeltaQs.begin());
}

void Snapshot::resizeTo(size_t newSize)
{
    int observedSize = getObservedSize();
    int calculatedSize = getCalculatedSize();

    // Don't shrink if not needed
    if (observedSize <= newSize && calculatedSize <= newSize)
```

```
        return;

    if (observedSize > newSize) {
        int toObserved = observedSize - newSize;
        auto endIt = std::next(observedDeltaQs.begin(), toObserved);
        observedDeltaQs.erase(observedDeltaQs.begin(), endIt);
    }

    if (calculatedSize > newSize) {
        int toCalculated = calculatedSize - newSize;
        auto endItC = std::next(calculatedDeltaQs.begin(),
    toCalculated);
        calculatedDeltaQs.erase(calculatedDeltaQs.begin(), endItC);
    }

    if (QTAs.size() > newSize) {
        int toSize = QTAs.size() - newSize;
        auto endIt = std::next(QTAs.begin(), toSize);
        QTAs.erase(QTAs.begin(), endIt);
    }
}
void Snapshot::addQTA(uint64_t time, const QTA &qta)
{
    QTAs[time] = qta;
}

std::optional<DeltaQRepr> Snapshot::getObservedDeltaQAtTime(std::
    uint64_t time)
{
    auto it = observedDeltaQs.find(time);
    if (it != observedDeltaQs.end()) {
        return it->second;
    }
    return std::nullopt;
}

std::optional<DeltaQRepr> Snapshot::getCalculatedDeltaQAtTime(std::
    uint64_t time)
{
    auto it = calculatedDeltaQs.find(time);
    if (it != calculatedDeltaQs.end()) {
        return it->second;
    }
    return std::nullopt;
}

std::size_t Snapshot::getObservedSize() const
{
    return observedDeltaQs.size();
}

std::size_t Snapshot::getCalculatedSize() const
{
    return calculatedDeltaQs.size();
}

```

```
std::vector<DeltaQRepr> Snapshot::getObservedDeltaQs() const &
{
    auto obs = std::vector<DeltaQRepr>();
    for (const auto &s : observedDeltaQs)
        obs.emplace_back(s.second);
    return obs;
}

std::vector<DeltaQRepr> Snapshot::getCalculatedDeltaQs() const &
{
    auto calc = std::vector<DeltaQRepr>();
    for (const auto &s : calculatedDeltaQs)
        calc.emplace_back(s.second);
    return calc;
}

std::vector<QTA> Snapshot::getQTAs() const &
{
    auto QTAsVec = std::vector<QTA>();
    for (const auto &q : QTAs) {
        QTAsVec.emplace_back(q.second);
    }
    return QTAsVec;
}

void Snapshot::setName(const std::string &name)
{
    observableName = name;
}

std::string Snapshot::getName() &
{
    return observableName;
}
```

F.4.5 TriggerManager.cpp

This class is the manager of triggers for a probe. It can add/remove/evaluate triggers.

```
#include "TriggerManager.h"

#include <algorithm>

TriggerManager::Trigger::Trigger(TriggerType t, TriggerDefs::Condition
    c, TriggerDefs::Action a, bool e)
    : type(t)
    , condition(std::move(c))
    , action(std::move(a))
    , enabled(e)
{
}

void TriggerManager::addTrigger(TriggerType type, TriggerDefs::
   Condition condition, TriggerDefs::Action action, bool enabled)
{
```

```
    triggers_.emplace_back(type, std::move(condition), std::move(
    action), enabled);
}

std::vector<TriggerManager::Trigger> TriggerManager::getTriggersByType
    (TriggerType type)
{
    std::vector<Trigger> result;
    for (auto &trigger : triggers_) {
        if (trigger.type == type) {
            result.push_back(trigger);
        }
    }
    return result;
}

void TriggerManager::evaluate(const DeltaQ &dq, const QTA &qta, std::
    uint64_t time) const
{
    for (const auto &trigger : triggers_) {
        if (trigger.enabled && trigger.condition(dq, qta)) {
            trigger.action(dq, qta, time);
        }
    }
}

void TriggerManager::removeTriggersByType(TriggerType type)
{
    triggers_.erase(std::remove_if(triggers_.begin(), triggers_.end(),
     [type](const auto &trigger) { return trigger.type == type; }),
    triggers_.end());
}

void TriggerManager::clearAllTriggers()
{
    triggers_.clear();
}

void TriggerManager::setTriggersEnabled(TriggerType type, bool enabled
    )
{
    for (auto &trigger : triggers_) {
        if (trigger.type == type) {
            trigger.enabled = enabled;
        }
    }
}
std::vector<TriggerManager::Trigger> TriggerManager::getAllTriggers()
    const
{
    std::vector<Trigger> result;
    for (const auto &trigger : triggers_) {
        result.push_back(trigger);
    }
    return result;
}
```

F.4.6 Triggers.cpp

This class contains the conditions of the triggers selected by the user. The trigger manager evaluates the conditions at runtime. The Actions namespace is WIP.

```
#include "Triggers.h"

namespace TriggerDefs
{
namespace Conditions
{
    Condition SampleLimit(int maxSamples)
    {
        return [maxSamples](const DeltaQ &dq, const QTA &) { return dq
    .getTotalSamples() > maxSamples; };
    }

    Condition QTABounds()
    {
        return [](const DeltaQ &dq, const QTA &qta) {
            const QTA &dqQta = dq.getQTA();
            return dqQta.perc_25 > qta.perc_25 || dqQta.perc_50 > qta.
    perc_50 || dqQta.perc_75 > qta.perc_75 || dqQta.cdfMax < qta.
    cdfMax;
        };
    }

    Condition FailureRate(double threshold)
    {
        return [threshold](const DeltaQ &dq, const QTA &) { return dq.
    getQTA().cdfMax < threshold; };
    }
}

namespace Actions
{
    Action LogToConsole(const std::string &message)
    {
        return [message](const DeltaQ &, const QTA &, std::uint64_t) {
     std::cout << "TRIGGER: " << message << "\n"; };
    }

    Action notify()
    {
        return [](const DeltaQ &dq, const QTA &qta, std::uint64_t) {
    };
    }

    Action SaveSnapshot(const std::string &filename)
    {
        return [](const DeltaQ &dq, const QTA &, std::uint64_t) { };
    }
}
```

```
}
```

F.5 parser

F.5.1 SystemBuilder.cpp

This class builds a new outcome diagram (system class) given an AST built when parsing.

```
#include "SystemBuilder.h"
#include <memory>
#include <stdexcept>

#include <locale>
System SystemBuilderVisitor::getSystem() const
{
    return system;
}

void SystemBuilderVisitor::checkForCycles() const
{
    std::set<std::string> visited;
    std::set<std::string> recursionStack;

    for (const auto &[node, _] : dependencies) {
        if (hasCycle(node, visited, recursionStack)) {
            throw std::invalid_argument("Cycle detected in system
    definition involving: " + node);
        }
    }
}

bool SystemBuilderVisitor::hasCycle(const std::string &node, std::set<
    std::string> &visited, std::set<std::string> &recursionStack)
    const
{
    if (recursionStack.find(node) != recursionStack.end()) {
        return true;
    }
    if (visited.find(node) != visited.end()) {
        return false;
    }

    visited.insert(node);
    recursionStack.insert(node);

    if (dependencies.find(node) != dependencies.end()) {
        for (const auto &neighbor : dependencies.at(node)) {
            if (hasCycle(neighbor, visited, recursionStack)) {
                return true;
            }
        }
    }

```

```
    recursionStack.erase(node);
    return false;
}

std::any SystemBuilderVisitor::visitStart(parser::DQGrammarParser::StartContext *context)
{
    for (const auto definition : context->definition()) {
        visitDefinition(definition);
    }

    if (context->system()) {
        visitSystem(context->system());
    }

    system.setOutcomes(outcomes);
    system.setProbes(probes);
    system.setOperators(operators);
    /*
    for (const auto &[name, link] : definitionLinks) {
        std::cout << name << " [ ";
        for (auto &name2 : link) {
            std::cout << name2 << " ";
        }
        std::cout << "]\n";
    }

    for (const auto &[name, op] : operatorLinks) {
        std::cout << name << " [ ";
        for (auto link : op) {
            std::cout << " [";
            for (auto lill : link) {
                std::cout << lill << " ";
            }
            std::cout << "]";
        }
        std::cout << "]\n";
    }
    */
    checkForCycles();
    return nullptr;
}

std::any SystemBuilderVisitor::visitDefinition(parser::DQGrammarParser::DefinitionContext *context)
{
    std::string probeName = context->IDENTIFIER()->getText();
    if (std::find(definedProbes.begin(), definedProbes.end(), probeName) != definedProbes.end()) {
        throw std::invalid_argument("Probe has already been defined");
    }
    if (allNames.find(probeName) != allNames.end()) {
        throw std::invalid_argument("Duplicate name detected: " + probeName);
    }
    allNames.insert(probeName);
```

```
    currentlyBuildingProbe = probeName;

    const auto chainComponents = std::any_cast<std::vector<std::shared_ptr<Observable>>>(visitComponent_chain(context->component_chain()));
    std::vector<std::shared_ptr<Observable>> probeCausalLinks;
    std::vector<std::string> links;
    for (auto &comp : chainComponents) {
        probeCausalLinks.push_back(comp);
        links.push_back(comp->getName());
    }

    const auto probe = std::make_shared<Probe>(probeName, probeCausalLinks);

    probes[probeName] = probe;
    definedProbes.push_back(probeName);
    definitionLinks[probeName] = links;
    currentlyBuildingProbe = "";
    return nullptr;
}

std::any SystemBuilderVisitor::visitSystem(parser::DQGrammarParser::SystemContext *context)
{
    const auto chainComponents = std::any_cast<std::vector<std::shared_ptr<Observable>>>(visitComponent_chain(context->component_chain()));

    std::vector<std::string> links;
    for (auto &comp : chainComponents) {
        links.push_back(comp->getName());
    }

    systemLinks = links;

    return nullptr;
}

std::any SystemBuilderVisitor::visitComponent(parser::DQGrammarParser::ComponentContext *context)
{
    if (context->behaviorComponent()) {
        return visitBehaviorComponent(context->behaviorComponent());
    } else if (context->probeComponent()) {
        return visitProbeComponent(context->probeComponent());
    } else if (context->outcome()) {
        return visitOutcome(context->outcome());
    }
    return nullptr;
}

std::any SystemBuilderVisitor::visitBehaviorComponent(parser::DQGrammarParser::BehaviorComponentContext *context)
{
    const std::string typeStr = context->BEHAVIOR_TYPE()->getText();
```

```
    std::string name = context->IDENTIFIER()->getText();

    if (operators.find(name) != operators.end()) {
        return std::dynamic_pointer_cast<Observable>(operators[name]);
    }
    if (allNames.find(name) != allNames.end()) {
        throw std::invalid_argument("Duplicate name detected: " + name);
    }

    allNames.insert(name);
    OperatorType type;
    if (typeStr == "a")
        type = OperatorType::ATF;
    else if (typeStr == "f")
        type = OperatorType::FTF;
    else if (typeStr == "p")
        type = OperatorType::PRB;
    else
        throw std::invalid_argument("Unknown operator type: " + typeStr);

    const auto op = std::make_shared<Operator>(name, type);
    if (type == OperatorType::PRB && context->probability_list()) {
        const auto probabilities = std::any_cast<std::vector<double>>(visitProbability_list(context->probability_list()));
        op->setProbabilities(probabilities);
    } else if (type == OperatorType::PRB && !context->probability_list()) {
        throw std::invalid_argument("A probabilistic operator must have probabilities");
    } else if (type != OperatorType::PRB && context->probability_list()) {
        throw std::invalid_argument("A non probabilistic operator cannot have probabilities");
    }

    std::vector<std::vector<std::shared_ptr<Observable>>> operatorPtrLinks;

    if (context->component_list()) {
        auto childrenChains = std::any_cast<std::vector<std::vector<std::shared_ptr<Observable>>>>(visitComponent_list(context->component_list()));

        for (auto &chain : childrenChains) {
            if (!chain.empty()) {
                operatorPtrLinks.push_back(chain);
            }
        }
    }

    op->setCausalLinks(operatorPtrLinks);

    if (context->component_list()) {
        auto childrenChains = std::any_cast<std::vector<std::vector<
```

```cpp
std::shared_ptr<Observable>>>>(visitComponent_list(context->component_list()));

        std::vector<std::vector<std::string>> childrenLinks;

        for (auto &chain : childrenChains) {
            if (!chain.empty()) {
                std::vector<std::string> chainNames;
                for (auto &comp : chain) {
                    chainNames.push_back(comp->getName());
                }
                childrenLinks.push_back(chainNames);
            }
        }

        operatorLinks[name] = childrenLinks;
    }

    operators[name] = op;
    return std::dynamic_pointer_cast<Observable>(op);
}
std::any SystemBuilderVisitor::visitProbeComponent(parser::DQGrammarParser::ProbeComponentContext *context)
{
    std::string name = context->IDENTIFIER()->getText();

    if (!currentlyBuildingProbe.empty()) {
        dependencies[currentlyBuildingProbe].push_back(name);
    }

    if (probes.find(name) != probes.end()) {
        return std::dynamic_pointer_cast<Observable>(probes[name]);
    }

    // Create a stub probe (may be fleshed out later)
    auto probe = std::make_shared<Probe>(name);
    probes[name] = probe;
    return std::dynamic_pointer_cast<Observable>(probe);
}

std::any SystemBuilderVisitor::visitProbability_list(parser::DQGrammarParser::Probability_listContext *context)
{
    std::locale::global(std::locale("C"));

    std::vector<double> probabilities;
    for (auto num : context->NUMBER()) {
        probabilities.push_back(std::stod(num->getText()));
    }
    return probabilities;
}

std::any SystemBuilderVisitor::visitComponent_list(parser::DQGrammarParser::Component_listContext *context)
{
    std::vector<std::vector<std::shared_ptr<Observable>>>
```

```
componentsChains;

    for (auto chainCtx : context->component_chain()) {
        auto chain = std::any_cast<std::vector<std::shared_ptr<Observable>>>(visitComponent_chain(chainCtx));
        componentsChains.push_back(chain);
    }

    return componentsChains;
}

std::any SystemBuilderVisitor::visitComponent_chain(parser::DQGrammarParser::Component_chainContext *context)
{
    std::vector<std::shared_ptr<Observable>> components;

    for (auto compCtx : context->component()) {
        auto component = std::any_cast<std::shared_ptr<Observable>>(visitComponent(compCtx));
        components.push_back(component);
    }
    return components;
}

std::any SystemBuilderVisitor::visitOutcome(parser::DQGrammarParser::OutcomeContext *context)
{
    std::string name = context->IDENTIFIER()->getText();

    if (outcomes.find(name) != outcomes.end()) {
        return std::dynamic_pointer_cast<Observable>(outcomes[name]);
    }
    if (allNames.find(name) != allNames.end()) {
        throw std::invalid_argument("Duplicate name detected: " + name);
    }
    allNames.insert(name);

    auto outcome = std::make_shared<Outcome>(name);
    outcomes[name] = outcome;
    return std::dynamic_pointer_cast<Observable>(outcome);
}
```

F.5.2 SystemParserInterface.cpp

This files contains functions to be called by the dashboard to avoid communicating directly to ANTLR. It throws errors which are caught by the caller if the parsing was unsuccessful.

```
#include "SystemParserInterface.h"
#include "SystemErrorListener.h"
#include <exception>
#include <fstream>
#include <iostream>
```

```cpp
#include <sstream>
#include <stdexcept>

/**
 * @brief Parses a system definition from a file.
 * @param filename Path to the file containing system definition.
 * @return Optional containing the parsed System if successful, nullopt on error.
 * @throws std::invalid_argument if parsing fails.
 */
std::optional<System> SystemParserInterface::parseFile(const std::string &filename)
{
    std::ifstream file(filename);
    if (!file) {
        std::cerr << "Error: Could not open file: " << filename << std::endl;
        return std::nullopt;
    }

    // Read entire file content
    std::stringstream buffer;
    buffer << file.rdbuf();
    std::string content = buffer.str();

    antlr4::ANTLRInputStream input(content);
    try {
        return parseInternal(input);
    } catch (std::exception &e) {
        throw std::invalid_argument(e.what());
    }
}

/**
 * @brief Parses a system definition from a string.
 * @param inputStr String containing system definition.
 * @return Optional containing the parsed System if successful, nullopt on error.
 * @throws std::invalid_argument if parsing fails.
 */
std::optional<System> SystemParserInterface::parseString(const std::string &inputStr)
{
    antlr4::ANTLRInputStream input(inputStr);
    try {
        return parseInternal(input);
    } catch (std::exception &e) {
        throw std::invalid_argument(e.what());
    }
}

/**
 * @brief Internal parsing implementation using ANTLR.
 * @param input ANTLR input stream containing system definition.
 * @return Optional containing the parsed System if successful, nullopt on error.
```

```cpp
 * @throws std::invalid_argument if parsing fails.
 */
std::optional<System> SystemParserInterface::parseInternal(antlr4::
    ANTLRInputStream &input)
{
    // Initialize lexer and parser
    parser::DQGrammarLexer lexer(&input);
    antlr4::CommonTokenStream tokens(&lexer);
    parser::DQGrammarParser parser(&tokens);

    // Configure error handling
    SystemErrorListener errorListener;
    lexer.removeErrorListeners();
    parser.removeErrorListeners();
    lexer.addErrorListener(&errorListener);
    parser.addErrorListener(&errorListener);

    try {
        // Parse and build system
        auto tree = parser.start();
        SystemBuilderVisitor visitor;
        visitor.visitStart(tree);
        return visitor.getSystem();
    } catch (std::exception &e) {
        throw std::invalid_argument(e.what());
    }
}
```

F.6 server

F.6.1 Server.cpp

This class represents the server which receives and sends messages from Erlang.

```cpp
#include "Server.h"
#include "../Application.h"
#include <arpa/inet.h>
#include <cstdint>
#include <cstring>
#include <fcntl.h>
#include <iostream>
#include <regex>
#include <signal.h>
#include <sys/socket.h>
#include <unistd.h>

#define TIMEOUT "to"
#define EXEC_OK "ok"
#define FAIL "fa"

// Inspired from https://beej.us/guide/bgnet/html//index.html#client-
    server-background

/**
```

```
 * @brief Constructs the Server and registers system observer.
 * @param port The TCP port to listen on.
 */
Server::Server(int port)
    : port(port)
    , server_fd(0)
    , new_socket(0)
    , server_started(false)
{
    Application::getInstance().addObserver([this]() { this->
   updateSystem(); });

    // Start worker thread (this can run independently)
    workerThread = std::thread([this]() {
        while (!shutdownWorker) {
            std::unique_lock lock(queueMutex);
            queueCond.wait(lock, [this] { return !sampleQueue.empty()
   || shutdownWorker; });

            while (!sampleQueue.empty()) {
                auto [name, sample] = sampleQueue.front();
                sampleQueue.pop();
                lock.unlock();

                if (system) {
                    system->addSample(name, sample);
                }

                lock.lock();
            }
        }
    });
}

/**
 * @brief Destructor cleans up sockets and joins threads.
 */
Server::~Server()
{
    std::lock_guard<std::mutex> lock(erlangMutex);
    if (erlang_socket > 0) {
        close(erlang_socket);
        erlang_socket = -1;
    }
    close(new_socket);
    close(server_fd);
    if (serverThread.joinable())
        serverThread.join();
}
/**
 * @brief Updates the system reference from Application.
 */
void Server::updateSystem()
{
    system = Application::getInstance().getSystem();
}
```

```cpp

/**
 * @brief Main server loop handling client connections.
 */
void Server::run()
{
    server_fd = socket(AF_INET, SOCK_STREAM, 0);
    if (server_fd == 0) {
        perror("Socket failed");
        return;
    }

    // Configure socket options
    int opt = 1;
    setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));
    setsockopt(server_fd, SOL_SOCKET, SO_REUSEPORT, &opt, sizeof(opt));

    // Set large buffer sizes
    int bufSize = 1 << 20; // 1MB
    setsockopt(server_fd, SOL_SOCKET, SO_RCVBUF, &bufSize, sizeof(bufSize));
    setsockopt(server_fd, SOL_SOCKET, SO_SNDBUF, &bufSize, sizeof(bufSize));

    // Bind socket to specified IP
    address.sin_family = AF_INET;
    address.sin_port = htons(port);

    // Parse IP address
    if (server_ip == "0.0.0.0" || server_ip.empty()) {
        address.sin_addr.s_addr = INADDR_ANY;
    } else {
        if (inet_pton(AF_INET, server_ip.c_str(), &address.sin_addr) <= 0) {
            std::cerr << "Invalid IP address: " << server_ip << std::endl;
            close(server_fd);
            return;
        }
    }

    if (bind(server_fd, (struct sockaddr *)&address, sizeof(address)) < 0) {
        perror("Bind failed");
        close(server_fd);
        return;
    }

    // Set non-blocking mode
    fcntl(server_fd, F_SETFL, O_NONBLOCK);

    if (listen(server_fd, SOMAXCONN) < 0) {
        perror("Listen failed");
        close(server_fd);
```

```
        return;
    }

    std::cout << "Server running on " << server_ip << ":" << port << std::endl;
    running = true;
    server_started = true;

    while (running) {
        int addrlen = sizeof(address);
        client_socket = accept(server_fd, (struct sockaddr *)&address, (socklen_t *)&addrlen);

        if (client_socket < 0) {
            if (errno == EWOULDBLOCK || errno == EAGAIN) {
                std::this_thread::sleep_for(std::chrono::milliseconds(100));
                cleanupThreads();
                continue;
            }
            if (!running)
                break; // Server was stopped
            perror("Accept failed");
            continue;
        }

        std::lock_guard<std::mutex> lock(clientsMutex);
        clientThreads.emplace_back(&Server::handleClient, this, client_socket);
    }

    // Cleanup
    close(server_fd);
    server_fd = 0;
    cleanupThreads();
    server_started = false;
}

/**
 * @brief Starts the server on specified IP and port.
 * @param ip The IP address to bind to (default: "0.0.0.0" for all interfaces)
 * @param port The port to listen on
 * @return true if server started successfully
 */
bool Server::startServer(const std::string &ip, int port)
{
    if (server_started) {
        std::cerr << "Server already running. Stop it first." << std::endl;
        return false;
    }

    this->server_ip = ip;
    this->port = port;

```

```
    serverThread = std::thread(&Server::run, this);

    // Wait a bit to see if server started successfully
    std::this_thread::sleep_for(std::chrono::milliseconds(100));
    return server_started;
}

/**
 * @brief Stops the server and closes all sockets.
 */
void Server::stopServer()
{
    if (!server_started) {
        std::cout << "Server not running." << std::endl;
        return;
    }

    running = false;
    server_started = false;

    // Close server socket to break accept loop
    if (server_fd > 0) {
        close(server_fd);
        server_fd = 0;
    }

    // Join server thread
    if (serverThread.joinable()) {
        serverThread.join();
    }

    // Cleanup client threads
    cleanupThreads();

    std::cout << "Server stopped." << std::endl;
}

/**
 * @brief Sets the Erlang endpoint for connections.
 * @param ip Erlang server IP address
 * @param port Erlang server port
 * @return true if endpoint set successfully
 */
bool Server::setErlangEndpoint(const std::string &ip, int port)
{
    std::lock_guard<std::mutex> lock(erlangMutex);

    // Close existing connection if any
    if (erlang_socket > 0) {
        close(erlang_socket);
        erlang_socket = -1;
    }

    erlang_ip = ip;
    erlang_port = port;

```

```
    std::cout << "Erlang endpoint set to " << ip << ":" << port << std::endl;
    return true;
}

/**
 * @brief Connects to the Erlang process.
 * @return true if connection succeeded.
 */
bool Server::connectToErlang()
{
    std::lock_guard<std::mutex> lock(erlangMutex);

    if (erlang_socket > 0)
        return true; // Already connected

    erlang_socket = socket(AF_INET, SOCK_STREAM, 0);
    if (erlang_socket < 0) {
        perror("Erlang socket creation failed");
        return false;
    }

    sockaddr_in erlang_addr {};
    erlang_addr.sin_family = AF_INET;
    erlang_addr.sin_port = htons(erlang_port);

    if (inet_pton(AF_INET, erlang_ip.c_str(), &erlang_addr.sin_addr) <= 0) {
        std::cerr << "Invalid Erlang IP: " << erlang_ip << std::endl;
        close(erlang_socket);
        erlang_socket = -1;
        return false;
    }

    if (connect(erlang_socket, (struct sockaddr *)&erlang_addr, sizeof(erlang_addr)) < 0) {
        perror("Failed to connect to Erlang");
        close(erlang_socket);
        erlang_socket = -1;
        return false;
    }

    std::cout << "Connected to Erlang on " << erlang_ip << ":" << erlang_port << std::endl;
    return true;
}

/**
 * @brief Sends a command to the Erlang process.
 * @param command The command string to send.
 */
void Server::sendToErlang(const std::string &command)
{
    signal(SIGPIPE, SIG_IGN); // Ignore SIGPIPE to prevent crashes on disconnect

```

```
    if (!connectToErlang()) {
        std::cerr << "Unable to send to Erlang: not connected.\n";
        return;
    }

    std::lock_guard<std::mutex> lock(erlangMutex);

    std::string msgWithNewline = command + "\n";
    ssize_t sent = send(erlang_socket, msgWithNewline.c_str(),
msgWithNewline.size(), 0);

    if (sent == -1) {
        perror("send failed");

        if (errno == EPIPE) {
            std::cerr << "Broken pipe: Erlang side likely disconnected
." << std::endl;
            close(erlang_socket);
            erlang_socket = -1;
        }

        return;
    }
    std::cout << "Sent to Erlang: " << command << std::endl;
}

/**
 * @brief Handles communication with a client.
 * @param clientSocket The client socket file descriptor.
 */
void Server::handleClient(int clientSocket)
{
    std::string buffer;
    char tempBuf[4096];

    while (running) {
        int valread = read(clientSocket, tempBuf, sizeof(tempBuf));
        if (valread <= 0) {
            if (valread == 0 || errno == ECONNRESET)
                break;
            if (errno == EWOULDBLOCK || errno == EAGAIN) {
                std::this_thread::sleep_for(std::chrono::milliseconds
(10));
                continue;
            }
            perror("Read failed");
            break;
        }

        buffer.append(tempBuf, valread);

        size_t pos;
        size_t offset = 0;
        while ((pos = buffer.find('\n', offset)) != std::string::npos)
 {
            std::string_view message(buffer.data() + offset, pos -
```

```
    offset);
            parseErlangMessage(message.data(), message.size());
            offset = pos + 1;
        }
        buffer.erase(0, offset);
    }

    close(clientSocket);
}

/**
 * @brief Cleans up finished client threads.
 */
void Server::cleanupThreads()
{
    std::lock_guard<std::mutex> lock(clientsMutex);
    auto it = clientThreads.begin();
    while (it != clientThreads.end()) {
        if (it->joinable()) {
            it->join();
            it = clientThreads.erase(it);
        } else {
            ++it;
        }
    }
}

/**
 * @brief Stops the server and worker threads.
 */
void Server::stop()
{
    running = false;
    {
        std::lock_guard lock(queueMutex);
        shutdownWorker = true;
    }
    queueCond.notify_all();
    if (workerThread.joinable())
        workerThread.join();
}

/**
 * @brief Parses messages from Erlang and adds samples to queue.
 * @param buffer The message buffer.
 * @param len Length of the message.
 */
void Server::parseErlangMessage(const char *buffer, int len)
{
    if (buffer == nullptr || len <= 0 || len >= 1024) {
        std::cerr << "error" << std::endl;
        return;
    }

    std::string message(buffer, len);

```

```
    // Parse message components
    size_t nPos = message.find("n:");
    size_t bPos = message.find(";b:");
    size_t ePos = message.find(";e:");
    size_t sPos = message.find(";s:");

    if (nPos != 0 || bPos == std::string::npos || ePos == std::string
::npos || sPos == std::string::npos) {
        std::cerr << "Failed to parse message: " << message << std::
endl;
        return;
    }

    // Extract message fields
    std::string name = message.substr(2, bPos - 2);
    std::string bStr = message.substr(bPos + 3, ePos - (bPos + 3));
    std::string eStr = message.substr(ePos + 3, sPos - (ePos + 3));
    std::string statusStr = message.substr(sPos + 3);

    // Convert to sample data
    uint64_t startTime = std::stoull(bStr);
    uint64_t endTime = std::stoull(eStr);
    Sample sample;
    Status status = Status::SUCCESS;

    if (statusStr == TIMEOUT || statusStr == FAIL) {
        status = (statusStr == TIMEOUT) ? Status::TIMEDOUT : Status::
FAILED;
    } else if (statusStr != EXEC_OK) {
        std::cerr << "Unknown status: " << statusStr << std::endl;
        return;
    }

    double long elapsed = (endTime - startTime) / 1'000'000'000.0L;
    sample = {startTime, endTime, elapsed, status};

    // Add to processing queue
    {
        std::lock_guard lock(queueMutex);
        sampleQueue.emplace(name, sample);
    }
    queueCond.notify_one();
}
```

Appendix G

C++ Header Files

G.1 Root

G.1.1 Application.h

```
#pragma once
#include "diagram/Observable.h"
#include "diagram/System.h"
#include "server/Server.h"
#include <functional>
#include <iostream>
#include <memory>
#include <mutex>
#include <vector>

struct SystemDiff {
    std::vector<std::string> addedProbes;
    std::vector<std::string> removedProbes;
    std::vector<std::string> changedProbes;

    std::vector<std::string> addedOutcomes;
    std::vector<std::string> removedOutcomes;
    std::vector<std::string> changedOutcomes;

    std::vector<std::string> addedOperators;
    std::vector<std::string> removedOperators;
    std::vector<std::string> changedOperators;
};

class Application
{

    std::shared_ptr<System> system = nullptr;
    std::vector<std::function<void()>> observers; // List of functions
     to notify
    Server *server = nullptr;
    bool componentsDiffer(const std::shared_ptr<Observable> &a, const
    std::shared_ptr<Observable> &b);
```

```

    SystemDiff diffWith(System &newSystem);

    Application();
    void notifyObservers();

public:
    static Application &getInstance();
    void setServer(Server *);
    void setSystem(System newSystem);
    std::shared_ptr<System> getSystem();
    void addObserver(std::function<void()> callback);
    void sendDelayChange(std::string &, double);

    bool startCppServer(const std::string &&, int);
    void stopCppServer();

    void setErlangEndpoint(const std::string &&, int);
    void setStubRunning(bool running);
};
```

G.2 dashboard

G.2.1 ColorRegistry.h

```
#pragma once
#include <QColor>
#include <string>
#include <unordered_map>

class ColorRegistry
{
public:
    static QColor getColorFor(const std::string &name);

private:
    static QColor generateDistinctColor(int index);
    static std::unordered_map<std::string, QColor> colorMap;
};
```

G.2.2 CustomLegendEntry.h

```


#pragma once

#include <QLabel>
#include <QWidget>
#include <qboxlayout.h>

class CustomLegendEntry : public QWidget
{
    Q_OBJECT
```

```
    QHBoxLayout *layout;
    QLabel *colorBox;
    QLabel *nameLabel;

public:
    CustomLegendEntry(const QString &name, const QColor &color,
   QWidget *parent = nullptr);
};
```

G.2.3 CustomLegendPanel.h

```
#pragma once

#include <QVBoxLayout>
#include <QWidget>
#include <map>
#include <qscrollarea.h>

/**
 * @class CustomLegendPanel
 * @brief A scrollable widget that displays a legend for a plot.
 */
class CustomLegendPanel : public QWidget
{
    Q_OBJECT

public:
    explicit CustomLegendPanel(QWidget *parent = nullptr);

    /**
     * @brief Adds a new entry to the legend.
     * @param name The display name for the entry.
     * @param color The color for the entry.
     */
    void addEntry(const QString &name, const QColor &color);

    /**
     * @brief Removes an entry from the legend by name.
     * @param name The name of the entry to remove.
     */
    void removeEntry(const QString &name);

    /**
     * @brief Clears all entries from the legend.
     */
    void clear();

private:
    std::map<QString, QWidget *> legendEntries;  ///< Map of legend
   entries by name
    QVBoxLayout *mainLayout;                    ///< Main layout of
   the panel
    QVBoxLayout *legendLayout;                  ///< Layout containing
    the legend entries
```

```
    QScrollArea *scrollArea;                  ///< Scroll area for
   the legend content
    QWidget *scrollContent;                   ///< Widget containing
    the scrollable content
};
```

G.2.4 DQPlotController.h

```

#ifndef DQPLOTCONTROLLER_H
#define DQPLOTCONTROLLER_H

#include <qlineseries.h>
#pragma once

// Project includes
#include "../diagram/System.h"
#include "DeltaQPlot.h"

// Qt includes
#include <QString>
#include <QtCharts/QLineSeries>

// C++ includes
#include <map>
#include <memory>
#include <string>
#include <vector>

// All series pertaining to an outcome
struct OutcomeSeries {
    QLineSeries *outcomeS;
    QLineSeries *lowerBoundS;
    QLineSeries *upperBoundS;
    QLineSeries *meanS;
    QLineSeries *qtaS;
};

// All series pertaining to a probe
struct ExpressionSeries {
    QLineSeries *obsS;
    QLineSeries *obsLowerBoundS;
    QLineSeries *obsUpperBoundS;
    QLineSeries *obsMeanS;
    QLineSeries *calcS;
    QLineSeries *calcLowerBoundS;
    QLineSeries *calcUpperBoundS;
    QLineSeries *calcMeanS;
    QLineSeries *qtaS;
};

class DeltaQPlot;

/**
 * @class DQPlotController
```

```
 * @brief Controls the data management and update logic of DeltaQPlot.
 */
class DQPlotController
{
public:
    /**
     * @brief Constructor with associated plot and selected components
    .
     */
    DQPlotController(DeltaQPlot *plot, const std::vector<std::string>
   &selectedItems);

    /**
     * @brief Destructor.
     */
    ~DQPlotController();

    /**
     * @brief Checks if a component is already being plotted.
     */
    bool containsComponent(std::string name);

    /**
     * @brief Updates the plot according to a new list of selected
   components.
     */
    void editPlot(const std::vector<std::string> &selectedItems);

    /**
     * @brief Adds a new component (probe or outcome) to the plot.
     */
    void addComponent(const std::string &name, bool isOutcome);

    QLineSeries *createAndAddLineSeries(const std::string &legendName)
   ;

    void addOutcomeSeries(const std::string &name);

    void removeOutcomeSeries(const std::string &name);

    void addExpressionSeries(const std::string &name, bool isProbe);

    void removeExpressionSeries(const std::string &name, bool isProbe)
   ;
    /**
     * @brief Returns a list of names of all currently plotted
   components.
     */
    std::vector<std::string> getComponents();
    /**
     * @brief Removes a plotted component by name (const reference).
     */
    void removeComponent(const std::string &name);

    /**
```

```
     * @brief Updates the data of all series based on provided time
    range and bin width.
     */
    void update(uint64_t timeLowerBound, uint64_t timeUpperBound);

    void setTitle();

    bool isEmptyAfterReset();

private:
    double updateOutcome(OutcomeSeries &, const std::shared_ptr<
    Outcome> &, uint64_t, uint64_t);

    double updateProbe(ExpressionSeries &, std::shared_ptr<Probe> &,
    uint64_t, uint64_t);

    double updateOperator(ExpressionSeries &, std::shared_ptr<Operator
    > &, uint64_t, uint64_t);

    void updateExpression(ExpressionSeries &, DeltaQRepr &&,
    DeltaQRepr &&, QTA &&, double maxDelay);

    DeltaQPlot *plot;

    std::mutex updateMutex;
    std::mutex resetMutex;

    std::map<std::string, std::pair<OutcomeSeries, std::shared_ptr<
    Outcome>>> outcomes;
    std::map<std::string, std::pair<ExpressionSeries, std::shared_ptr<
    Probe>>> probes;
    std::map<std::string, std::pair<ExpressionSeries, std::shared_ptr<
    Operator>>> operators;
};

#endif // DQPLOTCONTROLLER_H
```

G.2.5 DQPlotList.h

```
#pragma once

#ifndef DQ_PLOT_LIST_H
#define DQ_PLOT_LIST_H

#include "DQPlotController.h"
#include <QCheckBox>
#include <QListWidget>
#include <QPushButton>
#include <QVBoxLayout>
#include <QWidget>

class DQPlotController;

/**
 * @class DQPlotList
```

```
 * @brief A widget that displays and manages lists of available and selected plot components for a selected plot.
 *
 */
class DQPlotList : public QWidget
{
    Q_OBJECT

public:
    explicit DQPlotList(DQPlotController *controller, QWidget *parent = nullptr);

    /**
     * @brief Resets the widget's state.
     */
    void reset();

    /**
     * @brief Checks if the widget is empty after reset.
     * @return true if no components are selected, false otherwise.
     */
    bool isEmptyAfterReset();

    /**
     * @brief Updates both the available and selected lists.
     */
    void updateLists();

    /**
     * @brief Default destructor.
     */
    ~DQPlotList() = default;

private Q_SLOTS:
    /**
     * @brief Handles confirmation of selected components to add.
     */
    void onConfirmSelection();

    /**
     * @brief Handles removal of selected components.
     */
    void onRemoveSelection();

private:
    /**
     * @brief Adds an item to either the available or selected list.
     * @param name The name of the component.
     * @param isSelected Whether the item should be in the selected list.
     * @param category The category for the item (used for grouping).
     */
    void addItemToList(const std::string &name, bool isSelected, const QString &category);

```

```
    DQPlotController *controller; ///< The controller managing plot components.

    QListWidget *selectedList;    ///< List widget showing currently selected components.
    QListWidget *availableList;   ///< List widget showing available components.

    QPushButton *addButton;       ///< Button to add selected components.
    QPushButton *removeButton;    ///< Button to remove selected components.
};

#endif // DQ_PLOT_LIST_H
```

G.2.6 DelaySettingsWidget.h

```
#pragma once

#include <QComboBox>
#include <QHBoxLayout>
#include <QLabel>
#include <QPushButton>
#include <QSlider>
#include <QSpinBox>
#include <QVBoxLayout>
#include <QWidget>
#include <cmath>

/**
 * @brief A widget for configuring delay parameters for a selected observable.
 */
class DelaySettingsWidget : public QWidget
{
    Q_OBJECT

public:
    explicit DelaySettingsWidget(QWidget *parent = nullptr);

    /**
     * @brief Populates the observable combo box using the system's observable list.
     */
    void populateComboBox();

    /**
     * @brief Computes the maximum delay in milliseconds based on the slider and spinbox values.
     * @return Maximum delay in milliseconds.
     */
    double getMaxDelayMs() const;

Q_SIGNALS:
```

```
    /**
     * @brief Signal emitted when delay parameters have been changed and saved.
     */
    void delayParametersChanged();

private Q_SLOTS:
    /**
     * @brief Updates the label showing the current maximum delay based on UI values.
     */
    void updateMaxDelay();

    /**
     * @brief Handles saving the currently selected delay parameters to the system.
     */
    void onSaveDelayClicked();

    /**
     * @brief Loads the saved settings for the currently selected observable.
     */
    void loadObservableSettings();

private:
    QVBoxLayout *mainLayout;            ///< Main layout container.
    QLabel *settingsLabel;             ///< Label describing the purpose of the widget.
    QLabel *maxDelayLabel;             ///< Label showing the computed max delay.

    QHBoxLayout *settingsLayout;       ///< Layout for parameter controls.
    QComboBox *observableComboBox;     ///< Combo box for selecting observables.
    QSlider *delaySlider;              ///< Slider to set delay exponent.
    QSpinBox *binSpinBox;              ///< Spin box to set number of bins.

    QPushButton *saveDelayButton;      ///< Button to save the delay configuration.
};
```

G.2.7 DeltaQPlot.h

```

#ifndef DELTAQPLOT_H
#define DELTAQPLOT_H

#include "CustomLegendPanel.h"
#include <qboxlayout.h>
#pragma once

```

```
// Qt includes
#include <QChartView>
#include <QLineSeries>
#include <QToolButton>
#include <QValueAxis>
// C++ includes
#include <string>
#include <vector>

class DQPlotController;
class DQPlotList;

/**
 * @class DeltaQPlot
 * @brief A class representing a DeltaQ chart view that allows
   visualization of probes over time.
 */
class DeltaQPlot : public QWidget
{
    Q_OBJECT

public:
    /**
     * @brief Constructs a DeltaQPlot with selected components.
     * @param selectedItems List of selected component names.
     * @param parent Parent widget.
     */
    explicit DeltaQPlot(const std::vector<std::string> &selectedItems,
    QWidget *parent = nullptr);

    /**
     * @brief Destructor.
     */
    ~DeltaQPlot();

    /**
     * @brief Adds a QLineSeries to the chart.
     * @param series Pointer to the line series.
     * @param name Name of the series.
     */
    void addSeries(QLineSeries *series, const std::string &name);

    /**
     * @brief Updates the plot data using provided time range and bin
   width.
     */
    void update(uint64_t timeLowerBound, uint64_t timeUpperBound);

    /**
     * @brief Removes a series from the chart.
     * @param series Series to remove.
     */
    void removeSeries(QAbstractSeries *series);

    /**
     * @brief Updates plot with new set of selected components.
```

```
     */
    void editPlot(const std::vector<std::string> &selectedItems);

    /**
     * @brief Gets list of currently plotted component names.
     */
    std::vector<std::string> getComponents();

    bool isEmptyAfterReset();

    /**
     * @brief Updates an existing series with new data points.
     */
    void updateSeries(QLineSeries *series, const QVector<QPointF> &
   data);

    void updateXRange(double xRange);
    /**
     * @brief Returns the associated plot list.
     */
    DQPlotList *getPlotList();

    void setTitle(QString &&);

protected:
    /**
     * @brief Handles mouse press events on the chart.
     */
    void mousePressEvent(QMouseEvent *event) override;

Q_SIGNALS:
    /**
     * @brief Emitted whenout
    QHBox this plot is selected by the user.
     */
    void plotSelected(DeltaQPlot *plot);

private:
    QHBoxLayout *layout;
    QToolButton *toggleButton;
    QChartView *chartView;
    QChart *chart;

    QValueAxis *axisX;
    QValueAxis *axisY;

    QLineSeries *operationSeries;
    DQPlotController *controller;
    DQPlotList *plotList;

    CustomLegendPanel *legendPanel;
};

#endif // DELTAQPLOT_H
```

G.2.8 MainWindow.h

```cpp
#pragma once
#include "../diagram/System.h"
#include "DeltaQPlot.h"
#include "NewPlotList.h"
#include "ObservableSettings.h"
#include "Sidebar.h"
#include "StubControlWidget.h"
#include "TriggersTab.h"

#include <QHBoxLayout>
#include <QListWidget>
#include <QMainWindow>
#include <QPushButton>
#include <QTimer>
#include <QVBoxLayout>
#include <qboxlayout.h>
#include <qwidget.h>

/**
 * @class MainWindow
 * @brief The main application window containing plots and control
   panels.
 */
class MainWindow : public QMainWindow
{
    Q_OBJECT

    QHBoxLayout *mainLayout; ///< Main horizontal layout

    QScrollArea *scrollArea; ///< Scroll area for plots
    QGridLayout *plotLayout; ///< Grid layout for plot arrangement
    QWidget *plotContainer; ///< Container widget for plots

    QWidget *centralWidget; ///< Central widget for main layout

    QThread *timerThread; ///< Thread for update timer
    QTimer *updateTimer; ///< Timer for periodic updates

    QWidget *sideContainer; ///< Container for side panels
    QVBoxLayout *sideLayout; ///< Layout for side panels
    QTabWidget *sideTabWidget; ///< Tab widget for side panels
    TriggersTab *triggersTab; ///< Triggers configuration panel
    Sidebar *sidebar; ///< Main sidebar control panel
    ObservableSettings *observableSettings; ///< Observable settings
   panel

    StubControlWidget *stubWidget; ///< Stub control widget (
   placeholder)
    QPushButton *addPlotButton; ///< Button to add new plots

    QMap<DeltaQPlot *, QWidget *> plotContainers; ///< Map of plots to
    their containers
    uint64_t timeLowerBound; ///< Lower time bound for data updates

```

```
    std::mutex plotDelMutex; ///< Mutex for plot deletion safety
    std::mutex updateMutex; ///< Mutex for update operations

    int samplingRate {200}; ///< Current sampling rate in milliseconds

public:
    MainWindow(QWidget *parent = nullptr);

    ~MainWindow();

    /**
     * @brief Resets the window state, cleaning up empty plots.
     */
    void reset();

private Q_SLOTS:
    /**
     * @brief Updates all plots with new data.
     */
    void updatePlots();

    /**
     * @brief Handles adding new plots from sidebar selection.
     */
    void onAddPlotClicked();

    /**
     * @brief Removes a specific plot.
     * @param plot The plot to remove.
     */
    void onRemovePlot(DeltaQPlot *plot);

    /**
     * @brief Handles plot selection changes.
     * @param plot The newly selected plot.
     */
    void onPlotSelected(DeltaQPlot *plot);

protected:
    /**
     * @brief Handles context menu events for plot management.
     * @param event The context menu event.
     */
    void contextMenuEvent(QContextMenuEvent *event) override;

    /**
     * @brief Handles window resize events to adjust plot sizes.
     * @param event The resize event.
     */
    void resizeEvent(QResizeEvent *event) override;
};
```

G.2.9 NewPlotList.h

```
#pragma once
```

```

#ifndef NEW_PLOT_LIST_H
#define NEW_PLOT_LIST_H

#include "../diagram/System.h"
#include <QCheckBox>
#include <QListWidget>
#include <qlistwidget.h>
#include <qwidget.h>

/**
 * @brief A list widget for selecting observables to create a new plot
   .
 *
 */
class NewPlotList : public QListWidget
{
    Q_OBJECT

public:
    explicit NewPlotList(QWidget *parent = nullptr);

    /**
     * @brief Gets the names of all currently selected observables.
     * @return A vector of strings representing selected observables.
     */
    std::vector<std::string> getSelectedItems();

    /**
     * @brief Deselects all currently selected items in the list.
     */
    void deselectAll();

    /**
     * @brief Clears the list and repopulates it with updated
   observables from the system.
     */
    void reset();

private:
    /**
     * @brief Adds observable items to the list from the current
   system.
     */
    void addItems();
};

#endif // NEW_PLOT_LIST_H
```

G.2.10 ObservableSettings.h

```
#ifndef OBS_SETTINGS_H
#define OBS_SETTINGS_H

#include <qlabel.h>
```

```cpp
#pragma once
#include <QWidget>

#include "DelaySettingsWidget.h"
#include "QTAInputWidget.h"

class ObservableSettings : public QWidget
{
    Q_OBJECT

    QVBoxLayout *layout;

    QLabel *delayLabel;
    DelaySettingsWidget *delaySettingsWidget;

    QLabel *qtaLabel;
    QTAInputWidget *qtaInputWidget;

public:
    explicit ObservableSettings(QWidget *parent = nullptr);
};

#endif
```

G.2.11 SamplingRateWidget.h

```cpp
#pragma once

#include <QLabel>
#include <QPushButton>
#include <QSlider>
#include <QVBoxLayout>
#include <QWidget>

/**
 * @class SamplingRateWidget
 * @brief A widget for saving sampling  rate intervals.
 *
 * Provides a slider interface to select from predefined sampling
   rates
 * (in milliseconds) and emits the selected rate when saved.
 */
class SamplingRateWidget : public QWidget
{
    Q_OBJECT

public:
    explicit SamplingRateWidget(QWidget *parent = nullptr);

private Q_SLOTS:
    /**
     * @brief Handles slider value changes to update the displayed
   rate.
     * @param value The current slider index (0-based).
     */
```

```
    void onSliderValueChanged(int value);

    /**
     * @brief Handles save button clicks to emit the selected rate.
     */
    void onSaveClicked();

Q_SIGNALS:
    /**
     * @brief Emitted when a new sampling rate is saved.
     * @param milliseconds The selected sampling rate in milliseconds.
     */
    void onSamplingRateChanged(int milliseconds);

private:
    QSlider *slider; ///< Slider for selecting sampling rate
    QLabel *valueLabel; ///< Displays the current sampling rate
    QPushButton *saveButton; ///< Button to save the selected rate
    QVector<int> samplingRates; ///< Available sampling rate options (
   ms)
};
```

G.2.12 QTAInputWidget.h

```
#ifndef QTAINPUTWIDGET_H
#define QTAINPUTWIDGET_H

#include <QWidget>
#include <QLineEdit>
#include <QComboBox>
#include <QFormLayout>
#include <QLabel>
#include <QPushButton>

/**
 * @class QTAInputWidget
 * @brief A Qt widget for configuring QTAs for observables.
 */
class QTAInputWidget : public QWidget
{
    Q_OBJECT

public:
    explicit QTAInputWidget(QWidget *parent = nullptr);

    /**
     * @brief Gets the 25th percentile value (in seconds).
     * @return The value entered in the 25th percentile field.
     */
    double getPerc25() const;

    /**
     * @brief Gets the 50th percentile (median) value (in seconds).
     * @return The value entered in the 50th percentile field.
     */
```

```
    double getPerc50() const;

    /**
     * @brief Gets the 75th percentile value (in seconds).
     * @return The value entered in the 75th percentile field.
     */
    double getPerc75() const;

    /**
     * @brief Gets the maximum allowed CDF value (0 to 1).
     * @return The value entered in the CDF max field.
     */
    double getCdfMax() const;

    /**
     * @brief Gets the currently selected observable name.
     * @return The name of the selected observable.
     */
    QString getSelectedObservable() const;

public Q_SLOTS:
    /**
     * @brief Populates the observable dropdown with available
   observables.
     * @note Called automatically when the system updates.
     */
    void populateComboBox();

    /**
     * @brief Loads QTA settings for the selected observable into the
   UI fields.
     * @note Triggered when the dropdown selection changes.
     */
    void loadObservableSettings();

    /**
     * @brief Saves the current QTA settings to the system.
     * @note Called when the "Save" button is clicked.
     * @throws std::exception if validation fails (e.g., invalid CDF
   value).
     */
    void onSaveButtonClicked();

private:
    QComboBox *observableComboBox;   ///< Dropdown to select an
   observable (probe/outcome).
    QLineEdit *perc25Edit;           ///< Input field for the 25th
   percentile (seconds).
    QLineEdit *perc50Edit;           ///< Input field for the 50th
   percentile (seconds).
    QLineEdit *perc75Edit;           ///< Input field for the 75th
   percentile (seconds).
    QLineEdit *cdfMaxEdit;           ///< Input field for the max CDF
   value (0-1).
    QPushButton *saveButton;         ///< Button to save QTA settings.
```

```
    QLabel *qtaLabel;                         ///< Label describing the widget's purpose.
};

#endif // QTAINPUTWIDGET_H
```

G.2.13 Sidebar.h

```
#ifndef SIDEBAR_H
#define SIDEBAR_H

#include "DQPlotList.h"
#include "NewPlotList.h"
#include "SamplingRateWidget.h"
#include "SystemCreationWidget.h"
#include <QComboBox>
#include <QLabel>
#include <QPushButton>
#include <QSpinBox>
#include <QSplitter>
#include <QTextEdit>
#include <QVBoxLayout>
#include <QWidget>
#include <qboxlayout.h>

/**
 * @class Sidebar
 * @brief Main sidebar widget containing plot management controls and system configuration.
 *
 */
class Sidebar : public QWidget
{
    Q_OBJECT

    QVBoxLayout *newPlotListLayout; ///< Layout for new plot selection components
    QWidget *newPlotListWidget; ///< Container widget for new plot controls
    QLabel *newPlotLabel; ///< Label for new plot section
    NewPlotList *newPlotList; ///< List widget for selecting probes for new plots
    QPushButton *addNewPlotButton; ///< Button to create new plot

    QWidget *currentPlotWidget; ///< Container widget for current plot controls
    QVBoxLayout *currentPlotLayout; ///< Layout for current plot components
    QLabel *currentPlotLabel; ///< Label for current plot section
    DQPlotList *currentPlotList = nullptr; ///< List widget for managing current plot's probes

    QSplitter *mainSplitter; ///< Main splitter organizing sections vertically
    QVBoxLayout *layout; ///< Main layout of the sidebar
```

```

    SystemCreationWidget *systemCreationWidget; ///< Widget for system
     creation/configuration
    SamplingRateWidget *samplingRateWidget; ///< Widget for adjusting
   sampling rate

Q_SIGNALS:
    /**
     * @brief Emitted when the "Add plot" button is clicked.
     */
    void addPlotClicked();

    /**
     * @brief Emitted when sampling rate is changed.
     * @param milliseconds The new sampling rate in milliseconds.
     */
    void onSamplingRateChanged(int milliseconds);

private Q_SLOTS:
    /**
     * @brief Handles "Add plot" button click event.
     */
    void onAddPlotClicked();

    /**
     * @brief Handles sampling rate change events.
     * @param ms The new sampling rate in milliseconds.
     */
    void handleSamplingRateChanged(int ms);

public:
    /**
     * @brief Constructs a Sidebar widget.
     * @param parent The parent widget (optional).
     */
    explicit Sidebar(QWidget *parent = nullptr);

    /**
     * @brief Sets the current plot list widget.
     * @param currentPlotList The DQPlotList widget to display.
     */
    void setCurrentPlotList(DQPlotList *currentPlotList);

    /**
     * @brief Hides the current plot management section.
     */
    void hideCurrentPlot();

    /**
     * @brief Gets the new plot list widget.
     * @return Pointer to the NewPlotList widget.
     */
    NewPlotList *getPlotList() const
    {
        return newPlotList;
    }
```

```

    /**
     * @brief Clears new plot selection after plot creation.
     */
    void clearOnAdd();
};

#endif
```

G.2.14 SnapshotViewerWindow.h

```
#pragma once

#include <QChartView>
#include <QComboBox>
#include <QLabel>
#include <QSlider>
#include <QWidget>

#include "../maths/Snapshot.h"
#include <map>

/**
 * @brief A QWidget-based window for visualizing snapshots from fired
   triggers.
 */
class SnapshotViewerWindow : public QWidget
{
    Q_OBJECT

public:
    explicit SnapshotViewerWindow(std::vector<Snapshot> &snapshotList,
    QWidget *parent = nullptr);

    /**
     * @brief Sets the snapshots to display in the viewer.
     * @param snapshotList A vector of Snapshots.
     */
    void setSnapshots(std::vector<Snapshot> &snapshotList);

private Q_SLOTS:
    /**
     * @brief Slot triggered when the observable selection changes.
     * @param name Name of the newly selected observable.
     */
    void onObservableChanged(const QString &name);

    /**
     * @brief Slot triggered when the time slider is moved.
     * @param value Index of the snapshot in the selected observable.
     */
    void onTimeSliderChanged(int value);

private:
    /**
```

```
     * @brief Updates the chart view based on the current observable and time index.
     */
    void updatePlot();

    QChartView *chartView;                          ///< Chart view for plotting the snapshot data.
    QComboBox *observableSelector;                  ///< Dropdown for selecting an observable.
    QSlider *timeSlider;                            ///< Slider for selecting time index.
    QLabel *timeLabel;                              ///< Label showing the current time.

    std::map<std::string, Snapshot> snapshots;///< Map of observable name to snapshot.
    std::string currentObservable;                  ///< Currently selected observable.
};
```

G.2.15 StubControlWidget.h

```
#pragma once
#include "src/Application.h"
#include <QApplication>
#include <QGridLayout>
#include <QGroupBox>
#include <QHBoxLayout>
#include <QLabel>
#include <QLineEdit>
#include <QPushButton>
#include <QVBoxLayout>
#include <QWidget>

class StubControlWidget : public QWidget
{
    Q_OBJECT
public:
    StubControlWidget(QWidget *parent = nullptr);

private:
    // Erlang controls
    QPushButton *startErlangButton;
    QPushButton *stopErlangButton;

    // Server controls
    QPushButton *startServerButton;
    QPushButton *stopServerButton;
    QLineEdit *serverIpEdit;
    QLineEdit *serverPortEdit;

    // Erlang receiver settings
    QLineEdit *erlangReceiverIpEdit;
    QLineEdit *erlangReceiverPortEdit;
    QPushButton *setErlangEndpointButton;
```

```cpp

    QVBoxLayout *mainLayout;

private Q_SLOTS:
    // Erlang slots
    void onStartErlangClicked();
    void onStopErlangClicked();

    // Server slots
    void onStartServerClicked();
    void onStopServerClicked();

    // Erlang endpoint slot
    void onSetErlangEndpointClicked();
};
```

G.2.16 SystemCreationWidget.h

```cpp
#ifndef SYSTEMCREATIONWIDGET_H
#define SYSTEMCREATIONWIDGET_H

#include <QHBoxLayout>
#include <QLabel>
#include <QPushButton>
#include <QTextEdit>
#include <QVBoxLayout>
#include <QWidget>

/**
 * @class SystemCreationWidget
 * @brief A QWidget that allows users to create, edit, load, and save
   system definitions.
 */
class SystemCreationWidget : public QWidget
{
    Q_OBJECT

public:
    explicit SystemCreationWidget(QWidget *parent = nullptr);

    /**
     * @brief Retrieves the current system text from the editor.
     * @return The system text as a std::string.
     */
    std::string getSystemText() const;

    /**
     * @brief Sets the system text in the editor.
     * @param text The new system definition text.
     */
    void setSystemText(const std::string &text);

Q_SIGNALS:
    /**
     * @brief Emitted when the system is successfully updated.
```

```
     */
    void systemUpdated();

    /**
     * @brief Emitted when the system is successfully saved.
     */
    void systemSaved();

    /**
     * @brief Emitted when a system is successfully loaded.
     */
    void systemLoaded();

private Q_SLOTS:
    /**
     * @brief Parses the text and updates the system instance.
     */
    void onUpdateSystem();

    /**
     * @brief Saves the current system text to a file.
     */
    void saveSystemTo();

    /**
     * @brief Loads a system from a file and updates the editor.
     */
    void loadSystem();

private:
    QTextEdit *systemTextEdit;           ///< Editor widget for system
   text.
    QPushButton *updateSystemButton;     ///< Button to update system.
    QPushButton *saveSystemButton;       ///< Button to save system.
    QPushButton *loadSystemButton;       ///< Button to load system.
    QLabel *systemLabel;                 ///< Label describing the
   editor.

    QVBoxLayout *mainLayout;             ///< Layout for the main
   components.
    QHBoxLayout *buttonLayout;           ///< Layout for the buttons.
};

#endif // SYSTEMCREATIONWIDGET_H
```

G.2.17 TriggersTab.h

```
#pragma once

#include <QCheckBox>
#include <QComboBox>
#include <QFormLayout>
#include <QListWidget>
#include <QMap>
#include <QSpinBox>
```

```cpp
#include <QVBoxLayout>
#include <QWidget>

#include "../Application.h"
#include "src/maths/TriggerManager.h"

/**
 * @class TriggersTab
 * @brief Widget for managing and monitoring trigger conditions on
   observables.
 *
 * Provides UI for:
 * - Setting up trigger conditions (sample limits, QTA violations)
 * - Displaying triggered events
 * - Viewing snapshots of triggered states
 */
class TriggersTab : public QWidget
{
    Q_OBJECT

public:
    explicit TriggersTab(QWidget *parent = nullptr);

    ~TriggersTab();

    /**
     * @brief Adds a triggered message to the display list.
     * @param msg The message to display.
     */
    void addTriggeredMessage(const QString &msg);

private Q_SLOTS:
    /**
     * @brief Handles observable selection changes.
     * @param name The newly selected observable name.
     */
    void onObservableChanged(const QString &name);

    /**
     * @brief Handles trigger condition changes.
     */
    void onTriggerChanged();

    /**
     * @brief Handles triggered item clicks to show snapshots.
     * @param item The clicked list item.
     */
    void onTriggeredItemClicked(QListWidgetItem *item);

private:
    /**
     * @brief Captures snapshots when triggers are activated.
     * @param time The timestamp of the trigger event.
     * @param name The name of the observable that triggered.
     */
```

```
    void captureSnapshots(std::uint64_t time, const std::string &name);

    /**
     * @brief Populates the observable dropdown list.
     */
    void populateObservables();

    /**
     * @brief Updates checkbox states based on current triggers.
     */
    void updateCheckboxStates();

    QVBoxLayout *mainLayout;              ///< Main vertical layout
    QFormLayout *formLayout;              ///< Form layout for controls
    QComboBox *observableComboBox;        ///< Dropdown for observable selection

    QWidget *sampleLimitWidget;           ///< Container for sample limit controls
    QHBoxLayout *sampleLimitLayout;       ///< Layout for sample limit controls
    QCheckBox *sampleLimitCheckBox;       ///< Checkbox to enable sample limit trigger
    QSpinBox *sampleLimitSpinBox;         ///< Spinbox for sample limit threshold

    QCheckBox *qtaBoundsCheckBox;         ///< Checkbox for QTA bounds violation trigger

    QListWidget *triggeredList;           ///< List widget for triggered events

    /**
     * @brief Gets the currently selected observable.
     * @return Shared pointer to the current observable.
     * @throws std::runtime_error if system or observable doesn't exist.
     */
    std::shared_ptr<Observable> getCurrentObservable();

    static constexpr int sampleLimitThreshold = 500;      ///< Default sample limit threshold
    static constexpr double failureRateThreshold = 0.95;  ///< Failure rate threshold constant
};
```

G.3 diagram

G.3.1 Observable.h

```
#pragma once

#include "../maths/Snapshot.h"
```

```
#include "Sample.h"
#include "src/maths/TriggerManager.h"
#include <deque>
#include <math.h>
#include <mutex>

#define DELTA_T_BASE 0.001
#define MAX_DQ 30
class Observable
{
protected:
    std::string name;
    std::deque<Sample> samples;
    mutable bool sorted;

    std::deque<DeltaQ> confidenceIntervalHistory;
    double maxDelay {0.05};
    int deltaTExp {0}; // Exponent for dynamic binning
    int nBins {50}; // Number of bins

    TriggerManager triggerManager;

    ConfidenceInterval observedInterval;
    QTA qta;

    Snapshot observableSnapshot;

    std::mutex observedMutex;
    std::mutex samplesMutex;
    std::mutex paramMutex;

    bool recording = false;

private:
    void updateSnapshot(uint64_t timeLowerBound, DeltaQ &deltaQ);

public:
    /**
     * @brief Constructor an observable with its name
     */
    Observable(const std::string &name);

    virtual ~Observable() { };
    /**
     * Add a sample (outcome instance) to an observable
     */
    void addSample(const Sample &sample);
    /**
     * @brief Get all sample with endTime timeLowerBound -
    timeUpperBound
     * @return The sample in range timeLowerBound - timeUpperBound
     */
    std::vector<Sample> getSamplesInRange(std::uint64_t timeLowerBound
    , std::uint64_t timeUpperBound);
    /**
```

```
     * @brief Get observed DeltaQ in range timeLowerBound -
    timeUpperBound from snapshot, if it has not been calculated,
    calculate it
     * @return DeltaQ
     */
    DeltaQ getObservedDeltaQ(uint64_t, uint64_t);
    /**
     * @brief Calculate the observed DeltaQ in range timeLowerBound -
    timeUpperBound, add it to snapshot and ConfidenceInterval
     * @return The calculated DeltaQ
     */
    DeltaQ calculateObservedDeltaQ(uint64_t, uint64_t);

    /**
     * @brief Get DeltaQ representation for graphical plotting
     */
    DeltaQRepr getObservedDeltaQRepr(uint64_t, uint64_t);

    /**
     * @brief Set new parameters for a DeltaQ
     * @return new dMax
     */
    double setNewParameters(int newExp, int newNBins);

    double getBinWidth() const
    {
        return DELTA_T_BASE * std::pow(2, deltaTExp);
    }

    int getNBins() const
    {
        return nBins;
    }

    double getMaxDelay() const
    {
        return maxDelay;
    }

    QTA getQTA() const
    {
        return qta;
    }

    int getDeltaTExp() const
    {
        return deltaTExp;
    }

    const TriggerManager &getTriggerManager() const
    {
        return triggerManager;
    }

    /**
     * @brief Set recoding snapshot
```

```
     * @param bool is recording
     */
    void setRecording(bool);

    /**
     * @brief Set QTA for an observable
     * @param qta new QTA
     */
    void setQTA(const QTA &);
    /**
     * @brief add a trigger to an observable
     * @param type the observable type
     * @param condition the condition to evalute
     * @param action action to perform on trigger fired
     * @param enabled
     * @param sampleLimit sampleLimit sample limit for sample limit
    trigger
     */
    void addTrigger(TriggerType type, TriggerDefs::Condition condition
    , TriggerDefs::Action action, bool enabled, std::optional<int>
    sampleLimit);
    /**
     * @brief Remove observable trigger
     */
    void removeTrigger(TriggerType type);

    /**
     * @brief Get snapshot of observable
     */
    Snapshot getSnapshot();
    /**
     * @brief Get observable name
     * @return its name
     */
    [[nodiscard]] std::string getName() const &;
};
```

G.3.2 Operator.h

```
#pragma once

#include "../maths/DeltaQ.h"
#include "Observable.h"
#include "OperatorType.h"

/**
 * @class Operator This class represents an operator according to the
    DeltaQSD paradigm
 */
class Operator : public Observable
{
    OperatorType type;

    std::vector<double> probabilities; ///< The probabilities of the
    components inside the operator, only available for probabilistic
```

```
    operator

    std::vector<std::vector<std::shared_ptr<Observable>>> causalLinks;
    ///< The causal links for each children

    ConfidenceInterval calculatedInterval;

    std::mutex calcMutex;

    std::deque<DeltaQ> calculatedDeltaQHistory; ///< History for
    confidence intervals

public:
    Operator(const std::string &name, OperatorType);

    ~Operator();
    /**
     * @brief calculate a Calculated deltaQ with bounds timeLowerBound
    , timeUpperBound
     * @param timeLowerBound
     * @param timeUpperBound
     * @return calculated DeltaQ
     */
    DeltaQ calculateCalculatedDeltaQ(uint64_t, uint64_t);

    /**
     * @brief Get the representation of a calculated DeltaQ for
    plotting
     * @return the representation of a calculated DeltaQ
     */
    DeltaQRepr getCalculatedDeltaQRepr(uint64_t, uint64_t);

    void setProbabilities(const std::vector<double> &);

    /**
     * @brief Get the probabilities of a probabilistic operator
     * @return The probabilities
    */
    std::vector<double> getProbabilities()
    {
        return probabilities;
    }

    /**
     * @brief Get the links for a children
     * @return The links for a children
     */
    std::vector<std::shared_ptr<Observable>> getChildren();

    void setCausalLinks(std::vector<std::vector<std::shared_ptr<
    Observable>>> links)
    {
        causalLinks = links;
    }

```

```
    std::vector<std::vector<std::shared_ptr<Observable>>>
   getCausalLinks()
    {
        return causalLinks;
    }

    OperatorType getType()
    {
        return type;
    }
};
```

G.3.3 OperatorType.h

```
#pragma once
// AllToFinish, FirstToFinish, Probabilistic Choice
enum class OperatorType { ATF, FTF, PRB };
```

G.3.4 Outcome.h

```
/**
 * @author Francesco Nieri
 * @date 25/10/2024
 * Class representing an outcome O_n in a system
 */
#pragma once

#include "Observable.h"

class Outcome : public Observable
{
public:
    Outcome(const std::string &name);
    ~Outcome();
};
```

G.3.5 Probe.h

```
#pragma once
#include <string>
#include <vector>

#include "../maths/ConfidenceInterval.h"
#include "../maths/DeltaQ.h"
#include "Observable.h"
#include <map>
#include <memory>
#include <mutex>

/**
 * @class Class representing a probe containing causal link
 */
```

```
class Probe : public Observable
{
    std::vector<std::shared_ptr<Observable>> causalLinks;

    std::mutex calcMutex;

    ConfidenceInterval calculatedInterval;
    std::deque<DeltaQ> calculatedDeltaQHistory;

public:
    Probe(const std::string &name);

    /**
     * @brief Construct a probe with its causal links
     */
    Probe(const std::string &name, std::vector<std::shared_ptr<
    Observable>>);

    ~Probe();
    /**
     * @brief calculate a Calculated deltaQ with bounds timeLowerBound
    , timeUpperBound
     * @param timeLowerBound
     * @param timeUpperBound
     * @return calculated DeltaQ
     */
    DeltaQ calculateCalculatedDeltaQ(uint64_t timeLowerBound, uint64_t
    timeUpperBound);

    /**
     * @brief Get the representation of a calculated DeltaQ for
    plotting
     * @return the representation of a calculated DeltaQ
     */
    DeltaQRepr getCalculatedDeltaQRepr(uint64_t, uint64_t);

    std::vector<Bound> getBounds() const;

    std::vector<Bound> getObservedBounds() const;

    std::vector<Bound> getCalculatedBounds() const;


    void setCausalLinks(std::vector<std::shared_ptr<Observable>>
    newCausalLinks)
    {
        causalLinks = newCausalLinks;
    }

    std::vector<std::shared_ptr<Observable>> getCausalLinks()
    {
        return causalLinks;
    }
};
```

G.3.6 Sample.h

This struct represents an outcome instance.

```
#pragma once

#include <cstdint>

enum Status { SUCCESS, TIMEDOUT, FAILED };

/**
 * @struct Sample represent an outcome instance
 */
struct Sample {
    std::uint64_t startTime;
    std::uint64_t endTime;
    double long elapsedTime;
    Status status;
};
```

G.3.7 System.h

```
/**
 * @author Francesco Nieri
 * @date 26/10/2024
 * Class representing a DeltaQ system
 */
#pragma once

#include "../maths/DeltaQ.h"
#include "Observable.h"
#include "Operator.h"
#include "Outcome.h"
#include "Probe.h"
#include <memory>
#include <unordered_map>

class System
{
    std::unordered_map<std::string, std::shared_ptr<Outcome>> outcomes
    {}; ///< All outcome
    std::unordered_map<std::string, std::shared_ptr<Operator>>
   operators {}; ///< All operators
    std::unordered_map<std::string, std::shared_ptr<Probe>> probes {};
    /// < All probes
    std::unordered_map<std::string, std::shared_ptr<Observable>>
   observables {}; ///< The above grouped together

    std::string systemDefinitionText; ///< The definition of the
   system

    bool recordingTrigger = false;
    std::map<uint64_t, std::vector<Snapshot>> snapshots;

public:
```

```
    System() = default;

    [[nodiscard]] std::unordered_map<std::string, std::shared_ptr<
   Outcome>> &getOutcomes();

    [[nodiscard]] std::unordered_map<std::string, std::shared_ptr<
   Probe>> &getProbes();

    [[nodiscard]] std::unordered_map<std::string, std::shared_ptr<
   Operator>> &getOperators();

    [[nodiscard]] std::unordered_map<std::string, std::shared_ptr<
   Observable>> &getObservables();

    void setOutcomes(std::unordered_map<std::string, std::shared_ptr<
   Outcome>> outcomesMap);

    void setOperators(std::unordered_map<std::string, std::shared_ptr<
   Operator>> operatorsMap);

    void setProbes(std::unordered_map<std::string, std::shared_ptr<
   Probe>> probesMap);

    bool hasOutcome(const std::string &name);

    bool hasOperator(const std::string &name);

    std::shared_ptr<Outcome> getOutcome(const std::string &outcomeName
   );

    std::shared_ptr<Operator> getOperator(const std::string &);

    bool hasProbe(const std::string &name);

    std::shared_ptr<Probe> getProbe(const std::string &name);

    std::shared_ptr<Observable> getObservable(const std::string &
   observableName);

    void setSystemDefinitionText(std::string &text);

    std::string getSystemDefinitionText();

    void setObservableParameters(std::string &, int, int);

    /**
     * @brief Add outcome instance for an observable, if it exists
     */
    void addSample(std::string &componentName, Sample &sample);

    /**
     * @brief Set all observables to record snapshots
     */
     void setRecording(bool);

    bool isRecording() const;
```

```

    /**
     * @brief Add the snapshots of all observables for a trigger at time t
     */
    void getObservablesSnapshotAt(std::uint64_t);

    /**
     * Get the snapshots of all observables for a trigger at time t
     */
    std::map<std::uint64_t, std::vector<Snapshot>> getAllSnapshots();

    /**
     * @brief Get the name of all components
     */
    std::vector<std::string> getAllComponentsName();

    /**
     * @deprecated This may be used in the future
     * Calculate the resulting DeltaQ for the whole system
     */
    DeltaQ calculateDeltaQ();
};
```

G.4 ΔQ (maths)

G.4.1 ConfidenceInterval.h

```
#pragma once

#include "DeltaQ.h"
#include <vector>
// Upper and lower confidence bounds of a DeltaQ's CDF
struct Bound {
    double lowerBound {0};
    double upperBound {1};
    double mean {0};
};

/**
 * @class ConfidenceInterval
 * @brief Class representing the confidence interval of a window of DeltaQs
 */
class ConfidenceInterval
{
private:
    std::vector<Bound> bounds; ///< The bounds at each point
    unsigned int numBins;
    unsigned int size {0};
    std::vector<double> cdfSum; ///< The sum of the cdf at each bin
    std::vector<double> cdfSumSquares; ///< Variance at each bin
    std::vector<unsigned int> cdfSampleCounts; ///< Sample per bins
    double z {1}; ///< Confidence interval
```

```
    void updateConfidenceInterval();

public:
    /**
     * @brief Default constructor, set to 0 bins
     */
    ConfidenceInterval();
    /**
     * @brief Constructor for ConfidenceInterval with number of bins
     * @param numBins number of bins
     */
    ConfidenceInterval(int numBins);

    void setNumBins(int newNumBins);
    /**
     * @brief Add DeltaQ to intervals
     * @param DeltaQ DeltaQ to add
     */
    void addDeltaQ(const DeltaQ &);
    /**
     * @brief Remove DeltaQ from intervals
     * @param DeltaQ DeltaQ to remove
     */
    void removeDeltaQ(const DeltaQ &);

    /**
     * @brief Get current confidence bounds
     */
    std::vector<Bound> getBounds() const;
    /**
     * @brief Get number of bins
     * @return Number of bins
     */
    unsigned int getBins();
    /**
     * @brief Zero the confidence intervals
     */
    void reset();
};
```

G.4.2 DeltaQ.h

```
/**
 * @author: Francesco Nieri
 * @date 26/10/2024
 * Class representing a DeltaQ
 */

#ifndef DELTAQ_H
#define DELTAQ_H

#pragma once

#include "../diagram/Sample.h"
#include <array>
```

```cpp
#include <ostream>
#include <vector>

#include "QTA.h"

class DeltaQ
{
    double binWidth;
    std::vector<double> pdfValues;
    std::vector<double> cdfValues;
    int bins {0};

    QTA qta;
    unsigned int totalSamples {0};

    /**
     * Calculate PDF and CDF values given samples from an outcome
     */
    void calculateDeltaQ(std::vector<Sample> &samples);
    /**
     * Calculate CDF given PDF values
     */
    void calculateCDF();
    /**
     * Calculate PDF from a given CDF
     */
    void calculatePDF();

public:
    DeltaQ() = default;
    DeltaQ(double binWidth);
    DeltaQ(double binWidth, const std::vector<double> &values, bool
   isPdf);
    DeltaQ(double binWidth, std::vector<Sample> &);
    DeltaQ(double binWidth, std::vector<Sample> &, int);
    /**
     * Getters
     */
    [[nodiscard]] const std::vector<double> &getPdfValues() const;
    [[nodiscard]] const std::vector<double> &getCdfValues() const;
    [[nodiscard]] double getBinWidth() const;
    [[nodiscard]] int getBins() const;
    [[nodiscard]] double pdfAt(int x) const;
    [[nodiscard]] double cdfAt(int x) const;
    [[nodiscard]] const unsigned int getTotalSamples() const;
    [[nodiscard]] QTA getQTA() const;
    void calculateQuartiles(std::vector<Sample> &);
    void setBinWidth(double newWidth);
    /**
     * Operator Overloads
     */
    friend DeltaQ operator*(const DeltaQ &deltaQ, double constant);
    friend DeltaQ operator*(double constant, const DeltaQ &deltaQ);
    friend DeltaQ operator*(const DeltaQ &lhs, const DeltaQ &rhs);
    friend DeltaQ operator+(const DeltaQ &lhs, const DeltaQ &rhs);
    friend DeltaQ operator-(const DeltaQ &lhs, const DeltaQ &rhs);
```

```
    friend std::ostream &operator<<(std::ostream &os, const DeltaQ &
    deltaQ);

    /**
     * Comparison Operators
     */
    bool operator<(const DeltaQ &other) const;
    bool operator>(const DeltaQ &other) const;

    bool operator==(const DeltaQ &deltaQ) const;
    [[nodiscard]] std::string toString() const;
};

#endif // DELTAQ_H
```

G.4.3 DeltaQOperations.h

```
#pragma once

#include "DeltaQ.h"

/**
 * Perform discrete convolution between two DeltaQs
 */
DeltaQ convolve(const DeltaQ &lhs, const DeltaQ &rhs);
/**
 * Perform Fast Fourier Transform on two DeltaQs
 */
DeltaQ convolveFFT(const DeltaQ &lhs, const DeltaQ &rhs);
DeltaQ convolveN(const std::vector<DeltaQ> &deltaQs);

/**
 * Assume two independent outcomes with the same start event
 * All-to-finish outcome occurs when both end events occur
 * All-to-finish is defined as ΔQ_{LTF(A,B)} ΔQ_A * ΔQ_B
 */
DeltaQ allToFinish(const std::vector<DeltaQ> &deltaQs);

/**
 * Assume two independent outcomes with the same start event
 * First-to-finish outcome occurs when at least one end event occurs
 * We compute the probability that there are zero end events
 * First-to-finish is defined as
 * DeltaQ_{FTF(A, B)} = DeltaQ_A + DeltaQ_B - DeltaQ_A * DeltaQ_B
 */
DeltaQ firstToFinish(const std::vector<DeltaQ> &deltaQs);

/**
 *  Assume there are two possible outcomes OA and OB and
 * exactly one outcome is chosen during each occurrence of a start
   event
 * O_A occurs with probability p/(p+q)
 * O_B occurs with probability q/(p+q)
 * Therefore:
```

```
 * ΔQ_{PC(A,B)} = p/(p+q) ΔQ_A + q/(p+q) ΔQ_B
 */
DeltaQ probabilisticChoice(const std::vector<double> &probabilities,
    const std::vector<DeltaQ> &deltaQs);

DeltaQ rebin(const DeltaQ &source, double targetBinWidth);

/**
 * Choose the highest size from a list of DeltaQs
 */
int chooseLongestDeltaQSize(const std::vector<DeltaQ> &deltaQs);
```

G.4.4 DeltaQRepr.h

```

#pragma once

#include "ConfidenceInterval.h"
#include "DeltaQ.h"
#include <cstdint>
#include <vector>

/**
 * Class storing a DeltaQ representation for graphical plotting
 */

struct DeltaQRepr {
    std::uint64_t time;
    DeltaQ deltaQ;
    std::vector<Bound> bounds;

    DeltaQRepr()
        : time(0)
        , deltaQ()
        , bounds()
    {
    }

    // DeltaQRepr copy constructor
    DeltaQRepr(const DeltaQRepr &other)
        : time(other.time)
        , deltaQ(other.deltaQ)
        , bounds(other.bounds)
    {
    }

    // DeltaQRepr move semantics
    DeltaQRepr(DeltaQRepr &&other) noexcept
        : time(other.time)
        , deltaQ(std::move(other.deltaQ))
        , bounds(std::move(other.bounds))
    {
    }

    // DeltaQRepr assignment operators
```

```
    DeltaQRepr &operator=(const DeltaQRepr &other)
    {
        if (this != &other) {
            time = other.time;
            deltaQ = other.deltaQ;
            bounds = other.bounds;
        }
        return *this;
    }

    DeltaQRepr &operator=(DeltaQRepr &&other) noexcept
    {
        if (this != &other) {
            time = other.time;
            deltaQ = std::move(other.deltaQ);
            bounds = std::move(other.bounds);
        }
        return *this;
    }

    // Constructor with parameters
    DeltaQRepr(std::uint64_t t, const DeltaQ &dq, const std::vector<
   Bound> &b)
        : time(t)
        , deltaQ(dq)
        , bounds(b)
    {
    }
};
```

G.4.5 QTA.h

This class represent a sample QTA.

```
#ifndef QTA_H
#define QTA_H

#include <iostream>
#include <limits>
#define QTA_EPSILON std::numeric_limits<double>::epsilon()
/**
 * @struct QTA Quantitative Timeliness Agreement for an observable
 */
struct QTA {
    double perc_25 {0}; ///< 25 percentile value
    double perc_50 {0}; ///< 50 percentile value
    double perc_75 {0}; ///< 75 percentile value
    double cdfMax {0}; ///< Least failure rate
    bool defined = false; ///< If defined or not

    static QTA create(double p25, double p50, double p75, double cdf)
    {
        return QTA::create(p25, p50, p75, cdf, true);
    }

```

```cpp
    static QTA create(double p25, double p50, double p75, double cdf,
    bool isDefined)
    {
        if (!(p25 <= p50 && p50 <= p75)) {
            throw std::invalid_argument("Percentiles must be ordered:
    perc_25 < perc_50 < perc_75.");
        }
        if (cdf < 0.0 || cdf - 1.0 > QTA_EPSILON) {
            throw std::invalid_argument("cdfMax must be between 0 and
    1 (exclusive lower bound, inclusive upper).");
        }

        return QTA {p25, p50, p75, cdf, isDefined};
    }
};

#endif // QTA_H
```

G.4.6 Snapshot.h

```cpp
#pragma once

#include "ConfidenceInterval.h"
#include "DeltaQ.h"
#include "DeltaQRepr.h"
#include <map>
#include <optional>

/**
 * @class Snapshot
 * @brief Represents a snapshot of observed and calculated DeltaQ
   values along with QTAs
 */
class Snapshot
{
    std::string observableName; ///< Name of the observable being
    tracked.
    std::map<uint64_t, DeltaQRepr> observedDeltaQs; ///< Map of
    observed DeltaQ values at times t.
    std::map<uint64_t, DeltaQRepr> calculatedDeltaQs; ///< Map of
    calculated DeltaQ values at times t
    std::map<uint64_t, QTA> QTAs; ///< Map of QTAs at time t

public:
    /// @brief Default constructor.
    Snapshot() = default;

    // --- Observed DeltaQ Methods ---
    /**
     * @brief Adds an observed DeltaQ to the snapshot.
     * @param timestamp The lower time bound at which the DeltaQ was
    observed.
     * @param deltaQ The DeltaQ value to store.
     * @param bounds Confidence interval bounds for the DeltaQ.
     */
```

```cpp
    void addObservedDeltaQ(std::uint64_t timestamp, const DeltaQ &deltaQ, const std::vector<Bound> &bounds);

    /**
     * @brief Retrieves the oldest observed DeltaQ.
     * @return The oldest DeltaQRepr (based on timestamp order).
     */
    DeltaQ getOldestObservedDeltaQ() const;

    /// @brief Removes the oldest observed DeltaQ entry (FIFO order).
    void removeOldestObservedDeltaQ();

    // --- Calculated DeltaQ Methods ---
    /**
     * @brief Adds a calculated DeltaQ to the snapshot.
     * @param timestamp The lower time bound at which the DeltaQ was calculated.
     * @param deltaQ The DeltaQ value to store.
     * @param bounds Confidence interval bounds for the DeltaQ.
     */
    void addCalculatedDeltaQ(std::uint64_t timestamp, const DeltaQ &deltaQ, const std::vector<Bound> &bounds);

    /**
     * @brief Retrieves the oldest calculated DeltaQ.
     * @return The oldest DeltaQRepr (based on timestamp order).
     */
    DeltaQ getOldestCalculatedDeltaQ() const;

    /// @brief Removes the oldest calculated DeltaQ entry (FIFO order).
    void removeOldestCalculatedDeltaQ();

    // --- QTA Methods ---
    /**
     * @brief Adds a QTA to the snapshot.
     * @param timestamp The time associated with the QTA.
     * @param qta The QTA value to store.
     */
    void addQTA(std::uint64_t timestamp, const QTA &qta);

    // --- Size Management ---
    /// @return The number of observed DeltaQs stored.
    std::size_t getObservedSize() const;

    /// @return The number of calculated DeltaQs stored.
    std::size_t getCalculatedSize() const;

    /**
     * @brief Truncates observed/calculated DeltaQs to a specified size (removes oldest entries).
     * @param size The maximum number of entries to retain.
     */
    void resizeTo(size_t size);

    // --- Name Management ---
```

```
    /**
     * @brief Sets the name of the observable.
     * @param name New name for the observable.
     */
    void setName(const std::string &name);

    /// @return The current observable name.
    std::string getName() &;

    // --- Data Retrieval ---
    /// @return A vector of all observed DeltaQReprs (sorted by timestamp).
    std::vector<DeltaQRepr> getObservedDeltaQs() const &;

    /// @return A vector of all calculated DeltaQReprs (sorted by timestamp).
    std::vector<DeltaQRepr> getCalculatedDeltaQs() const &;

    /**
     * @brief Retrieves an observed DeltaQ at a specific timestamp.
     * @param timestamp The time to query.
     * @return The DeltaQRepr if found, or `std::nullopt` otherwise.
     */
    std::optional<DeltaQRepr> getObservedDeltaQAtTime(std::uint64_t timestamp);

    /**
     * @brief Retrieves a calculated DeltaQ at a specific timestamp.
     * @param timestamp The time to query.
     * @return The DeltaQRepr if found, or `std::nullopt` otherwise.
     */
    std::optional<DeltaQRepr> getCalculatedDeltaQAtTime(std::uint64_t timestamp);

    /// @return A vector of all QTAs (sorted by timestamp).
    std::vector<QTA> getQTAs() const &;
};
```

G.4.7 TriggerManager.h

```
#pragma once

#include "TriggerTypes.h"
#include "Triggers.h"
#include <memory>
#include <optional>
#include <vector>

/**
 * @class TriggerManager
 * @brief Manages triggers for an observable
 */
class TriggerManager
{
public:
```

```cpp
    struct Trigger {
        TriggerType type;

        TriggerDefs::Condition condition;

        TriggerDefs::Action action;

        bool enabled;

        std::optional<int> sampleLimitValue;

        Trigger(TriggerType t, TriggerDefs::Condition c, TriggerDefs::
   Action a, bool e = true);
    };
    /**
     * @brief Add a trigger for an observable,
     * @param type The type of the trigger
     * @param condition The condition for the trigger to be fired
     * @param action The action to perform when fired
     * @param enabled It the trigger is enabled
     */
    void addTrigger(TriggerType type, TriggerDefs::Condition condition
   , TriggerDefs::Action action, bool enabled = true);

    /**
     *  @brief Get all triggers set for a type
     */
    std::vector<Trigger> getTriggersByType(TriggerType type);

    /**
     * @brief Evaluate a DeltaQ to see if a trigger should be fired
     * @param dq The DeltaQ to evaluate
     * @param qta The qta to compare against to
     * @param std::uint64_t Keep the time to log it if the trigger
   fires
     */
    void evaluate(const DeltaQ &dq, const QTA &qta, std::uint64_t)
   const;

    /**
     * @brief Remove triggers if their type matches the param type
     * @param type The trigger's type'
     */
    void removeTriggersByType(TriggerType type);

    /**
     * @brief Remove all triggers
     */
    void clearAllTriggers();

    /**
     * @brief Enable/Disable all triggers with a type
     * @param type The type of trigger to enable/disable
     * @param enabled
     */
    void setTriggersEnabled(TriggerType type, bool enabled);
```

```

    /**
     * @brief Get all triggers
     */
    std::vector<Trigger> getAllTriggers() const;

private:
    std::vector<Trigger> triggers_;
};
```

G.4.8 TriggerTypes.h

```
#ifndef TRIGGERTYPE_H
#define TRIGGERTYPE_H
#pragma once

enum class TriggerType {
    SampleLimit,
    QTAViolation,
    Failure,
    Hazard
};

#endif //TRIGGERTYPE_H
```

G.4.9 Triggers.h

```
#pragma once

#include "DeltaQ.h"
#include "QTA.h"
#include "TriggerTypes.h"
#include <functional>
#include <iostream>
#include <string>

namespace TriggerDefs
{
using Condition = std::function<bool(const DeltaQ &, const QTA &)>;
using Action = std::function<void(const DeltaQ &, const QTA &, std::
   uint64_t)>;

namespace Conditions
{
    Condition SampleLimit(int maxSamples);
    Condition QTABounds();
    Condition FailureRate(double threshold);
}

namespace Actions
{
    Action LogToConsole(const std::string &message);
    Action notify();
```

```
    Action SaveSnapshot(const std::string &filename);
}
}
```

G.5 parser

G.5.1 SystemBuilder.h

```
#ifndef SYSTEMBUILDER_H
#define SYSTEMBUILDER_H

#include "../diagram/System.h"
#include "DQGrammarVisitor.h"
#include <memory>
#include <unordered_map>

/**
 * @brief Visitor class that builds a System from a parsed grammar tree.
 */
class SystemBuilderVisitor : public parser::DQGrammarVisitor
{
private:
    std::unordered_map<std::string, std::shared_ptr<Outcome>> outcomes;   ///< Map of outcome names to Outcome objects.
    std::unordered_map<std::string, std::shared_ptr<Operator>> operators; ///< Map of operator names to Operator objects.
    std::unordered_map<std::string, std::shared_ptr<Probe>> probes; ///< Map of probe names to Probe objects.

    std::vector<std::string> definedProbes; ///< List of defined probe names.
    System system; ///< The final system constructed by the visitor.

    std::string currentlyBuildingProbe; ///< Tracks the probe currently being built for dependency management.
    std::map<std::string, std::vector<std::string>> dependencies; ///< Graph of probe dependencies.

    std::unordered_map<std::string, std::vector<std::string>> definitionLinks; ///< For debugging: links between definitions.
    std::unordered_map<std::string, std::vector<std::vector<std::string>>> operatorLinks; ///< For debugging: operator chains.
    std::vector<std::string> systemLinks; ///< Top-level system observable links.
    std::unordered_set<std::string> allNames; ///< Tracks all used names to detect duplicates.

    /**
     * @brief Checks the dependency graph for cycles.
     * @throws std::invalid_argument if a cycle is detected.
     */
    void checkForCycles() const;

```

```cpp
    /**
     * @brief Recursive utility to detect cycles in a graph.
     * @param node Current node.
     * @param visited Set of visited nodes.
     * @param recursionStack Stack of nodes in the current DFS path.
     * @return True if a cycle is found.
     */
    bool hasCycle(const std::string& node,
                  std::set<std::string>& visited,
                  std::set<std::string>& recursionStack) const;

public:
    /**
     * @brief Returns the constructed System.
     * @return The system object.
     */
    System getSystem() const;

    // Visitor overrides from DQGrammarVisitor:
    std::any visitStart(parser::DQGrammarParser::StartContext *context
    ) override;
    std::any visitDefinition(parser::DQGrammarParser::
    DefinitionContext *context) override;
    std::any visitSystem(parser::DQGrammarParser::SystemContext *
    context) override;
    std::any visitComponent(parser::DQGrammarParser::ComponentContext
    *context) override;
    std::any visitBehaviorComponent(parser::DQGrammarParser::
    BehaviorComponentContext *context) override;
    std::any visitProbeComponent(parser::DQGrammarParser::
    ProbeComponentContext *context) override;
    std::any visitProbability_list(parser::DQGrammarParser::
    Probability_listContext *context) override;
    std::any visitComponent_list(parser::DQGrammarParser::
    Component_listContext *context) override;
    std::any visitOutcome(parser::DQGrammarParser::OutcomeContext *
    context) override;
    std::any visitComponent_chain(parser::DQGrammarParser::
    Component_chainContext *context) override;
};

#endif // SYSTEMBUILDER_H
```

G.5.2 SystemErrorListener.h

```cpp
#pragma once
#include "antlr4-runtime.h"
#include <iostream>
#include <stdexcept>

class SystemErrorListener : public antlr4::BaseErrorListener
{
public:
    void syntaxError(antlr4::Recognizer *recognizer, antlr4::Token *
    offendingSymbol, size_t line, size_t charPositionInLine, const std
```

```
    ::string &msg,
         std::exception_ptr e) override
    {
         throw std::runtime_error("Syntax error at line " + std::
    to_string(line) + ":" + std::to_string(charPositionInLine) + " - "
     + msg);
    }
};
```

G.5.3 SystemParserInterface.h

```
#pragma once

#include "../diagram/System.h"
#include "DQGrammarLexer.h"
#include "DQGrammarParser.h"
#include "SystemBuilder.h"
#include "antlr4-runtime.h"
#include <memory>
#include <optional>
#include <string>

/**
 * @class SystemParserInterface
 * @brief Provides an interface for parsing system definitions from
    files or strings.
 */
class SystemParserInterface
{
public:
    /**
     * @brief Parses a system definition from a file.
     * @param filename Path to the file containing system definition.
     * @return Optional containing the parsed System if successful,
    nullopt otherwise.
     */
    static std::optional<System> parseFile(const std::string &filename
    );

    /**
     * @brief Parses a system definition from a string.
     * @param input String containing system definition.
     * @return Optional containing the parsed System if successful,
    nullopt otherwise.
     */
    static std::optional<System> parseString(const std::string &input)
    ;

private:
    /**
     * @brief Internal parsing method using ANTLR input stream.
     * @param input ANTLR input stream containing system definition.
     * @return Optional containing the parsed System if successful,
    nullopt otherwise.
     */
```

```
    static std::optional<System> parseInternal(antlr4::
    ANTLRInputStream &input);
};
```

G.6 server

G.6.1 Server.h

```
#ifndef SERVER_H
#define SERVER_H

#include "../diagram/System.h"
#include <atomic>
#include <condition_variable>
#include <memory>
#include <mutex>
#include <netinet/in.h>
#include <queue>
#include <sys/socket.h>
#include <thread>

/**
 * @class Server
 * @brief TCP server for handling client connections and Erlang
    communication.
 */
class Server
{
public:
    /**
     * @brief Constructs a Server instance.
     * @param port The TCP port to listen on.
     */
    Server(int port);

    /**
     * @brief Destructor cleans up sockets and threads.
     */
    ~Server();

    /**
     * @brief Sends a command to the Erlang process.
     * @param command The command string to send.
     */
    void sendToErlang(const std::string &command);

    bool startServer(const std::string &ip = "0.0.0.0", int port =
    8080);

    void stopServer();

    bool setErlangEndpoint(const std::string &ip, int port);

    bool isServerRunning() const
```

```
    {
        return server_started;
    }

    /**
     * @brief Stops the server and worker threads.
     */
    void stop();

private:
    /**
     * @brief Main server loop running in a separate thread.
     */
    void run();

    int server_fd; ///< Server socket file descriptor
    int new_socket; ///< Client socket file descriptor
    struct sockaddr_in address; ///< Server address structure
    int port; ///< Listening port number

    std::thread serverThread; ///< Thread for server operations

    /**
     * @brief Updates the system reference from Application.
     */
    void updateSystem();

    /**
     * @brief Parses messages from Erlang.
     * @param buffer The message buffer.
     * @param len Length of the message.
     */
    void parseErlangMessage(const char *buffer, int len);

    std::shared_ptr<System> system; ///< Reference to the system being
    monitored

    std::vector<std::thread> clientThreads; ///< Active client handler
    threads
    std::mutex clientsMutex; ///< Mutex for client threads access
    std::atomic<bool> running {false}; ///< Server running state flag

    /**
     * @brief Handles communication with a client.
     * @param clientSocket The client socket file descriptor.
     */
    void handleClient(int clientSocket);

    /**
     * @brief Cleans up finished client threads.
     */
    void cleanupThreads();

    int erlang_socket = -1; ///< Socket for Erlang communication
    std::mutex erlangMutex; ///< Mutex for Erlang socket operations

```

```
    /**
     * @brief Establishes connection to Erlang.
     * @return true if connection succeeded.
     */
    bool connectToErlang();

    int client_socket; ///< Current client socket

    // Asynchronous sample processing
    std::queue<std::pair<std::string, Sample>> sampleQueue; ///< Sample processing queue
    std::mutex queueMutex; ///< Mutex for queue access
    std::condition_variable queueCond; ///< Condition variable for queue notifications
    std::thread workerThread; ///< Worker thread for sample processing
    bool shutdownWorker = false; ///< Flag to signal worker thread shutdown

    std::string erlang_ip = "127.0.0.1"; ///< Erlang server IP
    int erlang_port = 8081; ///< Erlang server port
    std::string server_ip = "0.0.0.0"; ///< Current C++ server IP
    bool server_started = false;
};

#endif
```

Appendix H

Build Configuration Files

H.1 src/CMakeLists.txt

```
add_library(${PREFIX}_application
    Application.cpp
    Application.h
)


target_link_libraries(${PREFIX}_application
    PRIVATE
    ${PREFIX}_server
)

# Make sure other libraries can find Application headers
target_include_directories(${PREFIX}_application
    PUBLIC ${CMAKE_SOURCE_DIR}
)

add_subdirectory(dashboard)
add_subdirectory(diagram)
add_subdirectory(maths)
add_subdirectory(server)
add_subdirectory(parser)

add_executable(RealTimeDeltaQSD main.cpp)

target_include_directories(RealTimeDeltaQSD PUBLIC ${CMAKE_SOURCE_DIR})

target_link_libraries(RealTimeDeltaQSD ${PREFIX}_server ${PREFIX}_diagram ${PREFIX}_dashboard ${PREFIX}_parser)
```

H.2 dashboard/CMakeLists.txt

```
add_library(${PREFIX}_dashboard
    ColorRegistry.cpp
```

```
    ColorRegistry.h
    CustomLegendEntry.h
    CustomLegendEntry.cpp
    CustomLegendPanel.h
    CustomLegendPanel.cpp
    DelaySettingsWidget.h
    DelaySettingsWidget.cpp
    DQPlotController.h
    DQPlotController.cpp
    DQPlotList.h
    DQPlotList.cpp
    DeltaQPlot.h
    DeltaQPlot.cpp
    MainWindow.h
    MainWindow.cpp
    NewPlotList.h
    NewPlotList.cpp
    ObservableSettings.h
    ObservableSettings.cpp
    SamplingRateWidget.h
    SamplingRateWidget.cpp
    Sidebar.h
    Sidebar.cpp
    SnapshotViewerWindow.h
    SnapshotViewerWindow.cpp
    StubControlWidget.h
    StubControlWidget.cpp
    SystemCreationWidget.h
    SystemCreationWidget.cpp
    QTAInputWidget.cpp
    QTAInputWidget.h
    TriggersTab.cpp
    TriggersTab.h
)


find_package(QT NAMES Qt6 Qt5 REQUIRED COMPONENTS Core Gyu Widgets
    Charts Graphs)
find_package(Qt6 REQUIRED COMPONENTS Core Gui Charts Widgets Graphs)

target_link_libraries(${PREFIX}_dashboard
        PUBLIC Qt6::Core Qt6::Gui Qt6::Widgets Qt6::Charts Qt6::Graphs
        ${PREFIX}_parser
    )

target_include_directories(${PREFIX}_dashboard
        PUBLIC
        ${CMAKE_SOURCE_DIR}
        ${PREFIX}_parser
)
```

H.3 diagram/CMakeLists.txt

```
add_library(${PREFIX}_diagram
```

```
    Outcome.h
    Outcome.cpp
    Operator.h
    Operator.cpp
    Probe.h
    Probe.cpp
    System.h
    System.cpp
    Sample.h
    Observable.h
    Observable.cpp
)

target_link_libraries(${PREFIX}_diagram
    PUBLIC
        ${PREFIX}_maths
        ${PREFIX}_application
)


target_include_directories(${PREFIX}_diagram
        PUBLIC
        ${CMAKE_SOURCE_DIR}
)

add_executable(diagram
        main.cpp
)

target_link_libraries(diagram
        PRIVATE
        ${PREFIX}_diagram
)
```

H.4 maths/CMakeLists.txt

```
add_library(${PREFIX}_maths
    ConfidenceInterval.h
    ConfidenceInterval.cpp
    DeltaQ.h
    DeltaQ.cpp
    DeltaQRepr.h
    DeltaQOperations.h
    DeltaQOperations.cpp
    QTA.h
    Triggers.h
    Triggers.cpp
    TriggerTypes.h
    TriggerManager.cpp
    TriggerManager.h
    Snapshot.h
    Snapshot.cpp
)

```

```
target_include_directories(${PREFIX}_maths
        PUBLIC
        ${CMAKE_SOURCE_DIR}
)

target_link_libraries(${PREFIX}_maths
    PRIVATE
    fftw3
)

add_executable(maths
        main.cpp
)


target_link_libraries(maths
        PRIVATE
        ${PREFIX}_maths
)
```

H.5 parser/CMakeLists.txt

```

find_package(antlr4-runtime REQUIRED CONFIG)

if(NOT antlr4-runtime_FOUND)
    # Manually specify paths (adjust according to your installation)
    set(ANTLR4_INCLUDE_DIR "/usr/local/include")
    set(ANTLR4_LIB_DIR "/usr/local/lib")

    find_library(ANTLR4_RUNTIME_LIB antlr4-runtime PATHS ${
   ANTLR4_LIB_DIR})
    if(NOT ANTLR4_RUNTIME_LIB)
        message(FATAL_ERROR "ANTLR4 runtime library not found")
    endif()

    add_library(antlr4-runtime SHARED IMPORTED)
    set_target_properties(antlr4-runtime PROPERTIES
        IMPORTED_LOCATION ${ANTLR4_RUNTIME_LIB}
        INTERFACE_INCLUDE_DIRECTORIES ${ANTLR4_INCLUDE_DIR}
    )
endif()

set(ANTLR_GENERATED_DIR ${CMAKE_CURRENT_BINARY_DIR}/generated)
file(MAKE_DIRECTORY ${ANTLR_GENERATED_DIR})

# Generate ANTLR files
find_program(ANTLR_EXECUTABLE antlr4)

execute_process(
    COMMAND ${ANTLR_EXECUTABLE}
        -Dlanguage=Cpp
        -visitor -no-listener
        -o ${ANTLR_GENERATED_DIR}
```

```
        -package parser
        ${CMAKE_CURRENT_SOURCE_DIR}/DQGrammar.g4
    WORKING_DIRECTORY ${CMAKE_CURRENT_SOURCE_DIR}
    OUTPUT_VARIABLE ANTLR_OUTPUT
    ERROR_VARIABLE ANTLR_ERROR
    RESULT_VARIABLE ANTLR_RESULT
)
if(NOT ANTLR_RESULT EQUAL 0)
    message(FATAL_ERROR "ANTLR generation failed: ${ANTLR_ERROR}")
endif()

message(STATUS "ANTLR generated: ${ANTLR_OUTPUT}")

add_library(${PREFIX}_parser
    ${ANTLR_GENERATED_DIR}/DQGrammarLexer.h
    ${ANTLR_GENERATED_DIR}/DQGrammarLexer.cpp
    ${ANTLR_GENERATED_DIR}/DQGrammarParser.h
    ${ANTLR_GENERATED_DIR}/DQGrammarParser.cpp
    ${ANTLR_GENERATED_DIR}/DQGrammarBaseVisitor.cpp
    ${ANTLR_GENERATED_DIR}/DQGrammarVisitor.cpp
    ${ANTLR_GENERATED_DIR}/DQGrammarBaseVisitor.h
    ${ANTLR_GENERATED_DIR}/DQGrammarVisitor.h
     SystemBuilder.h
     SystemBuilder.cpp
     SystemErrorListener.h
     SystemParserInterface.h
     SystemParserInterface.cpp
)
target_link_libraries(${PREFIX}_parser
    PUBLIC antlr4-runtime
)

target_include_directories(${PREFIX}_parser
    PUBLIC
    ${CMAKE_SOURCE_DIR}
    ${ANTLR_GENERATED_DIR}
    ${ANTLR4_INCLUDE_DIR}
)

add_executable(parser
    main.cpp
)

target_link_libraries(parser
    PRIVATE
    ${PREFIX}_parser
    antlr4-runtime
 ${PREFIX}_diagram
)
```

H.6 server/CMakeLists.txt

```
add_library(${PREFIX}_server
    Server.cpp
```

```
    Server.h
)

target_link_libraries(${PREFIX}_server
    PUBLIC
        ${PREFIX}_diagram
    )

target_include_directories(${PREFIX}_server
    PUBLIC
    ${CMAKE_SOURCE_DIR}
)
```

Appendix I

Erlang Source Files

I.1 Root

I.1.1 dqsd_otel.erl

The ΔQ adapter, it can start, fail, end spans and start and end with_spans, communicates to the TCP client to send outcome instances to the oscilloscope.

```erlang

-module(dqsd_otel).
-behaviour(application).
-author("Francesco Nieri").

-export([start/0, start_span/1, start_span/2,  end_span/2, fail_span
    /1, with_span/2, with_span/3, span_process/3]).
-export([start/2, stop/1]).
-export([init_ets/0]).
-export([set_stub_running/1]).
-export([handle_c_message/1]).

-include_lib("opentelemetry_api/include/otel_tracer.hrl").


%% @moduledoc
%% `dqsd_otel` is an Erlang module built on top of OpenTelemetry
%% to pair with the DeltaQ oscilloscope. It tracks spans start, end,
    uses a custom timeout defined by the user in the oscilloscope,
%%
%% Features:
%% - Span lifecycle management (start/end/fail/timeout)
%% - Dynamic timeouts for spans
%% - Supports toggling stub behavior at runtime
%%
%% Usage:
%%   {Ctx, Pid} = dqsd_otel:start_span(<<"my_span">>).
%%   dqsd_otel:end_span(Ctx, Pid).
%%   dqsd_otel:fail_span(Pid)


```

```erlang
%%%=======================
%%% Application Callbacks
%%%=======================


start(_Type, _Args) ->
    dqsd_otel_sup:start_link().

stop(_State) ->
    ok.

init_ets() ->
    ets:new(timeout_registry, [named_table, public, set]),
    ets:new(otel_state, [named_table, public, set]),
    ets:insert(otel_state, {stub_running, false}),
    {ok, self()}.


%%%=======================
%%% For testing purposes
%%% ======================

set_stub_running(Bool) when is_boolean(Bool) ->
    ets:insert(otel_state, {stub_running, Bool}),
    io:format("Stub running set to: ~p~n", [Bool]),
    ok.

%%%=======================
%%% Public API
%%%=======================
%% @doc Starts the otel_wrapper application and all dependencies.
-spec start() -> {ok, [atom()]} | {error, term()}.
start() ->
    application:ensure_all_started(dqsd_otel).

%% @doc Starts a span with the given name, if the stub is running.
%% Returns a tuple of SpanContext and the internal span process PID or `ignore`.
-spec start_span(binary()) -> {opentelemetry:span_ctx(), pid() | ignore}.
start_span(Name) ->
    SpanCtx = ?start_span(Name),
    case ets:lookup(otel_state, stub_running) of
        [{_, true}] ->
            case ets:lookup(timeout_registry, Name) of
                [{_, T}] ->
                    StartTime = erlang:system_time(nanosecond),
                    Pid = spawn(?MODULE, span_process, [Name, StartTime, T]),
                    {SpanCtx, Pid};
                [] ->
                    {SpanCtx, ignore}
            end;
        _ ->
            {SpanCtx, ignore}
    end.
```

```erlang

%% @doc Starts a span with attributes.
-spec start_span(binary(), map()) -> {opentelemetry:span_ctx(), pid() | ignore}.
start_span(Name, Attrs) when is_map(Attrs) ->
    SpanCtx = ?start_span(Name, Attrs),
    case ets:lookup(otel_state, stub_running) of
        [{_, true}] ->
            case ets:lookup(timeout_registry, Name) of
                [{_, T}] ->
                    StartTime = erlang:system_time(nanosecond),
                    Pid = spawn(?MODULE, span_process, [Name, StartTime, T]),
                    {SpanCtx, Pid};
                [] ->
                    {SpanCtx, ignore}
            end;
        _ ->
            {SpanCtx, ignore}
    end.

%% @doc Ends the span and reports it, unless stub is disabled or Pid is `ignore`.
-spec end_span(opentelemetry:span_ctx(), pid() | ignore) -> ok | term().
end_span(Ctx, Pid) ->
    ?end_span(Ctx),
        case Pid of
            ignore -> ok;
        _ when is_pid(Pid) ->
            Pid ! {<<"end_span">>, erlang:system_time(nanosecond)}
    end.


%% @doc Fail the span and reports it to the oscilloscope, unless stub is disabled or Pid is `ignore`.
-spec fail_span( pid() | ignore) -> ok | term().
fail_span(Pid) ->
    case Pid of
        ignore -> ok;
    _ when is_pid(Pid) ->
        Pid ! {<<"fail_span">>, erlang:system_time(nanosecond)}
    end.



%% @doc Executes Fun inside a span with attributes.
-spec with_span(binary(), fun(() -> any())) -> any().
with_span(Name, Fun) ->
    ?with_span(Name, #{},
        fun(_SpanCtx) ->
            Pid = start_with_span(Name),
            Result = Fun(),
            end_with_span(Pid),
            Result
        end).
```

```erlang

%% @doc Executes Fun inside a span with attributes.
-spec with_span(binary(), fun(() -> any()), map()) -> any().
with_span(Name, Fun, Attrs) when is_map(Attrs), is_function(Fun, 0) ->
    ?with_span(Name, Attrs,
        fun(_SpanCtx) ->
            Pid = start_with_span(Name),
            Result = Fun(),
            end_with_span(Pid),
            Result
        end).

start_with_span(Name) ->
    case ets:lookup(otel_state, stub_running) of
        [{_, true}] ->
            case ets:lookup(timeout_registry, Name) of
                [{_, T}] ->
                    StartTime = erlang:system_time(nanosecond),
                    Pid = spawn(?MODULE, span_process, [Name,
   StartTime, T]),
                    Pid;
                [] ->
                    ignore
            end;
        _ ->
            ignore
    end.

end_with_span(Pid) ->
        case Pid of
            ignore -> ok;
        _ when is_pid(Pid) ->
            Pid ! {<<"end_span">>, erlang:system_time(nanosecond)}
    end.



%%%======================
%%% Span Worker
%%%======================

span_process(NameBin, StartTime, Timeout) ->
    Deadline = StartTime + (Timeout * 1000000),
    Timer = erlang:send_after(Timeout, self(), {<<"timeout">>,
   Deadline}),
    receive
        {<<"fail_span">>, EndTime} ->
            io:format("failure"),
            erlang:cancel_timer(Timer),
            send_span(NameBin, StartTime, EndTime, <<"fa">>);
        {<<"end_span">>, EndTime} ->
            erlang:cancel_timer(Timer),
            send_span(NameBin, StartTime, EndTime, <<"ok">>);
        {<<"timeout">>, Deadline} ->
            send_span(NameBin, StartTime, Deadline, <<"to">>)
    end.
```

```


%%=======================
%%% Handle Incoming Messages from C
%%=======================

handle_c_message(Bin) when is_binary(Bin) ->
    case binary:split(Bin, <<";">>, [global]) of
        [<<"set_timeout">>, Name, TimeoutBin] ->
            case string:to_integer(binary_to_list(TimeoutBin)) of
                {Timeout, _} when is_integer(Timeout) ->
                    ets:insert(timeout_registry, {Name, Timeout}),
                    io:format("dqsd_otel: Timeout set: ~p = ~p~n", [Name, Timeout]);
                _ ->
                    io:format("dqsd_otel: Invalid timeout: ~p~n", [TimeoutBin])
            end;
        [<<"start_stub">>] ->
            ets:insert(otel_state, {stub_running, true}),
            io:format("dqsd_otel: Stub enabled~n");
        [<<"stop_stub">>] ->
            ets:insert(otel_state, {stub_running, false}),
            io:format("dqsd_otel: Stub stopped~n");
        _ ->
            io:format("dqsd_otel: Unknown command: ~p~n", [Bin])
    end.

%%=======================
%%% Sending Span Data to C
%%=======================

send_span(NameBin, Start, End, StatusBin) ->
    Data = io_lib:format("n:~s;b:~p;e:~p;s:~s~n", [
        NameBin,
        Start,
        End,
        StatusBin
    ]),
    dqsd_otel_tcp_client:send_span(lists:flatten(Data)).
```

I.1.2 dqsd_otel_app.erl

```
%%%-------------------------------------------------------------------
%% @doc dqsd_otel public API
%% @end
%%%-------------------------------------------------------------------

-module(dqsd_otel_app).

-behaviour(application).

-export([start/2, stop/1]).

start(_StartType, _StartArgs) ->
```

```
    dqsd_otel_sup:start_link().

stop(_State) ->
    ok.
```

I.1.3 dqsd_otel_sup.erl

The supervisor of the adapter. It start the TCP server, client and adapter.

```

-module(dqsd_otel_sup).
-behaviour(supervisor).
-export([start_link/0, init/1]).

start_link() ->
    supervisor:start_link({local, ?MODULE}, ?MODULE, []).

init([]) ->
    ChildSpecs = [
        #{id => ets_init,
          start => {dqsd_otel, init_ets, []},
          restart => temporary,
          shutdown => brutal_kill,
          type => worker,
          modules => [dqsd_otel]},

        #{id => tcp_server,
          start => {dqsd_otel_tcp_server, start_link, []},
          restart => permanent,
          shutdown => brutal_kill,
          type => worker,
          modules => [dqsd_otel_tcp_server]},

        #{id => tcp_client,
          start => {dqsd_otel_tcp_client, start_link, []},
          restart => permanent,
          shutdown => brutal_kill,
          type => worker,
          modules => [dqsd_otel_tcp_client]}
    ],
    {ok, {{one_for_one, 5, 10}, ChildSpecs}}.
```

I.1.4 dqsd_otel_tcp_client.erl

The TCP client, it can send outcome instances to the oscilloscope.

```

    -module(dqsd_otel_tcp_client).
    -behaviour(gen_server).

    -export([start_link/0, send_span/1]).
    -export([init/1, handle_call/3, handle_cast/2, handle_info/2,
  terminate/2]).
    -export([disconnect/0, try_connect/2]).
```

```erlang

-define(SERVER, ?MODULE).

-record(state, {
    socket = undefined,
    logged_disconnected = false
}).

%%% Public API

start_link() ->
    gen_server:start_link({local, ?SERVER}, ?MODULE, [], []).

send_span(Data) ->
    gen_server:cast(?SERVER, {send, Data}).

%% @doc Connect to a custom IP and Port. Replaces any existing connection.
-spec try_connect(string() | binary(), integer()) -> ok.
try_connect(IP, Port) ->
    gen_server:cast(?SERVER, {try_connect, IP, Port}).

-spec disconnect() -> ok.
disconnect() ->
    gen_server:cast(?SERVER, disconnect).

%%% gen_server callbacks

init([]) ->
    State = #state{socket = undefined, logged_disconnected = false},
    {ok, State}.

handle_cast({send, _Data}, State = #state{socket = undefined, logged_disconnected = false}) ->
    io:format("dqsd_otel: No socket. Dropping subsequent spans.~n"),
    {noreply, State#state{logged_disconnected = true}};

handle_cast({send, _Data}, State = #state{socket = undefined}) ->
    %% Already logged, suppress further logs
    {noreply, State};

handle_cast({send, Data}, State = #state{socket = Socket}) ->
    case gen_tcp:send(Socket, Data) of
        ok ->
            {noreply, State};
        {error, Reason} ->
            io:format("dqsd_otel: TCP send failed: ~p~n", [Reason]),
            NewState = State#state{socket = undefined, logged_disconnected = true},
            {noreply, NewState}
    end;

handle_cast({try_connect, IP, Port}, State) ->
```

```erlang
    IPStr = case IP of
        Bin when is_binary(Bin) -> binary_to_list(Bin);
        Str when is_list(Str)   -> Str
    end,
    case gen_tcp:connect(IPStr, Port, [binary, {active, false}]) of
        {ok, Socket} ->
            io:format("dqsd_otel: Adapter connected to ~s:~p~n", [IPStr, Port]),

            case State#state.socket of
                undefined -> ok;
                OldSocket -> catch gen_tcp:close(OldSocket)
            end,
            {noreply, State#state{socket = Socket, logged_disconnected = false}};
        {error, Reason} ->
            io:format("dqsd_otel: Connection to ~s:~p failed: ~p~n", [IPStr, Port, Reason]),
            {noreply, State}
    end;

handle_cast(disconnect, State = #state{socket = undefined}) ->
    io:format("dqsd_otel: Socket already disconnected.~n"),
    {noreply, State};

handle_cast(disconnect, State = #state{socket = Socket}) ->
    io:format("dqsd_otel: Disconnecting TCP socket.~n"),
    gen_tcp:close(Socket),
    {noreply, State#state{socket = undefined, logged_disconnected = false}}.

handle_info(_, State) ->
    {noreply, State}.

handle_call(_, _From, State) ->
    {reply, ok, State}.

terminate(_Reason, State) when is_record(State, state) ->
    case State#state.socket of
        undefined -> ok;
        Socket -> gen_tcp:close(Socket)
    end;
terminate(_Reason, _Other) ->
    ok.
```

I.1.5 dqsd_otel_tcp_server.erl

The TCP server accepts messages from the oscilloscope and forwards them to the adapter to set the various settings.

```erlang
%%-------------------------------------------------------------------
%% @doc
%% TCP server for receiving and processing messages from an oscilloscope.
```

```erlang
%%% Allows starting and stopping a TCP server on a given IP and port.
%%%
%%% When a line-delimited binary message is received, it is passed to
%%% `dqsd_otel:handle_c_message/1` for further processing.
%%%
%%%----------------------------------------------------------------------

-module(dqsd_otel_tcp_server).
-behaviour(gen_server).

%%% API
-export([start_link/0, start_server/2, stop_server/0]).

%%% gen_server callbacks
-export([init/1, handle_call/3, handle_cast/2, handle_info/2]).

-record(state, {
    socket,     %% Listening socket
    acceptor    %% PID of acceptor process
}).

-spec start_link() -> {ok, pid()} | ignore | {error, term()}.
start_link() ->
    gen_server:start_link({local, ?MODULE}, ?MODULE, [], []).

%%-----------------------------------------------------------------------
%% @doc Starts the TCP listener on the given IP and Port.
%% Spawns an accept loop to handle incoming connections.
%%
%% IP can be a string, binary, or tuple (e.g., "127.0.0.1" or {127,0,0,1}).
%% Port is an integer.
%%-----------------------------------------------------------------------

-spec start_server(string() | binary() | tuple(), integer()) -> ok | {error, term()}.
start_server(IP, Port) ->
    gen_server:call(?MODULE, {start, IP, Port}).

%%-----------------------------------------------------------------------
%% @doc Stops the TCP server and closes the listening socket.
%% Also shuts down the acceptor process.
%%
%% @spec stop_server() -> ok | {error, not_running}
%%-----------------------------------------------------------------------
stop_server() ->
    gen_server:call(?MODULE, stop).

%%-----------------------------------------------------------------------
%% @private
%% gen_server init callback.
%% Initializes the state without an open socket or acceptor.
%%-----------------------------------------------------------------------
init([]) ->
    {ok, #state{}}.

```

```
%%--------------------------------------------------------------------
%% @private
%% Handles the start request. Binds to the given IP and port.
%% On success, starts the accept loop.
%%--------------------------------------------------------------------
handle_call({start, IP, Port}, _From, State) ->
    Options = [binary, {packet, line}, {active, false}, {reuseaddr, true}, {ip, parse_ip(IP)}],
    case gen_tcp:listen(Port, Options) of
        {ok, Socket} ->
            Acceptor = spawn(fun() -> accept_loop(Socket) end),
            io:format("dqsd_otel: Listening socket started on ~p:~p~n", [IP, Port]),
            {reply, ok, State#state{socket = Socket, acceptor = Acceptor}};
        {error, Reason} ->
            io:format("dqsd_otel: Could not start listening socket on ~p:~p~n", [IP, Port]),
            {reply, {error, Reason}, State}
    end;

%%--------------------------------------------------------------------
%% @private
%% Handles stop request. Stops acceptor and closes socket.
%% Returns error if server is not running.
%%--------------------------------------------------------------------
handle_call(stop, _From, #state{socket = undefined, acceptor = undefined} = State) ->
    io:format("dqsd_otel: Server not running.~n"),
    {reply, {error, not_running}, State};


handle_call(stop, _From, #state{socket = Socket, acceptor = Acceptor}) ->
    catch exit(Acceptor, shutdown),
    gen_tcp:close(Socket),
    io:format("dqsd_otel: Stopped listening from oscilloscope~n"),
    {reply, ok, #state{socket = undefined, acceptor = undefined}}.

handle_cast(_, State) -> {noreply, State}.
handle_info(_, State) -> {noreply, State}.

%%--------------------------------------------------------------------
%% @private
%% Accept loop that runs in a separate process.
%% Accepts TCP connections and spawns a handler for each one.
%%--------------------------------------------------------------------
accept_loop(ListenSocket) ->
    {ok, Socket} = gen_tcp:accept(ListenSocket),
    spawn(fun() -> handle_client(Socket) end),
    accept_loop(ListenSocket).

%%--------------------------------------------------------------------
%% @private
%% Handles a client socket connection. Reads line-by-line.
%% Forwards trimmed binary lines to `dqsd_otel:handle_c_message/1`.
```

```
%%----------------------------------------------------------------------
handle_client(Socket) ->
    case gen_tcp:recv(Socket, 0) of
        {ok, Line} ->
            Trimmed = binary:replace(Line, <<"\n">>, <<>>, [global]),
            dqsd_otel:handle_c_message(Trimmed),
            handle_client(Socket);
        {error, closed} ->
            gen_tcp:close(Socket)
    end.

%%----------------------------------------------------------------------
%% @private
%% Parses an IP address from string, binary, or already-parsed tuple.
%%----------------------------------------------------------------------
parse_ip("127.0.0.1") -> {127,0,0,1};
parse_ip("0.0.0.0") -> {0,0,0,0};
parse_ip(IP) when is_list(IP) ->
    {ok, Addr} = inet:parse_address(IP), Addr;
parse_ip(IP) -> IP.
```

Appendix J

Erlang Application Files

J.1 Root

J.1.1 dqsd_otel.app.src

The app.src file of the adapter.

```
{application, dqsd_otel,
    [{description, "An OpenTelemetry Wrapper for the ΔQ Oscilloscope"},
     {vsn, "git"},
     {modules, [dqsd_otel, dqsd_otel_sup, dqsd_otel_tcp_client, dqsd_otel_tcp_server]},
     {registered, [dqsd_otel_sup]},
     {applications, [kernel, stdlib, opentelemetry_exporter, opentelemetry, opentelemetry_api]},
     {mod, {dqsd_otel, []}},
     {env, []},
     {mantainers,["Francesco Nieri"]},
     {licenses,["Apache 2.0"]},
     {links,[{"https://github.com/fnieri/dqsd_otel"}]}]}.
```

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**