École polytechnique de Louvain

The △Q Oscilloscope: Real-Time Observation of Large Erlang Applications using △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. What happens when there is an overload? How can we feel something is wrong with the system early enough?

This thesis aims to provide further proof about how the ΔQSD paradigm can be used to study the behaviour of running systems and to explore tradeoffs in system design, thanks to the implementation of the Δ**Q oscilloscope**, a real time graphical dashboard that gives insights into a running Erlang system. Furthermore, the development of an Erlang application (ΔQ adapter), named `dqsd_otel`, allows the running system to communicate with the oscilloscope to receive real time insights about the execution of the former.

The oscilloscope performs statistical computations on the time series data it receives and displays the results in real time, thanks to the ΔQSD paradigm. We provide a set of triggers which are set 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. Furthermore, the implementation of efficient algorithms allows for the computations to be done rapidly on precise representations of components.

We first provide an extensive summary of ΔQSD concepts, which have been extended to allow the instrumentation of Erlang systems. 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

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

Chapter 1

Introduction

1.1 Context

ΔQSD is an industrial-strength approach for large-scale system design that can predict performance and feasibility early on in the design process. Developed over 30 years by a small group of people around Predictable Network Solutions Ltd, the paradigm has been applied in various industrial-scale problems with huge success and large savings in costs [1]. Moreover, it is the basis of Broadband forum's TR452 standard series, used in instrumenting data networks [2].

ΔQSD has important properties which make its application to distributed projects interesting, it supports:

- A compositional approach that considers performance and failure as first-class citizens.

- Stochastic approach to capture uncertainty throughout the design approach.

- Performance and feasibility can be predicted at high system load for partially defined systems [1].

Modern software development practices successfully fail to adequately consider essential quality requirements or even to consider properly whether a system can actually meet its intended outcomes, particularly when deployed at scale, the ΔQSD paradigm addresses this problem! [3]

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 Δ**Q oscilloscope** comes in.

1.2 Objective

This project will develop a practical tool, the Δ**Q 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 [4]. The tool will provide useful information for these applications both for understanding their behaviour, for diagnosing performance issues, and for optimising performance over their lifetime. [5]

The ΔQ Oscilloscope will perform statistical computations to show real time graphs about the performance of system components. With the oscilloscope prototype we will present in this paper, 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 oscilloscope targets large distributed applications handling many independent tasks where performance and reliability are important. [1]

1.3   Previous work

The ΔQSD paradigm has been formalised across different papers [3] [6] and was brought to the attention of engineers via tutorials [1] and to students at Université Catholique de Louvain. [7]

A Jupyter notebook workbench has been made available on GitHub [8], it shows real time ΔQ graphs for typical outcome diagrams but is not adequate to be scaled to real time systems, it is meant as an interactive tool to show how the ΔQSD paradigm can be applied to real life examples.

Observability tools such as Erlang tracing [9] and OpenTelemetry [10] 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.

1.4   Contributions

There are a few contributions that make the master thesis and thus, the oscilloscope, possible:

- A graphical interface to display probes' ΔQs.

- An Erlang OpenTelemetry adapter to give OpenTelemetry spans a notion of failure, to translate them to outcome instances and to communicate with the oscilloscope.

- The implementation of ΔQSD concepts from theory to practice, including a textual syntax to create outcome diagrams, derived from the original algebraic syntax.

- An efficient Fast Fourier Transform convolution algorithm based on the FFTW3 library.

- A system of triggers to catch rare events when system behaviour fails to meet quality requirements. The triggers being fired create a snapshot of the system, which gives users insights about their system's behaviour.

- Synthetic applications to test the effectiveness of ΔQSD on diagnosing running systems.

These contributions can show that the ΔQSD paradigm can be translated to real-time observation of running system. Furthermore, it proves that the paradigm has its practical applications and is not limited to a theoretical view of system design.

1.5 Roadmap

The following thesis will give the reader everything that is needed to use the oscilloscope and exploit it to its full potential.

We divided the thesis in multiple chapters, below is the roadmap of the content:

- The background chapter gives the reader an extensive background into 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 library we base our Erlang adapter on. Lastly, we provide what we believe are the current limitations of the observability tool and how we plan to tackle them.

- The design chapter 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 on how the different parts of our design interact together and how to correctly apply the concepts we introduced. Lastly, after having introduced the oscilloscope, we explain abstract concepts implemented in it, like triggers and sliding windows.

- We then present the oscilloscope in two different chapters, first providing "user level concepts" of how ΔQSD is used and what the user should expect visually from the dashboard. It also provides a complete explanation on how to write outcome diagrams and what the different sections on the dashboard do. Secondly, a more low level explanation, which goes into more technical details of the parts that compose the oscilloscope and the mathematical explanations of ΔQSD concepts explained in the previous chapter.

- We then provide synthetic applications which have been tested with the oscilloscope that demonstrate the usefulness of the oscilloscope in a distributed setting. We also perform evaluations of the performance of the different parts we have developed to understand the overhead that are present.

We end by providing future possibilities which can be explored to improve the application. In the appendix, we 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 complete background of the concepts key to understanding the ΔQSD paradigm.

Secondly, we provide a comprehensive background into 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 a metrics-based, quality-centric paradigm that uses formalised outcome diagrams to explore the performance consequences of design decisions. [6]

Key concepts of ΔQSD are **quality attenuation (ΔQ)** and **outcome diagrams** [1].

Outcome diagrams capture dependency and causality properties of the system. The ΔQSD paradigm derives bounds on performance expressed as probability distribution, encompassing all possible executions of the system. [6]

The following sections are a summary of multiple articles and presentations formalising the paradigm.

2.1.1 Outcome

An outcome o is a specific system behaviour that can be observed to start at some point in time and **may** be observed to complete at some later time. [2] Formally, what the system obtains by performing one of its tasks. One task corresponds to one outcome and viceversa. When an outcome is performed, it means that the task of an outcome is performed.

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.

Even though these outcomes are defined as "black boxes", they still have timeliness constraints like any other outcome. [6]

Observables Each outcome has two starting sets of events: the starting sets and the ending sets. Such sets are called the *observables*. Once an event from the starting set occurs, there is no guarantee that a corresponding event in the terminating set will occur within the duration limit (required time to complete). An observable is *done* when it occurs during the time limit. [3]

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}. [3]

Graphical Representation Outcomes are represented as circles, with the starting and terminating set of events being represented by boxes.
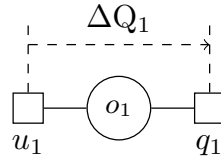
Figure 2.1: The outcome (circle) and the starting set (left) and terminating set (right) of events. [6]

2.1.2 Quality attenuation (ΔQ)

Assume a component C which receives a message m_{in} and outputs a message m_{out} after a delay d. Over multiple executions, we will have observed multiple delays which can be represented as a cumulative definition where p percent of delays have delay $\leq d$. [3]

Δ**Q** is a cumulative distribution function that defines both *latency* and *failure probability* between a start and end event. [1]

In an ideal system, an outcome would deliver a desired behaviour without error, failure, delay, but this is not the case. The quality of an outcome response "attenuated to the relative ideal" (the cumulative distribution function) is called "quality attenuation" (ΔQ) and can depend on many factors (geographical, physical . . .). Its distribution may be modelled by a random variable.

As ΔQ captures deviation from ideal behavior and incorporates delay, which is a continuous random variable, and failures/timeouts, which are discrete variables, it can be described mathematically as an *Improper Random Variable*, where the probability of a finite or bounded delay < 1.

$\Delta\mathbf{Q(x)}$ is the probability that an outcome O occurs in time $t \leq x$. The ***intangible mass*** $1 - \lim_{x\to\infty} \Delta Q(x)$ of a ΔQ will encode the probability of failure/timeout/exception occuring. [6]
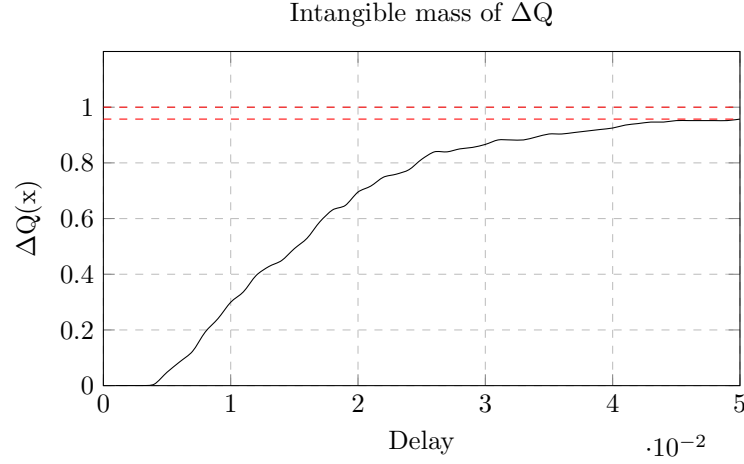
Figure 2.2: Intangible mass (red) of a ΔQ with failure rate of about 5%

2.1.3 Failure semantics

In the CDF representation of a ΔQ, there is an f percent probability that the delay is infinite, this is what failure models. Concretely, it means that an input message m_{in} **has no output message** m_{out}. [3]

Combining delay and failure in a single quantity is what makes ΔQSD a great choice to explore feasibility in system design. [1]

2.1.4 Partial ordering

A CDF of a ΔQ is *less than* the other if its CDF is everywhere to the left and above the other. Mathematically, it is a partial order.

If two ΔQs intersect, they are not ordered. [1]

2.1.5 Timeliness

Timeliness is defined as a relation between an observed ΔQ_{obs} and a required ΔQ_{req}. Timeliness is delivering results within required time bounds (sufficiently often).

A system *satisfies timeliness* if $\Delta Q_{obs} \leq \Delta Q_{req}$. [3]

2.1.6 QTA, required ΔQ

The *Quantitative Timeliness Agreement* (QTA) maps objective measurements to the subjective perception of application performance. It specifies what the base system does and its limits. [2]

Slack   There is performance *slack* when a ΔQ is strictly less than the requirement,

Hazard   There is performance *hazard* when a ΔQ is strictly greater than the requirement. [6]

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.
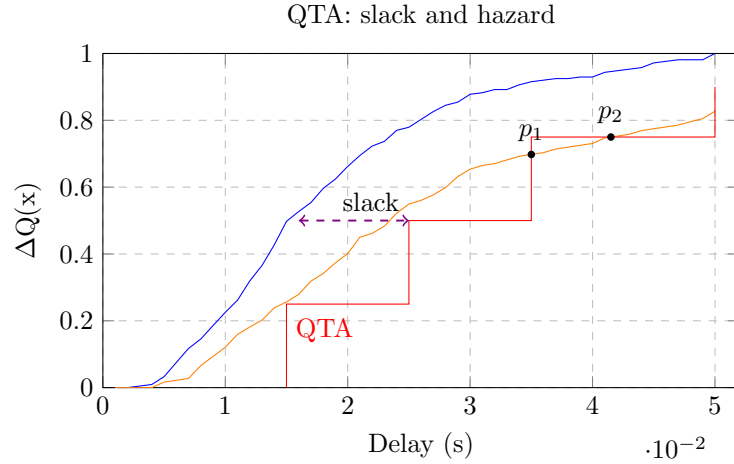
Figure 2.3: The system in blue is showing slack and satisfies the requirement. The system in orange is showing signs that it cannot handle the stress, it is not respecting the system requirements imposed by the QTA.

2.1.7   Outcome diagram

An outcome diagram is central to capture the causal relationships between the outcomes. It shows the causal connections between all the outcomes we are interested in, and it allows computing the ΔQ for the whole system [1]. It maps a system's behaviour as seen from outside to concrete outcomes [3]. There are four different operators that represent the relationships between outcomes. [1]

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, the two outcomes can be sequentially composed. The total delay ΔQ$_{AB}$ is given by the convolution of the PDFs of O_A and O_B ($O_A \circledast O_B$).
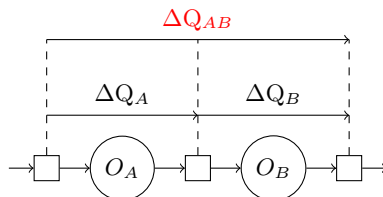
Figure 2.4: Sequential composition of O_A and O_B.

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

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

Thus, ΔQ_{AB}:

$$\Delta Q_{AB} = PDF_A \circledast PDF_B \qquad (2.2)$$

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:

$$\Delta Q_{FTF(A,B)} = Pr[d_A > t \wedge d_B > t]$$
$$= Pr[d_A > t] \cdot Pr[d_B > t] = (1 - \Delta Q_A) \cdot (1 - \Delta Q_B) \qquad (2.3)$$
$$\Delta Q_{FTF(A,B)} = \Delta Q_A + \Delta Q_B - \Delta Q_A \cdot \Delta Q_B$$
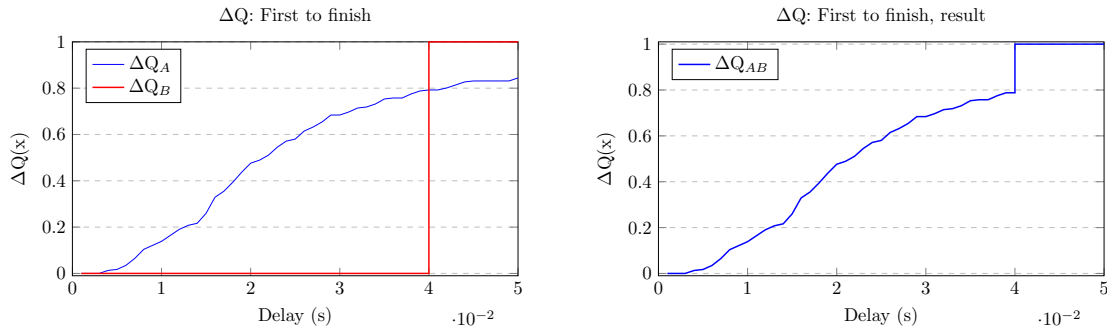
Figure 2.5: Left: $\Delta Q_{(A,B)}$. Right: $\text{FTF}_{(A,B)} = \Delta Q_{AB}$

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:

$$\Delta Q_{ATF(A,B)} = Pr[d_A \leq t \wedge d_B \leq t]$$
$$= Pr[d_A \leq t] \cdot Pr[d_B \leq t] = \Delta Q_A \cdot \Delta Q_B \qquad (2.4)$$
$$\Delta Q_{ATF(A,B)} = \Delta Q_A \cdot \Delta Q_B$$
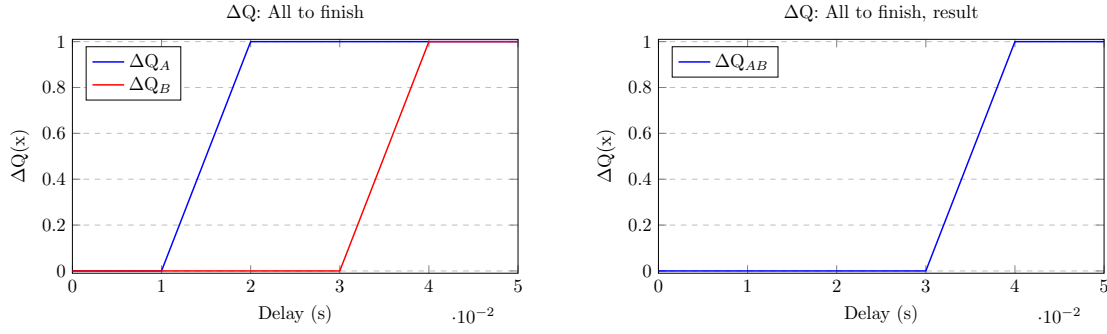
Figure 2.6: Left: $\Delta Q_{(A,B)}$. Right: $\text{ATF}_{(A,B)} = \Delta Q_{AB}$

Probabilistic choice (PC)

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

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

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

$$\Delta Q_{PC}(A, B) = \frac{p}{p+q}\Delta Q_A + \frac{q}{p+q}\Delta Q_B \tag{2.5}$$
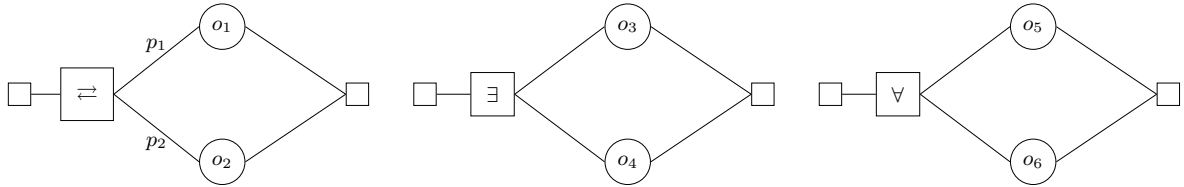
Figure 2.7: The possible operators in an outcome diagram: Probabilistic choice, first-to-finish, all-to-finish

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

These operators can be assembled together to create an outcome diagram, later on, 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.
An outcome diagram can be "unboxed" by refining the outcomes that compose it. We can adapt a situation described in Mind your Outcomes to understand how refinement can allow the user to have a very precise representation of a system. [6]

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.
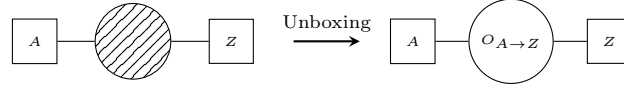
Figure 2.8: Refinement from black box to named outcome.

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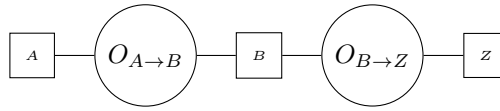
Figure 2.9: Further refinement from one task to two tasks.

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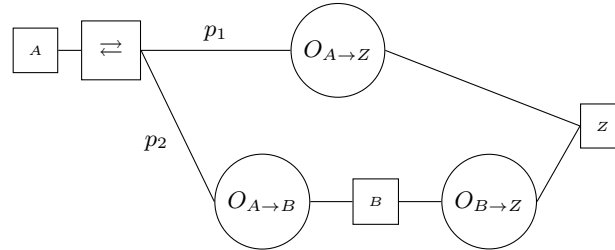
Figure 2.10: Refinement as probabilistic choice of executing either one or two tasks.

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. Let us explain the following assumption clearly.

Assume 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).
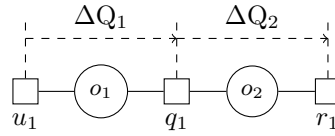
At low load, the two components behavior will be independent, the system will behave **linearly**. According to the superposition principle, the overall delay will be the sum of the two delays, as will the overall processor utilisation. [11]

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 ($o_1 \circledast o_2$).
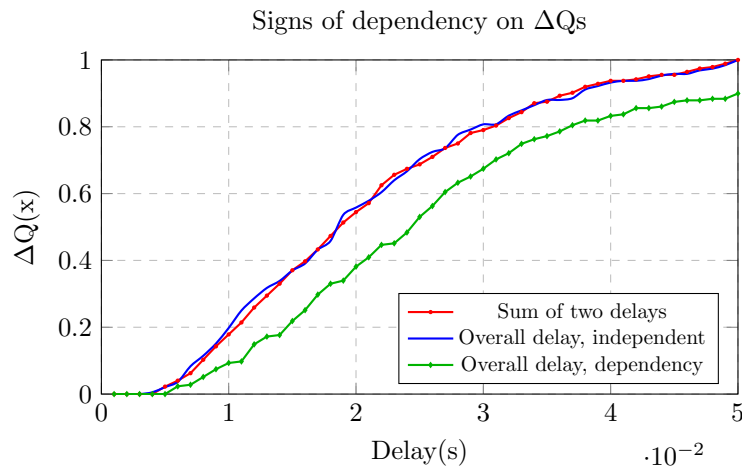
Figure 2.11: When the components are independent, the sum of the two delays (blue) and the overall delay (red) can be superposed.
When o_1 and o_2 show initial signs of dependency, the overall delay (green) can be seen deviating from the sum of the two delays.

When the system is far from being overloaded, the effect is noticeable thanks to ΔQSD. As the cliff edge of overload is approached, the nonlinearity will increase [5]. These theoretical results can be observed in practice in the oscilloscope. We will explore such cases in the synthetic applications section.

2.2 Observability

Observability refers to 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. [12]

In the case of the Erlang programming language, we explain below two tools that can be used to observe an Erlang program.

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 function `erlang:trace/3`. 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 [9]. One can observe function calls, messages being sent and received, process being spawned, garbage collecting . . . .

```
-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 2.12: erlang:trace/3 specification

Nevertheless, in Erlang Tracing 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

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. [12]

OpenTelemetry is available for a plethora of languages [13], including Erlang, although, as of writing this, only traces are available in Erlang [14].

The Erlang Ecosystem Foundation has a working group focused on evolving the tools related to observability, including OpenTelemetry and the runtime observability monitoring tools [15].

Traces

Traces are why we are basing our program on top of OpenTelemetry, traces follow the whole "path" of a request in an application, traces are comprised of one or more spans. Traces can propagate to multiple services and record multiple paths in different microservices [16].

Span   A span is a unit of work or operation. Spans can be correlated to each other and can be assembled into a trace. 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 [16]. 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 [17]. 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": "0515505510cb55c13",
  "start_time": "2022-04-29T18:52:58.114304Z",
  "end_time": "2022-04-29T22:52:58.114561Z",
  "attributes": {
    "http.route": "some_route"
  },
}
```

Figure 2.13: Example of span with a parent, indicating that child and parent are related and are both part of the same trace [17].

Monitoring OpenTelemetry spans

OpenTelemetry gives the possibility to export traces to backends and monitoring such as Jaeger, Zipkin, Datadog [18]. There, a user can monitor their workflows, analyse dependencies, troubleshoot their programs by observing the flow of the requestscitejg. These monitoring tools give extensive details about a running system, but may fail to capture essential 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 [19] might have, where you can observe cloud instances, instances cost, dependency graphs . . . but the oscilloscope can nevertheless provide precise insights about dependency, overload thanks to the ΔQSD paradigm.

This is also the reason why we work alongside OpenTelemetry, the oscilloscope can be put next to a monitoring tool where one might export spans to. An engineer might consult the main monitoring tool to get the global picture of a running app, and the oscilloscope to give more precise insights to understand the system's behaviour.
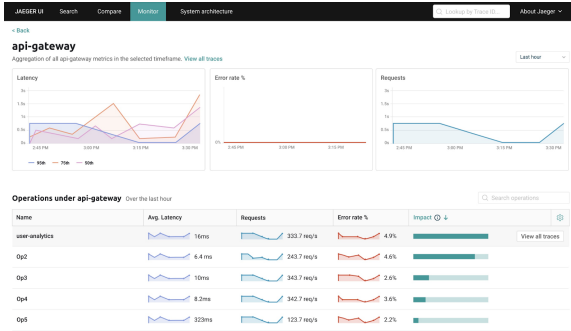
(a) Jaeger interface [20].
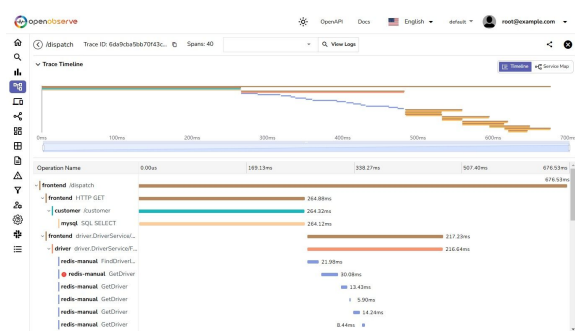
(b) A span analysis on OpenObserve [21]

Span macros

OpenTelemetry provides macros to start, end and interact with spans in Erlang, the following code excerpts are taken from the instrumentation wiki. [14]

?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. [22]

```erlang
parent_function() ->
    ?with_span(parent, #{}, fun child_function/0).


child_function() ->
    %% this is the same process, so 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() ->
                %% do work here. 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.

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

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

14

2.3   Current observability problems

A legitimate question to pose would be why one would need an additional tool to observe their system, monitoring tools are already plenty and provide useful insights into an application's behaviour. While they may seem adequate to provide a global oversight of applications, they fail to diagnose real time problems like overload, dependent behaviour early enough and in a quick manner.

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 instead of the usual 1 second. Between its start and its end, the user instrumenting the application sees nothing in their dashboard.

This is a big 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.
By leveraging ΔQSD notion of early failure and QTAs, problems can be detected right away in the oscilloscope.
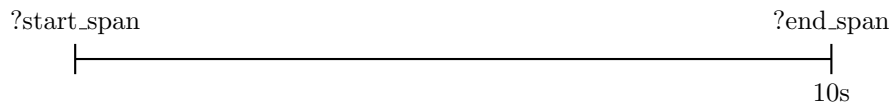
Figure 2.15: Execution of a long span in OpenTelemetry, the user will only be notified after 10 seconds that the function has ended (and taken too long).
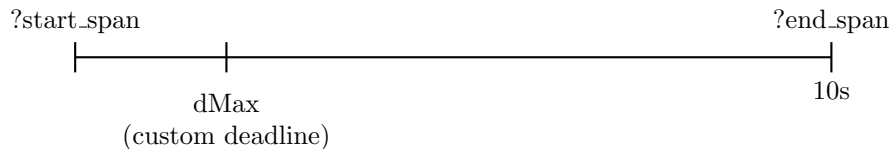
Figure 2.16: Execution of a long span in OpenTelemetry, the $dMax$ deadline allows knowing that the span has taken too long.

2.3.1   Handling of long spans

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

OpenTelemetry limits the length of its spans, moreover, those who are not terminated are lost and not exported. Why? They are the ones that tell more about a program execution. 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 program . . . lost. Moreover, a span could not be terminated for trivial reasons: refreshing a tab, network failures, crashes . . . [13]. Hazel Weakly, the author of the cited article states that there are a few hacks 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; why the need to circumvent limitations when observing a system?

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.

Chapter 3

Design

This chapter aims to first extend the concepts of ΔQSD, giving more insights into how the systems need to be instrumented to correctly work together, and how the different parts need to be integrated to interact 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 split the design of the oscilloscope in two. First explaining the Erlang side, where the system to be tested is. Secondly, we explain the C++ side. Both chapters explain how probes can be inserted and made to work together.

- Lastly, we provide high level concepts of triggers and execution windows, the key elements of the oscilloscope.

3.1 Measurement concepts

3.1.1 Probes

A system instrumented with OpenTelemetry has spans and traces to observe the execution of an operation [17]. 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.

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 [5].

Consider the figure below, a probe is attached at every component to measure their ΔQs (c_2, c_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, while p_2 and p_3 are child spans which represent single instances of execution.
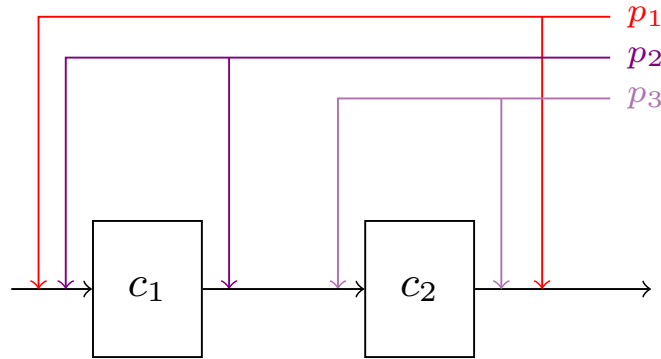
Figure 3.1: Probes inserted in a component diagram. In an applications instrumented with OpenTelemetry, p_1 could be considered the root span, c_1 and c_2 its children spans sharing a causal link.

3.1.2 Extending failure

Recall the definition of failure: *"an input message m_{in} that has no output message m_{out}"* [3]. In the previous section 2.16, we also introduced the notion of a maximum delay.

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! Moreover, 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.

The notion of failure is extended to the following definition:

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

We can leverage this new definition to observe the system and the ΔQs in real time.

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 $t_s + \text{timeout}$

- $(t_s,\mathcal{F})$: 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 (delay) from the outcome instances can be represented as a CDF, which is a ΔQ!

What can be considered a failed execution? Imagine a queue with a buffer: the buffer queue being full and dropping incoming messages can be modeled as a failure.

More generally, 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.

On another note, the way of handling "errored" spans in OpenTelemetry can differ from user to user [23], so the adapter will not handle ending and setting statuses for "failed" spans.

In any case, by the new definition of failure, **timed out and failed will both be considered as a failure** in the calculation of 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

Before delving deeper into the parts, we present the global system design diagram. We recognise two separate parts.

- The application side, where the Erlang system under test is. Consequently, it's where the ΔQ adapter will be, which performs the translation of spans to outcome instances.

- The oscilloscope side, where the ΔQ oscilloscope receives information from the adapter attached to the system under test to display graphs, define outcome diagrams and set parameters for probes.
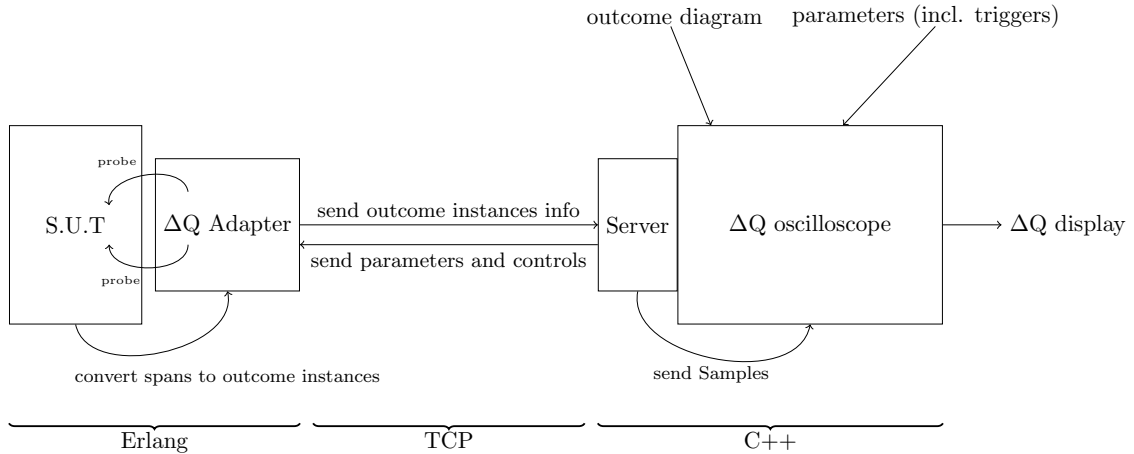
Figure 3.2: Global system design diagram.

3.2.1   System under test

The system under test **(S.U.T)** is the Erlang system the engineer wishes to observe, 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 [1].

3.2.2   ΔQ Adapter

The ΔQ adapter is the `dqsd_otel` Erlang application [24], it starts and ends Open-Telemetry 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, given 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.

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. The plethora of attributes they have is nevertheless too much for the oscilloscope. [16]

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"` is the name of the probe.

```
% 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 the following chapters.

3.3   Oscilloscope side

3.3.1   Server

The server 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 [25]. 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 allow 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 outcome, 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 to the probes' time series.

In the system below, which is equal to the one defined above, 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.
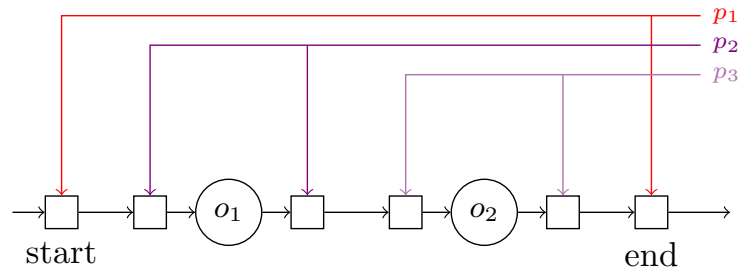
Figure 3.4: Probes inserted in the outcome diagram of the previous component diagram in Fig. 3.1.

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. 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. Furthermore, this difference is remarked in the definition of outcome diagrams, for which we provide a syntax in the following chapter.

As for operators, probes are automatically attached to the components inside them and to the start event and end events of the operators (its observables).
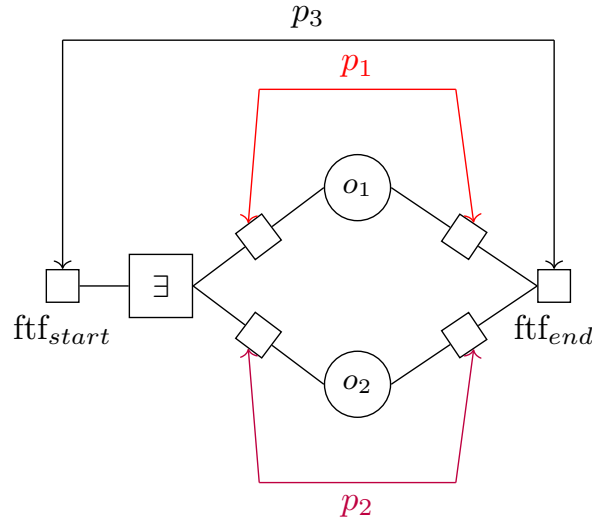
Figure 3.5: Probes inserted into an operator.

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 Triggers

Much like an oscilloscope that has a trigger mechanism to capture periodic signals or investigate a transient event [26], 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 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.

3.4.1 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. All the ΔQs which are calculated during the system's execution are stored away. Then, if no trigger is fired, older ΔQs are removed. Otherwise, the oscilloscope keeps recording ΔQs without removing older ones, to allow the user to look at the state of the system before and after the trigger.

3.5 Sliding execution windows

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

3.5.1 Sampling window

Suppose we are at time t, the observed (and calculated, if applicable) ΔQs at time t we will display are the ΔQs obtained from the outcome instances who ended within a sampling window in the **window of time** $(t-1)_l$ - $(t-1)_u$, with $t-1$ equal to $t-x$, and x the sampling rate. The sampling rate is how often ΔQs are calculated.
This is to account for various overheads that need to be taken into consideration. They could be network overhead, the adapter overhead, C++ latency . . . 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.
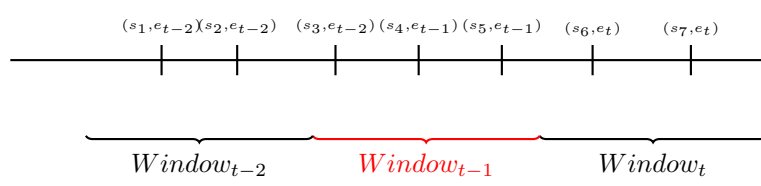
The sampling window then advances every x seconds, setting the new window:

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

3.5.2 Polling window

The polling window is the window of ΔQs which are stored to keep a snapshot of the system over time and over which confidence bounds are calculated.

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 snapshot and to the confidence bounds.

The limit of ΔQs for a polling window (subsequently snapshots and confidence bounds) is 30 ΔQs. At $t = 31$, the older ΔQs will be removed from the polling window and in turn from the snapshots and confidence bounds. Newer sampled ΔQs will be added, keeping the limit of ΔQs in a polling window to 30.

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 ΔQSD was implemented in the oscilloscope, the parameters that define a probe's ΔQ, its representation and what can be done with ΔQs. 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.

- Lastly, we explain the different controls present on the oscilloscope dashboard.

4.1 ΔQSD concepts

Originally, ΔQ(x) denotes the probability that an outcome occurs in a time $t \leq x$, defining then the "intangible mass" of such IRV as $1 - \lim_{x \to \infty} \Delta Q(x)$. [6] We then extend the original definition to fit real time constraints, needing to calculate ΔQs continuously.

For a given probe, ΔQ(t_l, t_u, $dMax$) is the probability that its instances with end time $t_l \leq t_e \leq t_u$ occur in time $t \leq dMax$.

4.1.1 Representation of a ΔQ

We provide a class to calculate the ΔQ of a probe 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 between t_l and t_u, calculating its *probability density function* (PDF) and then calculating the *empirical cumulative distribution function* (ECDF) based on its 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 then lost between calculations, as the interest shifts towards calculating the resulting PDFs and ECDFs. 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 Section 3.3.3, the probes p_2 and p_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

ΔQ histogram PDF and resulting ECDF
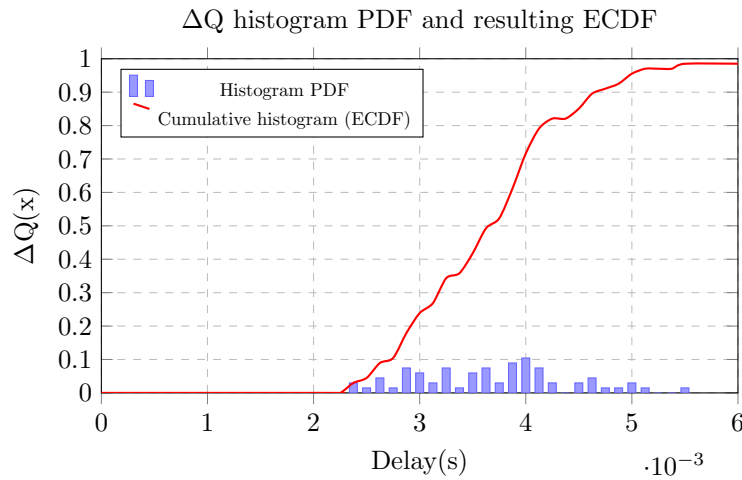
Figure 4.1: Blue bins: PDF of a sample ΔQ. Red: Resulting CDF of ΔQ PDF, the ECDF is displayed on the dashboard.

The PDFs of the observed and calculated $\Delta Qs(t_l, t_u, dMax)$ are not shown on screen, only the ECDFs are plotted. Moreover, confidence bounds can then be calculated over multiple ΔQs, we will see how in following sections.

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

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.

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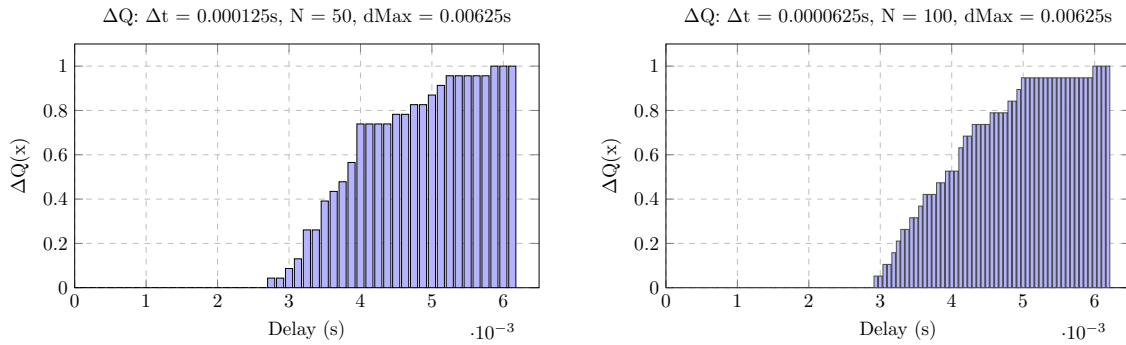
Figure 4.2: Left: Sample ΔQ with higher resolution but lower precision.
Right: Sample ΔQ with lower resolution but higher precision.
Both ΔQs have the same *dMax*, but the amount of precise information they provide is far different.

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 y observing the shown plots.
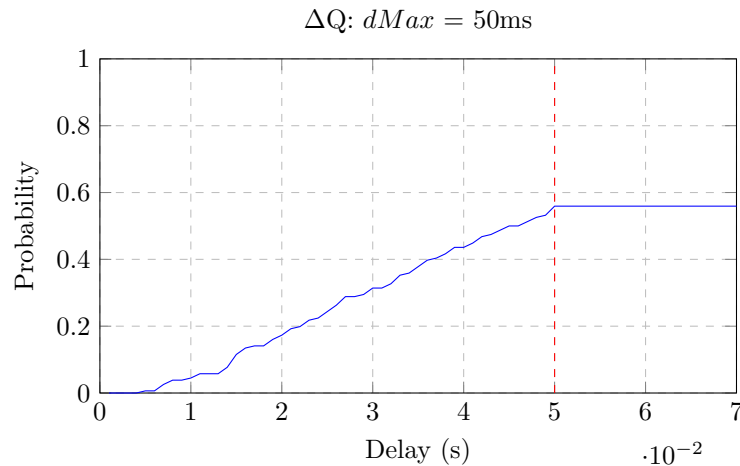
Figure 4.3: ΔQ: *dMax* = 50ms, the ΔQ will stay constant when delay > *dMax*.

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 adpater 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 maximum amount of failures accepted for an observable. An observed ΔQ will calculate that based on the samples collected.

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, moreover, as it is an ECDF, it is not as precise as a CDF and, based on the number of bins, may not be precise. This is why we include the mean and confidence bounds of ΔQs in the plot.

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, which give a probability range over which the true CDF of the ΔQ should fall. [27]

The bounds are updated dynamically by inserting or removing a ΔQ. Every time a new ΔQ is calculated, 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 calculated.
This allows us to consider a small window of execution rather than observing the execution since the start for the bounds, this can help in observing stationarity of the system, where less sampled 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.
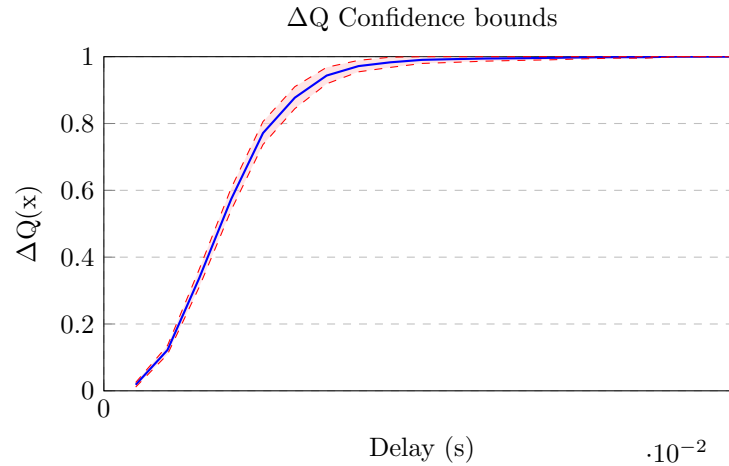
Figure 4.4: Upper and lower bounds (dashed, red) of the mean (blue) of multiple ΔQs. In a system that behaves linearly, the bounds will be close to the mean, once the overload is approaching, or a system is showing behaviour that diverges from a linear one, the bounds will appear larger.

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 following functions:

- The observed ΔQ at time t, with a mean and confidence bounds of previous observed ΔQs.

- If applicable, the calculated ΔQ from the causally linked components observed in a probe, with a mean and confidence bounds of previous calculated ΔQs.

- Its QTA (if defined).

This allows for the user to observe if a ΔQ has deviated from normal execution, analyse its stationarity, nonlinearity and observe its execution.

In the photo below we can observe the multiple elements as they are displayed in real time in the oscilloscope.

- (1): The **Observed ΔQ**.

- (2): The **Calculated ΔQ**.

- (3): The **QTA**.

- Arrow, blue, below: The mean of the observed ΔQs (yellow) with the confidence bounds of the mean. Upper bound (dark green) and lower bound (light green).

- Arrow, red, above: The mean of the calculated ΔQs (ochre) with the confidence bounds of the mean. Upper bound (purple) and lower bound (magenta).
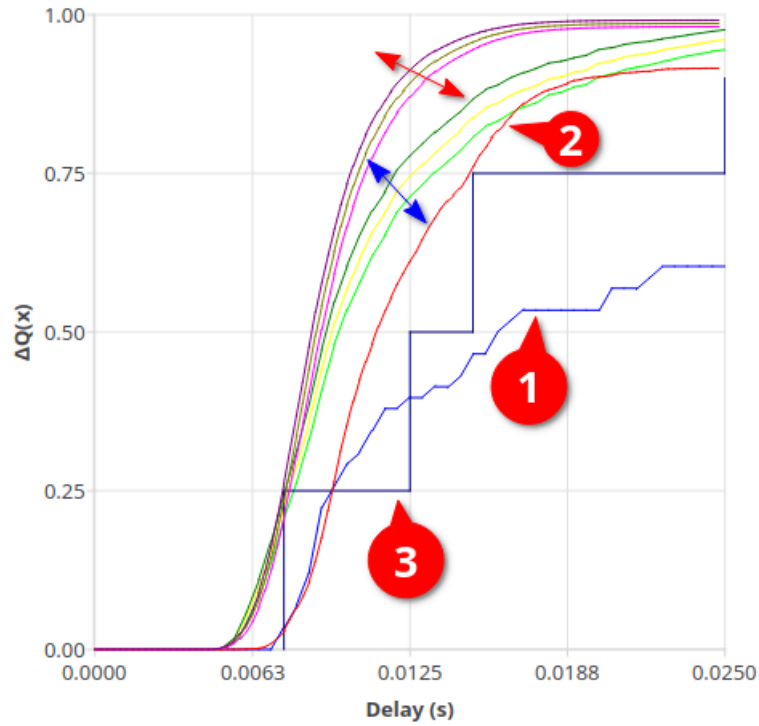
Figure 4.5: ΔQs, confidence bounds, means and QTA for a probe observing the causal link of multiple components.

4.3 Outcome diagram

An abstract syntax for constructing outcome diagrams has already been defined in a previous paper [3], nevertheless, the oscilloscope needs a textual way to define an outcome diagram.

We define thus a grammar to create an outcome diagram in our oscilloscope, our grammar 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, these "sub-outcome diagrams" can then be linked together to from the global system outcome diagram. Recall Section 3.3.3, 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 begin and end 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.

4.4   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

System creation   In this tab the user can create its system using the grammar defined before, he can save the text he used to define the system or load it, the system 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, he 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 he wants 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 calculated 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.

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 maximum amount of failures allowed, they can change this dynamically during execution.

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.

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, 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.

Connection controls

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.

4.4.2 Plots window

To the left, the main window shows the plots of the probes being updated in real time.

4.5 Triggers

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

4.5.1 Load

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

$$\#\text{instances}_{probe}(\Delta\text{Q}(t_l, t_u, dMax)) > \text{maxAllowedInstances}$$

4.5.2 QTA

A trigger on an observed ΔQ can be fired if:

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

Chapter 5

Oscilloscope: implementation

The following chapter gives a more technical description 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.

- Next we give a technical explanation of the parser generator we used to parse the outcome diagram syntax.

- Lastly, we briefly talk about the dashboard graphical framework.

5.1 ΔQSD implementation

A probe's ΔQ can be represented internally by a PDF and displayed as an ECDF. Here is how both can be calculated given n outcome instances.

PDF

We approximate the PDF of the observed ΔQ via a histogram. 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:

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

Where s_i the number of successful outcome instances whose elapsed time is contained in the bin i, n the total number of instances. [28]

ECDF

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

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

5.1.1   dMax

We introduced $dMax$ in the previous section, we provide here the full equation that allows $dMax$ to be calculated:

$$dMax = \Delta_{tbase} * 2^n * N \tag{5.3}$$

Where:

- Δ_{tbase} represents the base width of a bin, equal to 1ms.

- N the number of bins.

5.1.2   Operations

In a previous section we talked about the possible operations that can be performed on and between ΔQs, the time complexity of FTF, ATF and PC is trivially $\mathcal{O}(N)$ where N is the number of bins.

As to convolution, 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. Below we present two ways to perform convolution.

Convolution

Naïve convolution   Given two ΔQ binned PDFs f and g, the result of the convolution $f \circledast g$ is given by [29]:

$$(f \circledast g)[n] = \sum_{m=0}^{N} = f[m]g[n-m] \tag{5.4}$$

Fast Fourier Transform Convolution   FFTW (Fastest Fourier Transform in the West) is a C subroutine library [30] 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. [31]

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 $\mathcal{O}(nlogn)$. This is why the naïve convolution algorithm is not used. We will analyse the time gains in a later chapter.

FFT and naïve convolution produce the same results in our program barring ε differences (around 10^{-18}) in bins whose result should be 0.

FFTs algorithms are plenty, the choice of the one to use is left up to the subroutine via the parameter `FFTW_ESTIMATE` [32].

Arithmetical operations

We can apply a set of arithmetical operations between ΔQs ECDFs, and on a ΔQ.

Scaling (multiplication) A ΔQ can be scaled w.r.t. a constant $0 \leq j \leq 1$. It is equal to binwise multiplication on ECDF bins.

$$\hat{f}_r(i) = \hat{f}(i) \cdot j \tag{5.5}$$

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 ECDFs of the ΔQs:

$$\Delta Q_{AB}(i) = \hat{f}_A(i)[\cdot, +, -]\hat{f}_B(i) \tag{5.6}$$

5.1.3 Confidence bounds

Here is how we calculate the mean and lower/upper confidence for the ΔQs ECDF at bin $i \quad \forall i < N$. [28]

For x_{ij} the value of an ECDF j at bin i, the mean of all ECDFs for the bin over a window is:

$$\mu_i = \frac{1}{n_i} \sum_{j=1}^{n_i} x_{ij} \tag{5.7}$$

Its variance:

$$\sigma_i^2 = \frac{1}{n_i} \sum_{j=1}^{n_i} x_{ij}^2 - \mu_i^2 \tag{5.8}$$

The confidence intervals CI_i for a bin i can then be calculated as:

$$CI_i = \mu_i \cdot \frac{\sigma_i}{\sqrt{n_i}} \tag{5.9}$$

5.1.4 Rebinning

Rebinning refers to the aggregation of multiple bins of a bin width i to another bin width j. Operations between ΔQs can be done on ΔQs that have the same bin width, this is why it is fundamental that all probes have a common Δ_{tbase}. This allows for fast rebinning to a common bin width.
Given two ΔQs ΔQ_i, ΔQ_j:

$$\Delta_{Tij} = \max \{\Delta_{Ti}, \Delta_{Tj}\}$$

and the PDF of the rebinned ΔQ at bin b, from the original PDF of n bins, where $k = \frac{\Delta_T i}{\Delta_{Tj}}$:

$$p'_b = \sum_{n=b \cdot k}^{b+1 \cdot k-1} p_n, \quad b = 0, 1, \ldots \lceil \frac{N}{k} \rceil \tag{5.10}$$

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.

(a) Sample ΔQ with 1ms bins

(b) ΔQ on the left after rebinning to 4ms bins
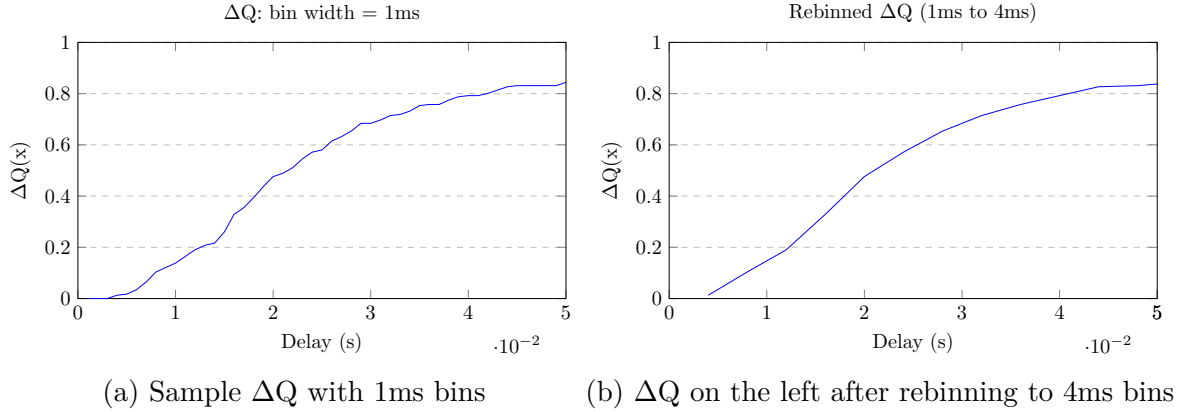
5.2 Adapter

The adapter, called `dqsd_otel` is a rebar3 [33] 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 macros as for `?start_span` and `?with_span` and `?end_span`. This is to make the adapter less of an encumbrance for the user.

Moreover, 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: Binary name of the probe.

- 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/1, with_span/2

```
with_span/1: -spec with_span(binary(), fun(() -> any())) -> any().
with_span/2: -spec with_span(binary(), map(), fun(() -> any())) ->
↪  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 OpenTelemetry `with_span` macro.

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

end_span

```
-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`.

As is the case for `start_span`, `end_span` incorporates an OpenTelemetry macro, in this case `?end_span(Ctx)`.

fail_span

```
-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 an 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 and it will send an outcome instance to the oscilloscope indicating that an execution of a probe has timed out.

The 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. [34]

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 $dMax$.

They start time and end time are supplied 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 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:Observed name, b: Start time (beginning), e: End time (end or
↪  deadline), s: The status
```

5.2.3   TCP connection

The adapter is composed of two `gen_server` 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 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 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.

5.3.1 ANTLR

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 [35].

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

ANTLR uses Adaptive LL(*) *(ALL(*))* parser, namely, it will move grammar analysis to parse-time, without the use of static grammar analysis. [36]

5.3.2 Grammar

ANTLR provides a yacc-like metalanguage [36] to write grammars. Below, is the grammar for our system:

```
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;

start: definition* system? EOF;

definition: IDENTIFIER '=' 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;
```

Limitations

A previous version was implemented in Lark [37], 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 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. [38] 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.

Chapter 6

Application on synthetic programs

This section aims to provide an example of how the oscilloscope could be used to instrument an application, in this case, a synthetic one. We explain how the ΔQSD paradigm can be applied to explore tradeoffs in design and to gain more insights into a running system.

6.1 System with sequential composition

We model a first system with two sequentially composed component. We choose two model the two components as M/M/1/K queues.

Why M/M/1/K queues? An average component in a distributed system can be modeled as an M/M/1/K, due to the exponential inter-arrival rate of messages λ, the exponential distribution of the execution delay μ, the buffer size of messages K of a component and the failure rate f. [1]

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

- λ: The arrival rate.

- s: The service time, is the time it takes to serve a message.

- μ: The service **rate** and $\mathrm{E}[s] = \dfrac{1}{\mu}$

- Offered load: $\rho = \dfrac{\lambda}{\mu}$

We will control λ to show its effects on the offered load. 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, the delay will tend to 1.

- Once ρ is approaching high load ($\rho > 0.8$) we can observe the failure increasing quickly, but we can observe the system starting to get bad after $\rho > 0.5$! [1]

6.1.1 System composition

The system has two components `worker_1`, `worker_2`. Each individual component is made of a buffer 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 then reduces its available buffer size.

The buffer notifies its worker, which then does N loops, which are defined upon start, of fictional work. The worker then passes a message to `worker_2`'s queue, which has another queue of same size, who passes the message to `worker_2`'s worker, which does the same amount of loops. When a worker completes its work, it notifies the queue, freeing one "message" from its buffer size and allowing the next message to be executed.

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

A probe "*probe*" is defined, which observes the execution from when the message is sent to `worker_1` up until `worker_2` is done.
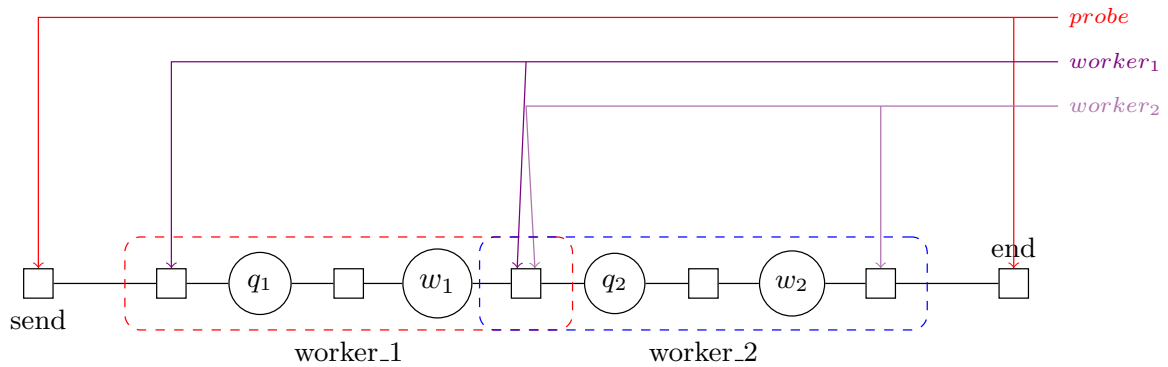
Figure 6.1: Outcome diagram of the M/M/1/K queue with the colored lines representing the probes that were inserted.

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. The oscilloscope can provide knowledge of the system, here is an example of worker_1 and worker_2 as observed in the oscilloscope.

Imagine the engineer supposes the workers executions should a maximum of 4 ms to complete, but doesn't actually know how long the executions should take. The engineer, after having set the required parameters observes in the following graph in the oscilloscope Section 6.1.2.

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 system's behaviour. By setting $dMax$ to 8 ms, they can observe the worker's ΔQs failure approaching 0.
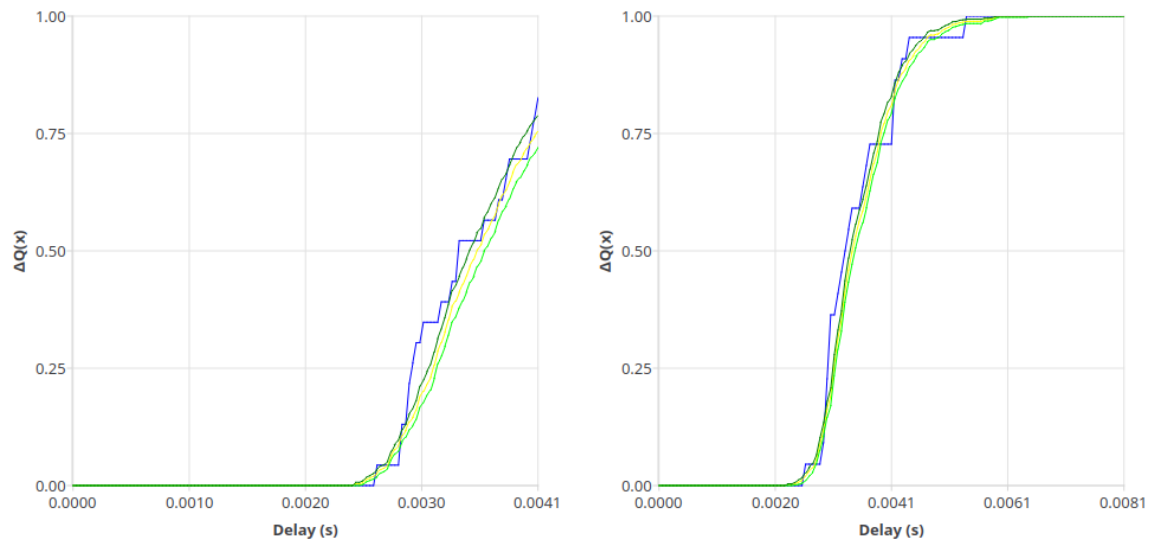
(a) worker_1 ΔQ with confidence bounds plot with 4 ms $dMax$.

(b) worker_1 ΔQ wiht confidence bounds plot with 8 ms $dMax$

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

Low Load

Let's first observe the system at low load, we will send 50 messages per second to observe the system under test to get key properties.
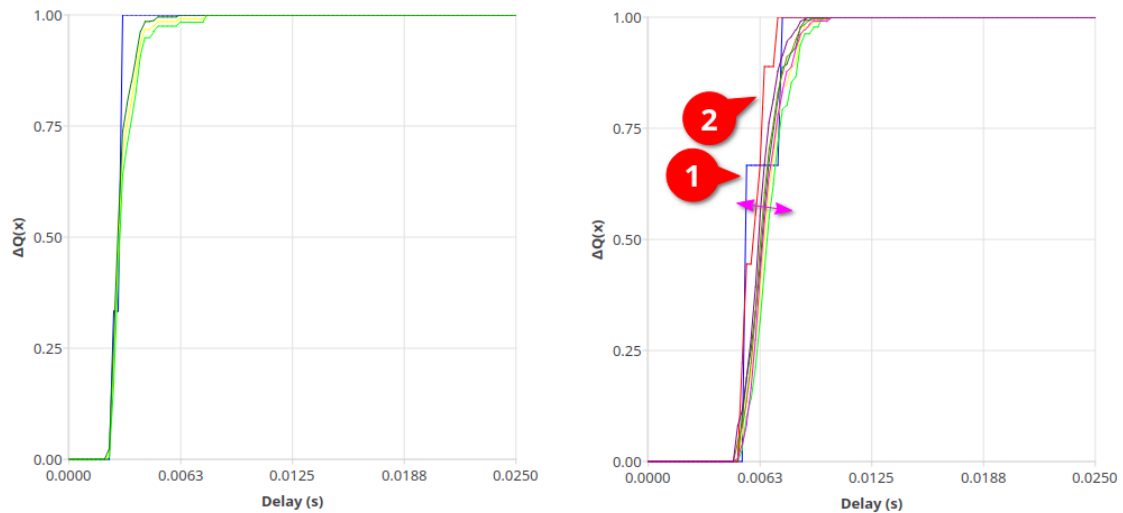
Figure 6.3: Left: worker_1 ΔQ, in blue, the observed ΔQ, in green, the confidence bounds. Right: probe ΔQ, in blue, the observed ΔQ, in red the calculated ΔQ. In magenta, the two ΔQs confidence bounds overlapping.

We first observe the worker's ΔQ, we can observe that the average execution takes $\approx$ 30ms. We then have $\mu_{worker} = \dfrac{1}{0.0033} \approx 300$ req/s. Thus $\rho = \dfrac{50}{322} = 0.16$, we are in nice grounds!

At low load, we can observe in the oscilloscope the probe **observed ΔQ** and **calculated ΔQ** confidence bounds overlap.

Early signs of overload

We can see how even at load $= 0.5$ the system is starting to show bad behaviour. Let us observe what happens when $\lambda = 150 \rightarrow \rho = 0.5$.
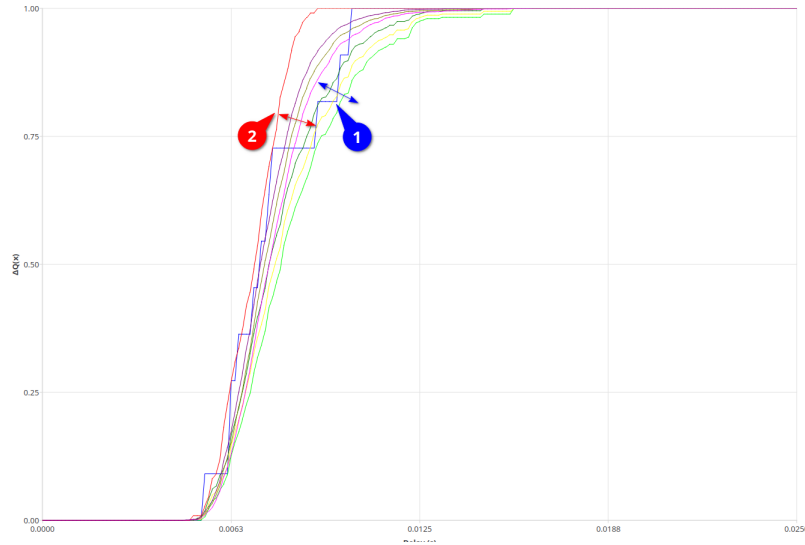
Figure 6.4: probe ΔQs, in blue (1), the observed ΔQ, in red (2) the calculated ΔQ. Arrow, above, blue: Observed ΔQ confidence bounds. Arrow, below, red: Calculated ΔQ confidence bounds.
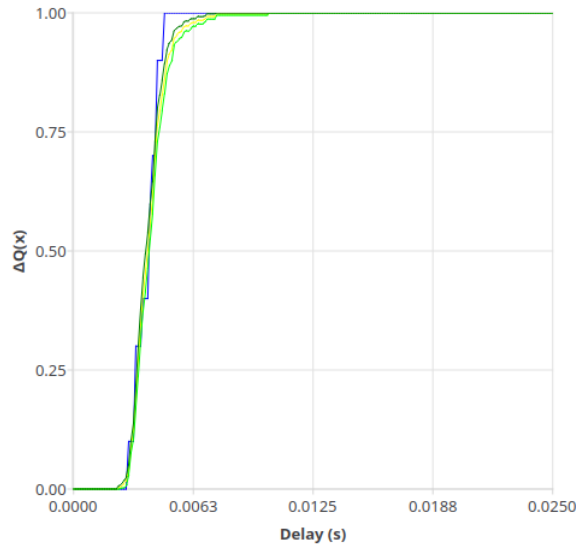
Figure 6.5: worker_1 ΔQ (blue) and its confidence bounds

.

Recall 2.11, we can start to observe early signs of dependency! At load 0.5 the calculated ΔQ is deviating from the observed one. This is a sign that the performance is degrading. Worker_1 is slowing down, but we nevertheless do not observe failures in probe's ΔQs.

High load

Performance at 0.5 offered load are already remarkable, the ΔQs are shifting. We can go even further and observe the system under high load situations. We set $\lambda = 200 \to \rho = 0.83$, just above the high load threshold.
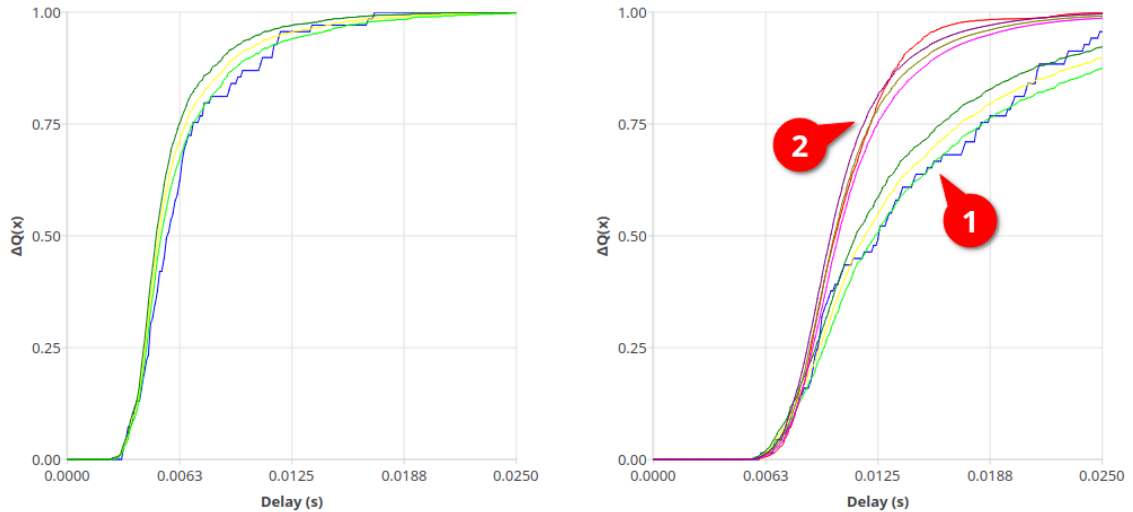
Figure 6.6: Left: worker_1 ΔQ. Right: probe ΔQ, in blue (1), the observed ΔQ with its confidence bounds, in red (2) the calculated ΔQ with its confidence bounds.

This is what we expected previously, and confirms what is expected by queueing theory, ΔQ is capable of observing the basic observation requirements and capable of recognising dependency. While what is expected by the execution of the queue (observed ΔQ) is a nice normally distributed CDF with little to no failure. What we can actually observe is a degraded performance in both workers and the probe observed execution.

The workers CDF has completely degraded, with the average request taking almost double the time as under normal queueing conditions.

Further degradation can be observed by increasing $\lambda = 300, 350 \to \rho \geq 1$.
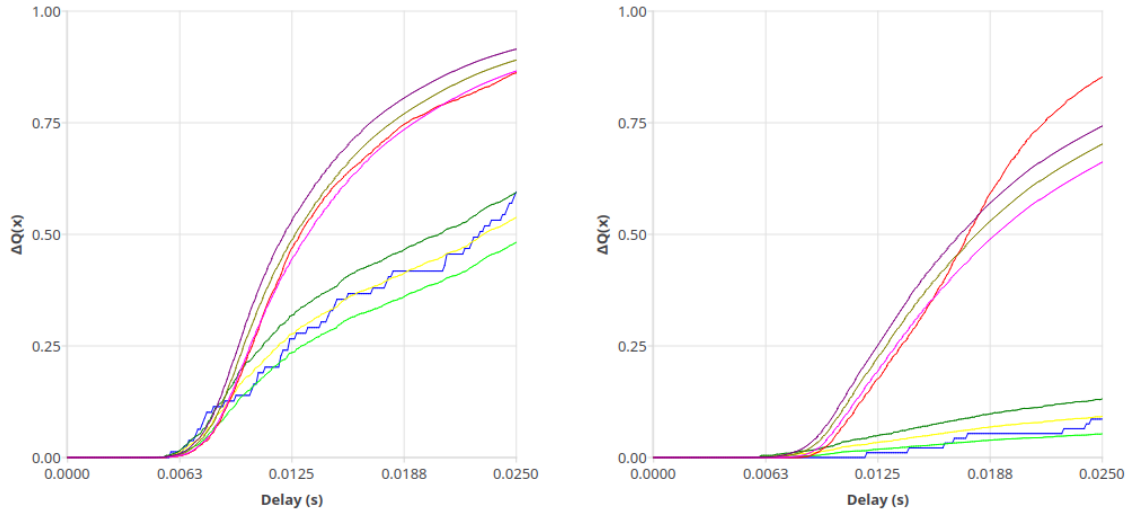
Figure 6.7: Left: probe ΔQs at $\lambda = 300$. Right: probe ΔQs at $\lambda = 350$

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

Triggers

By observing the system under test in high load cases, we can set the load trigger by setting the sampling window to 1 second and trigger when outcome instances $\gtrsim 150$. We can also set a trigger based on observation of the running system.

QTA trigger By observing the system, we create a QTA for the probe with: $25\% = 0.0075$ s, $50\% = 0.0125$ s, $75\% = 0.015$s and minimum intangible mass $= 0.9$.

By setting the trigger to fire for $\Delta Q_{obs} < QTA$. We captured a handful of snapshots. Here, $\lambda = 150$.
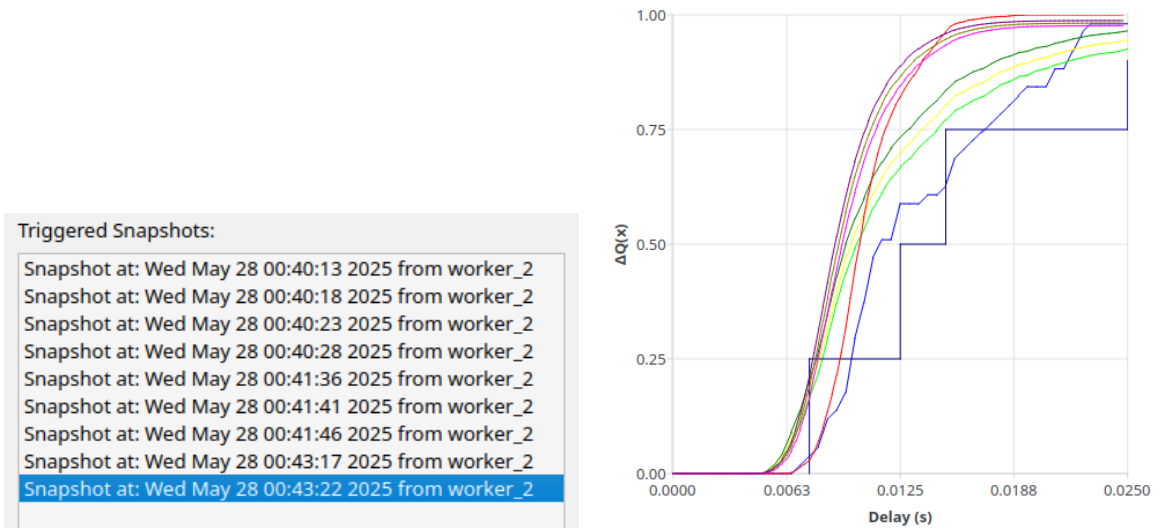
Figure 6.8: Left: Fired triggers. Right: Sample QTA violation.

QTA triggers can help to detect overhead even before high load becomes evident!

Instances trigger   By knowing the inner details of the system, setting a QTA on the number of instances can be useful. Here 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.
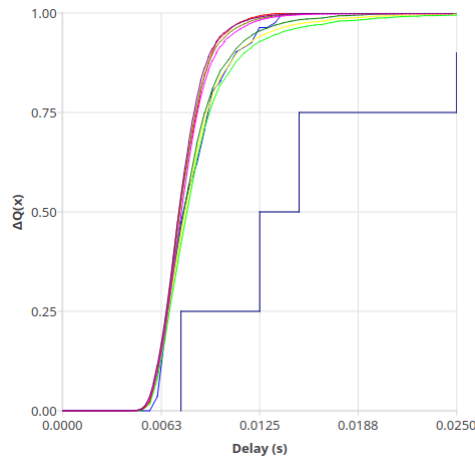
6.2   Detecting slower workers in workers

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

49

Why first to finish?   Recall the previous FTF graph Fig. 2.5. Assume a send request to "the cloud" that waits for a response or a timeout, it is modeled by a FTF operator.

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 Fig. 6.1? The result on the oscilloscope will clearly show that something is wrong!
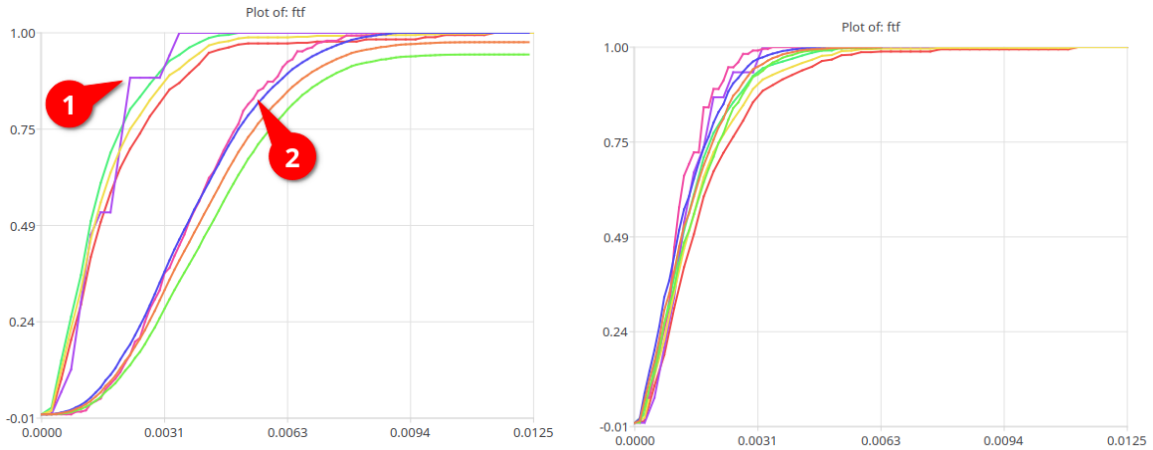
Figure 6.9: *(Left)* FTF plot **with wrong outcome diagram definition** as shown in the oscilloscope. (1) Observed ΔQ. (2) Calculated ΔQ.
(Right) FTF plot **with correct outcome diagram definition** as shown in the oscilloscope. Observed ΔQ and calculated ΔQ overlapping.

On the left, we can observe how the **calculated ΔQ** (2) is clearly greater than the **observed ΔQ** (1). A difference this drastic 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 is chosen, the two graphs will overlap.

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.
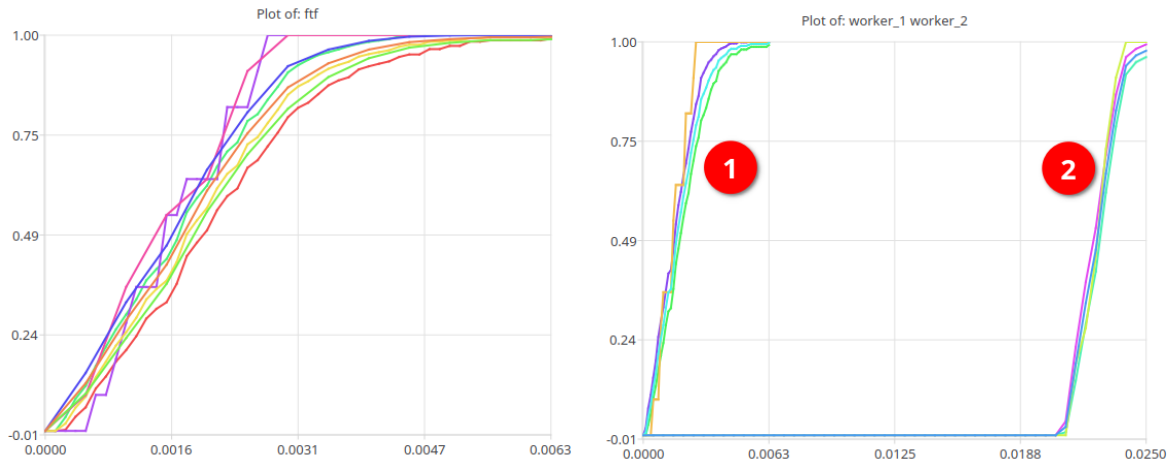
Figure 6.10: *(Left)* FTF plot of worker_1 and worker_2, observed and calculated ΔQ overlapping.
(Right) worker_1 (1) and worker_2 (2) ΔQs.
The FTF plot correctly displays how worker_2 does not have an effect on the ftf plot.

6.2.2 All to finish application

We can extend the previous application to an all-to-finish operator, this operator can for instance parallel work, a task that requires a lot of computation and can be done in separate pieces by separate workers. [1]

Introducing a slower component

Like we did for the FTF operator, let's introduce a slower work into the mix. 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). The delay is a 2ms sleep on worker_2.

Worker's performance We present here the worker's graphs as shown on the oscilloscope.
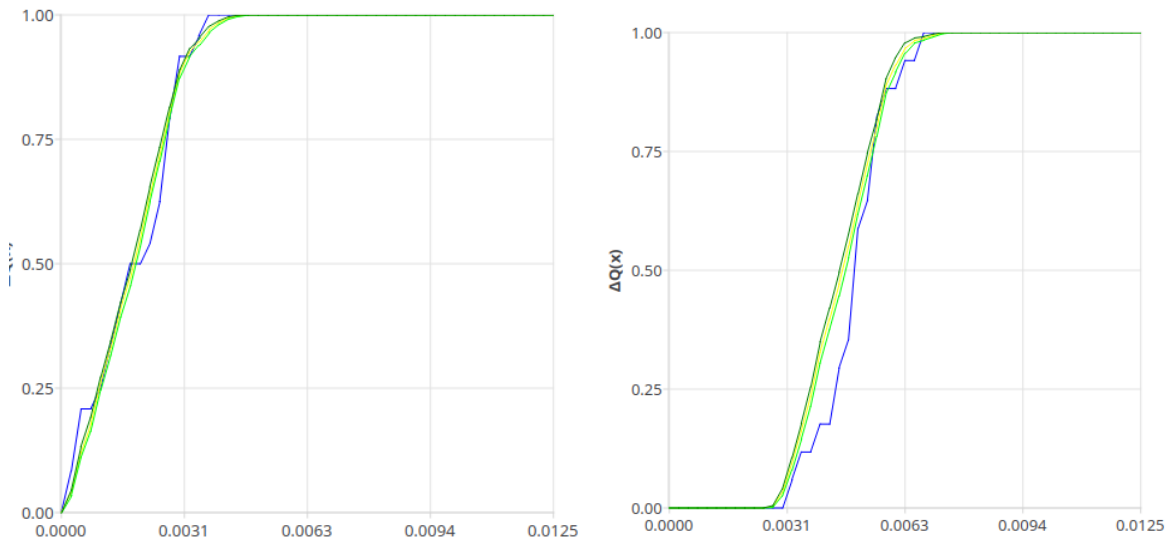
Figure 6.11: Left: worker_1 plot. Right: worker_2 plot.

The difference in the worker's ΔQ can be noticed with $\Delta Q_w 2 > \Delta Q_w 1$. The difference can then be observed in the all-to-finish plot, where the operator's ΔQs (both observed and calculated) can be overlaid on top of worker_2 ΔQ, showing once again that the ΔQSD foundation is sound.
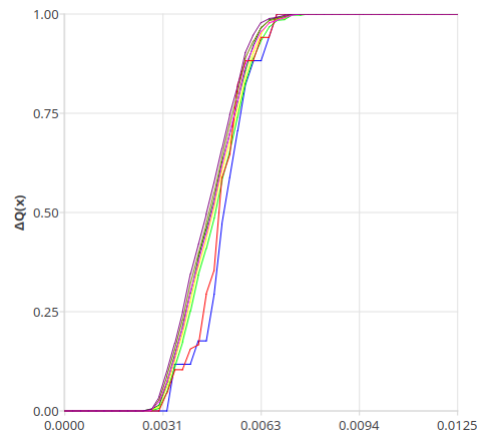
These plots show the usefulness of ΔQSD, the system can be decomposed to understand which part of the system is showing hazards, furthermore, the causal relationships can be observed to determine the behaviour of a part down to the single component.

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. We 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 evalute 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 C.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.

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 $600 > 512$, $300 > 256$ ...). This is because the algorithm is based on ΔQs which are zero-padded to the nearest power of 2, this is to make the calculation more accurate.

While we limit the number of bins to 1000 right now, this limit could potentially be
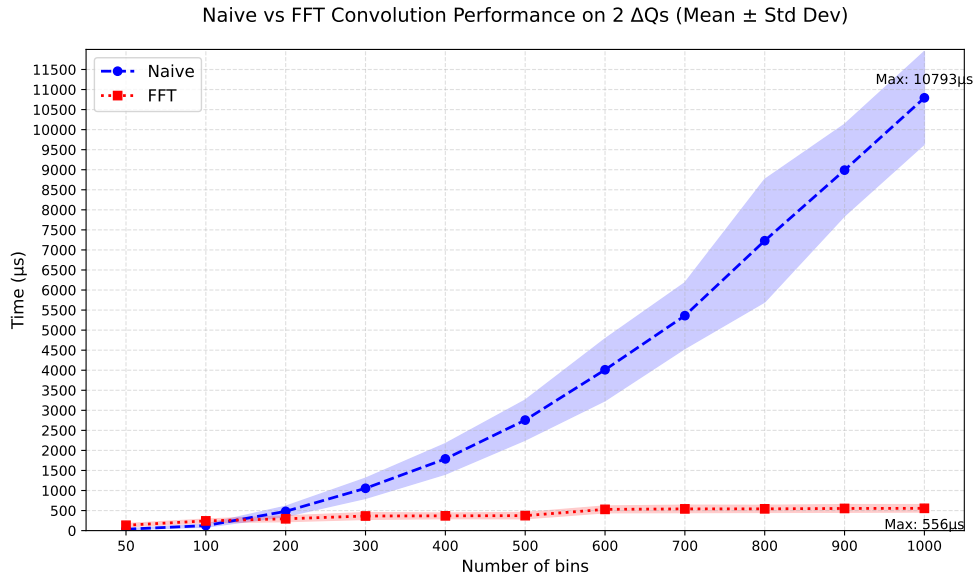
Figure 7.1: Performance comparison of two convolution algorithms

increased as the convolution algorithm is very efficient (0.5 ms for 1024 bins).

7.2 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` $\rightarrow$ `end_span`.

- `with_span` with the following function: `fun()` $\rightarrow$ `ok`.

- `start_span` $\rightarrow$ `fail_span`.

We ran the simulation for 25000 subsequent iterations, these are the results.
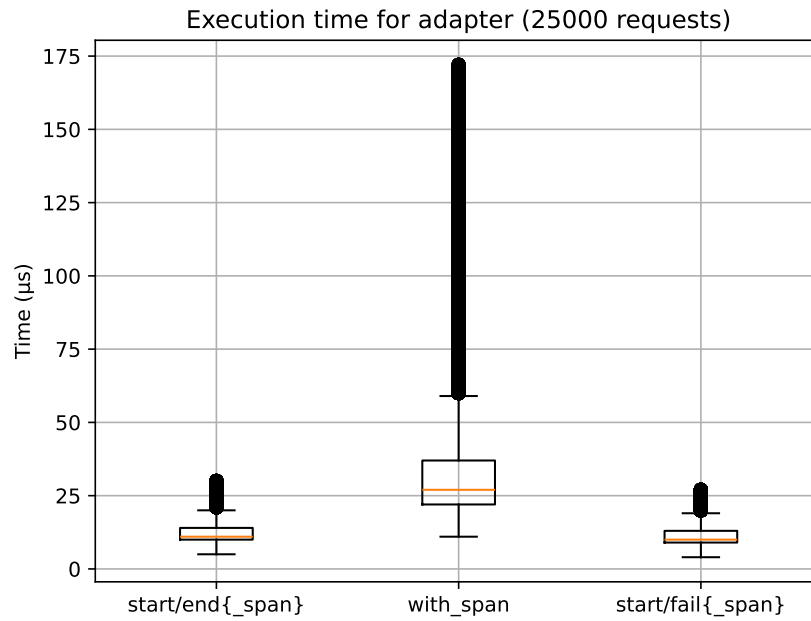
The overhead is minimal, around 10 microseconds on average to start and end/fail a span. The same cannot be said about with span, the increased overhead is nevertheless due to a function needing to be called inside it for it to record a span.

7.3   GUI plotting performance

We evaluated the performance of the GUI plotting routine for an observed ΔQ. The routine first prepares the ΔQ, creating vectors QPointF (a Qt class representing a point for a QtChart), representing the x and y values of the ΔQs CDF. The vectors are created for the lower bound, the upper bound the mean of the window of ΔQs and the observed ΔQ.

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

Mean time of computing vectors of QPointF and displaying observed ΔQ series
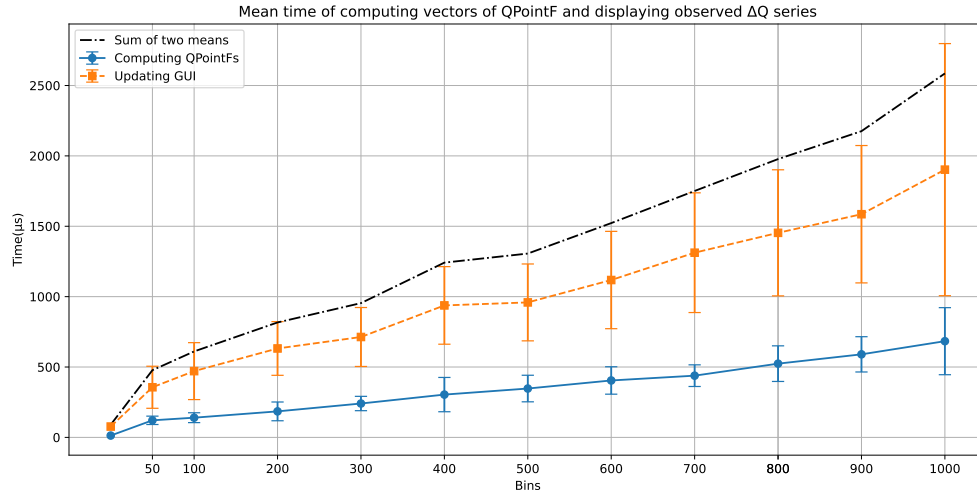
The result scales up to 2 ms for 1000 bins, since multiple plots have to be shown, if the probe observes the causal relationships of outcome and operators, the calculated ΔQ will need to be plotted too, the time increase will be twofold. If a user decides to have finer grained representations and multiple plots at once, decreasing (decreasing or increasing?) the polling rate will avoid the plots frame skipping.

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 $\Delta\mathbf{Q}$ **oscilloscope**, to observe running Erlang applications and plot the system behaviour in real time thanks to the ΔQSD paradigm to diagnose performance issues and to optimise performance over lifetime.
While we can not say that we are 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 application 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 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 over to the user of the outcome diagrams, the parameters of probes, their QTAs, the triggers they want to set for a given probe, a system of snapshots which allows observing 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. Thise

8.1 Future improvements

We believe the oscilloscope and the Erlang application can be drastically improved, the size of the project and its intended goal is too big to be encompassed in a single master thesis. 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, 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 Adapter improvements

- As suggested by Bryan Naegele, a member of the observability group of Erlang, the adapter, instead of working on top of OpenTelemetry, could be directly included inside the context of a span by using the ctx library [39], which provides deadlines for contexts, propagating the value in otel_ctx, making it available to the OpenTelemetry span processor. Leveraging `erlang:send_after` as we already do, we could create outcome instances with telemetry events to handle successful executions and timeouts. The span processor will then be responsible for creating outcome instances, without creating the need for custom functions in the adapter, like we have now.

8.1.3 Real applications

A flaw of the oscilloscope and adapter is that they have not been tested on real applications, while their usefulness has been proven on synthetic applications, the lack of real life applications is a weakness.

8.1.4 Licensing limitations

Lastly, a notable limitation is created by **Qt**, namely, QtCharts. The usage of Qt does not allow us to release our project under BSD/MIT licenses, but rather a GPLv3 one (we cannot release it under LGPL due to QtCharts). [40]

Bibliography

[1]   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`.

[2]   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`.

[3]   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`.

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

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

[6]   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`.

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

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

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

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

[11]   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.

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

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

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

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

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

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

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

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

[20] 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/.

[21] 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/.

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

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

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

[25] Francesco Nieri. *$\Delta QOscilloscope$*. Accessed: (28/05/2025). URL: https://github.com/fnieri/DeltaQOscilloscope.

[26] 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.

[27] 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.

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

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

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

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

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

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

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

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

[36] 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`.

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

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

[39] Tristan Sloughter. *ctx.* Accessed: (21/05/2025). 2023. URL: `https://github.com/tsloughter/ctx`.

[40] 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

Appendix B

User manual

B.1 Sidebar: Outcome diagram and plots

B.1.1 Creating the system

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

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.

B.1.2 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.

B.1.3 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, as for saving, you can load any file extension.
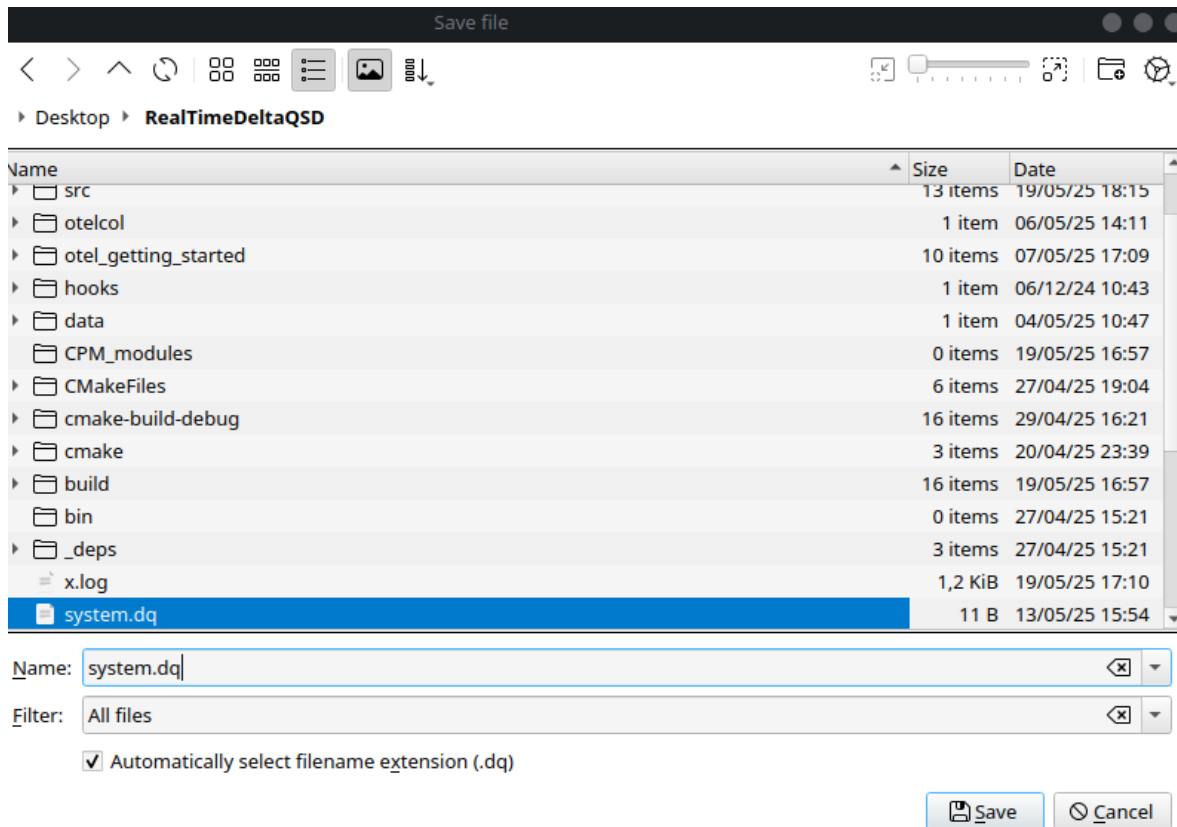
B.1.4   Managing the plots

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. Then by clicking the **"Add plot"** button, the selected probes will be added to the plot.

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 components"** zone, by clicking **"Remove selected components"** the components will be removed from the plot. Inversely, in the **"Available components"** section, you can select the plots you want to add, and by clicking **"Add selected components"** you can add the selected components to the selected plot.

B.1.5   Removing a plot

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

B.2   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.

B.2.1   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.

B.2.2   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.

Figure B.1: Probes settings tab. Above: QTA settings. Below: Probe parameters settings.

B.3 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 *"Deleted snapshot"*.

B.4 Instrumenting the Erlang application

B.4.1 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,
        opentelemetry,
        opentelemetry_api,
    opentelemetry_exporter,
    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 were to change name, you can still find the project in https://github.com/fnieri/).

B.4.2 Starting spans

To start spans you need to call:

```
{ProbeCtx, Pid} = dqsd_otel:start_span(<<"probe">>, #{attributes =>
↪  [{attr, <<"my_attr_5o5s10">>}]}),
```

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.

B.4.3 Ending spans

To end spans 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 timedout The function calls OpenTelemetry `?end_span` macro, effectively replacing it.

B.4.4 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.

B.5 Establishing connection to the oscilloscope

WIP

Part I

Source Code Appendix

Appendix C

C++ Source Files

C.1 Root folder

C.1.1 main.cpp

```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)
{
    // 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);
```

```
34
35       app.setPalette(lightPalette);
36   }
37
38   int main(int argc, char *argv[])
39   {
40
41       Server server(8080);
42       signal(SIGPIPE, SIG_IGN); // Ignore SIGPIPE so when Erlang closes
         socket it will not crash
43       Application &application = Application::getInstance();
44       application.setServer(&server);
45       System system = System();
46       application.setSystem(system);
47       QApplication app(argc, argv);
48       setLightMode(app);
49       MainWindow window;
50       window.show();
51
52       int result = app.exec();
53       server.stop();
54       return result;
55   }
```

C.1.2 Application.cpp

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

```
1   #include "Application.h"
2
3   Application::Application()
4   {
5   }
6   Application &Application::getInstance()
7   {
8       static Application instance;
9       return instance;
10  }
11
12  std::shared_ptr<System> Application::getSystem()
13  {
14      return system;
15  }
16
17  void Application::addObserver(std::function<void()> callback)
18  {
19      observers.push_back(callback);
20  }
21
22  void Application::notifyObservers()
23  {
24      for (auto &observer : observers) {
25          observer();
26      }
27  }
```

72

```
28
29  void Application::setServer(Server *s)
30  {
31      server = s;
32  }
33
34  void Application::sendDelayChange(std::string &name, double newDelay)
35  {
36      server->sendToErlang("set_timeout;" + name + ';' + std::to_string(
    newDelay));
37  }
38
39  void Application::setStubRunning(bool running)
40  {
41      if (running)
42          server->sendToErlang("start_stub");
43      else
44          server->sendToErlang("stop_stub");
45  }
46
47  SystemDiff Application::diffWith(System &newSystem)
48  {
49      SystemDiff diff;
50
51      auto oldSystem = getSystem();
52      if (!oldSystem) {
53          // No system yet: everything is new
54          for (const auto &[name, _] : newSystem.getProbes())
55              diff.addedProbes.push_back(name);
56          for (const auto &[name, _] : newSystem.getOperators())
57              diff.addedOperators.push_back(name);
58          for (const auto &[name, _] : newSystem.getOutcomes())
59              diff.addedOutcomes.push_back(name);
60          return diff;
61      }
62
63      auto &oldProbes = oldSystem->getProbes();
64      auto &newProbes = newSystem.getProbes();
65
66      for (const auto &[name, probe] : newProbes) {
67          if (!oldProbes.count(name)) {
68              diff.addedProbes.push_back(name);
69          } else if (componentsDiffer(oldProbes.at(name), probe)) {
70              diff.changedProbes.push_back(name);
71          }
72      }
73      for (const auto &[name, _] : oldProbes) {
74          if (!newProbes.count(name))
75              diff.removedProbes.push_back(name);
76      }
77
78      auto &oldOps = oldSystem->getOperators();
79      auto &newOps = newSystem.getOperators();
80
81      for (const auto &[name, op] : newOps) {
82          if (!oldOps.count(name)) {
```

```
 83                diff.addedOperators.push_back(name);
 84            } else if (componentsDiffer(oldOps.at(name), op)) {
 85                diff.changedOperators.push_back(name);
 86            }
 87        }
 88        for (const auto &[name, _] : oldOps) {
 89            if (!newOps.count(name))
 90                diff.removedOperators.push_back(name);
 91        }
 92
 93        auto &oldOut = oldSystem->getOutcomes();
 94        auto &newOut = newSystem.getOutcomes();
 95
 96        for (const auto &[name, out] : newOut) {
 97            if (!oldOut.count(name)) {
 98                diff.addedOutcomes.push_back(name);
 99            } else if (componentsDiffer(oldOut.at(name), out)) {
100                diff.changedOutcomes.push_back(name);
101            }
102        }
103        for (const auto &[name, _] : oldOut) {
104            if (!newOut.count(name))
105                diff.removedOutcomes.push_back(name);
106        }
107
108        return diff;
109 }
110
111 bool Application::componentsDiffer(const std::shared_ptr<Observable> &
        a, const std::shared_ptr<Observable> &b)
112 {
113     if (!a || !b || a->getName() != b->getName())
114         return true;
115
116     // For Probes, check causalLinks. For Operators, check type,
        probabilities, children.
117     if (auto pa = std::dynamic_pointer_cast<Probe>(a)) {
118         auto pb = std::dynamic_pointer_cast<Probe>(b);
119         if (!pb)
120             return true;
121
122         auto ca = pa->getCausalLinks();
123         auto cb = pb->getCausalLinks();
124
125         if (ca.size() != cb.size())
126             return true;
127         for (size_t i = 0; i < ca.size(); ++i) {
128             if (ca[i]->getName() != cb[i]->getName())
129                 return true;
130         }
131     }
132
133     if (auto oa = std::dynamic_pointer_cast<Operator>(a)) {
134         auto ob = std::dynamic_pointer_cast<Operator>(b);
135         if (!ob)
136             return true;
```

```
137
138          if (oa->getType() != ob->getType())
139              return true;
140
141          auto ca = oa->getCausalLinks();
142          auto cb = ob->getCausalLinks();
143
144          if (ca.size() != cb.size())
145              return true;
146          for (size_t i = 0; i < ca.size(); ++i) {
147              if (ca[i].size() == cb[i].size()) {
148                  for (size_t j = 0; j < ca[i].size(); ++j) {
149                      if (ca[i][j]->getName() != cb[i][j]->getName())
150                          return true;
151                  }
152
153              } else {
154                  return true;
155              }
156          }
157          auto probsA = oa->getProbabilities();
158          auto probsB = ob->getProbabilities();
159          if (probsA != probsB)
160              return true;
161      }
162
163      return false;
164 }
165
166 void Application::setSystem(System newSystem)
167 {
168      if (!system) {
169          system = std::make_shared<System>(newSystem);
170          notifyObservers();
171          return;
172      }
173      SystemDiff diff = diffWith(newSystem);
174
175      // Apply removals
176      for (const auto &name : diff.removedProbes) {
177          system->getProbes().erase(name);
178          system->getObservables().erase(name);
179      }
180      for (const auto &name : diff.removedOperators) {
181          system->getOperators().erase(name);
182          system->getObservables().erase(name);
183      }
184
185      for (const auto &name : diff.removedOutcomes) {
186          system->getOutcomes().erase(name);
187          system->getObservables().erase(name);
188      }
189
190      // Apply changes and additions
191      for (const auto &name : diff.changedProbes) {
192          system->getProbes()[name]->setCausalLinks(newSystem.getProbes
```

```
      ().at(name)->getCausalLinks());
193         system->getObservables()[name] = newSystem.getProbes().at(name
      );
194      }
195      for (const auto &name : diff.addedProbes) {
196          system->getProbes()[name] = newSystem.getProbes().at(name);
197          system->getObservables()[name] = newSystem.getProbes().at(name
      );
198      }
199      for (const auto &name : diff.changedOperators) {
200          system->getOperators()[name] = newSystem.getOperators().at(
      name);
201          system->getObservables()[name] = newSystem.getOperators().at(
      name);
202      }
203      for (const auto &name : diff.addedOperators) {
204          system->getOperators()[name] = newSystem.getOperators().at(
      name);
205          system->getObservables()[name] = newSystem.getOperators().at(
      name);
206      }
207
208      for (const auto &name : diff.changedOutcomes)
209          system->getOutcomes()[name] = newSystem.getOutcomes().at(name)
      ;
210      for (const auto &name : diff.addedOutcomes) {
211          system->getOutcomes()[name] = newSystem.getOutcomes().at(name)
      ;
212          system->getObservables()[name] = newSystem.getOutcomes().at(
      name);
213      }
214      std::string defText = newSystem.getSystemDefinitionText();
215      system->setSystemDefinitionText(defText);
216
217      notifyObservers();
218 }
219
220 bool Application::startCppServer(const std::string &&ip, int port)
221 {
222      return server->startServer(ip, port);
223 }
224
225 void Application::stopCppServer()
226 {
227      server->stopServer();
228 }
229 void Application::setErlangEndpoint(const std::string &&ip, int port)
230 {
231      server->setErlangEndpoint(ip, port);
232 }
```

C.2   Dashboard

This folder contains the widgets that compose the dashboard.

C.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;
}
```

C.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();
}
```

C.2.3   CustomLegendPanel.cpp

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

```cpp
#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);
    }
```

```
51  }
52
53  /**
54   * @brief Clears all entries from the legend panel.
55   */
56  void CustomLegendPanel::clear()
57  {
58      for (auto it = legendEntries.begin(); it != legendEntries.end();)
        {
59          QWidget *entry = it->second;
60          legendLayout->removeWidget(entry);
61          entry->deleteLater();
62          it = legendEntries.erase(it);
63      }
64  }
```

C.2.4 DQPlotController.cpp

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

```
1
2   #include "DQPlotController.h"
3   #include "../Application.h"
4   #include <QMetaObject>
5   #include <QVector>
6   #include <QtConcurrent>
7   #include <algorithm>
8   #include <cstdlib>
9   #include <qcontainerfwd.h>
10  #include <qlineseries.h>
11  using namespace std::chrono;
12  DQPlotController::DQPlotController(DeltaQPlot *plot, const std::vector
        <std::string> &selectedItems)
13      : plot(plot)
14  {
15      auto system = Application::getInstance().getSystem();
16      for (const auto &name : selectedItems) {
17          addComponent(name, (system->hasOutcome(name)));
18      }
19      setTitle();
20  }
21
22  DQPlotController::~DQPlotController()
23  {
24      outcomes.clear();
25      probes.clear();
26      operators.clear();
27  }
28
29  bool DQPlotController::isEmptyAfterReset()
30  {
31      auto system = Application::getInstance().getSystem();
32
33      std::lock_guard<std::mutex> lock(resetMutex);
34
```

```cpp
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);
        }
    }
```

```
87      auto system = Application::getInstance().getSystem();
88
89      for (const auto &name : selectedItems) {
90          if (!containsComponent(name)) {
91              addComponent(name, system->hasOutcome(name));
92          }
93      }
94      setTitle();
95  }
96
97  void DQPlotController::addComponent(const std::string &name, bool
        isOutcome)
98  {
99      auto system = Application::getInstance().getSystem();
100     if (isOutcome) {
101         addOutcomeSeries(name);
102     } else {
103         addExpressionSeries(name, system->hasProbe(name));
104     }
105 }
106
107 std::vector<std::string> DQPlotController::getComponents()
108 {
109     std::vector<std::string> components;
110     components.reserve(probes.size() + outcomes.size() + operators.
        size());
111
112     for (const auto &kv : probes) {
113         components.push_back(kv.first);
114     }
115     for (const auto &kv : outcomes) {
116         components.push_back(kv.first);
117     }
118     for (const auto &kv : operators) {
119         components.push_back(kv.first);
120     }
121     return components;
122 }
123
124 QLineSeries *DQPlotController::createAndAddLineSeries(const std::
        string &legendName)
125 {
126     auto series = new QLineSeries();
127     plot->addSeries(series, legendName);
128     return series;
129 }
130
131 void DQPlotController::addOutcomeSeries(const std::string &name)
132 {
133     auto system = Application::getInstance().getSystem();
134
135     auto lowerBoundSeries = createAndAddLineSeries(name + "(O) lower
        bound");
136     auto upperBoundSeries = createAndAddLineSeries(name + "(O) upper
        bound");
137     auto outcomeSeries = createAndAddLineSeries(name + "(O) observed")
```

81

```cpp
      ;
138       auto meanSeries = createAndAddLineSeries(name + "(O) mean");
139       auto qtaSeries = createAndAddLineSeries(name + "(O) qta");

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

144       outcomes[name] = {series, system->getOutcome(name)};
145   }

147   void DQPlotController::addExpressionSeries(const std::string &name,
        bool isProbe)
148   {
149       auto system = Application::getInstance().getSystem();

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

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

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

163       ExpressionSeries series = {
164           .obsS = obsSeries,
165           .obsLowerBoundS = obsLowerBoundSeries,
166           .obsUpperBoundS = obsUpperBoundSeries,
167           .obsMeanS = obsMeanSeries,
168           .calcS = calcSeries,
169           .calcLowerBoundS = calcLowerBoundSeries,
170           .calcUpperBoundS = calcUpperBoundSeries,
171           .calcMeanS = calcMeanSeries,
172           .qtaS = qtaSeries,
173       };
174       if (isProbe)
175           probes[name] = {series, system->getProbe(name)};
176       else
177           operators[name] = {series, system->getOperator(name)};
178   }

180   void DQPlotController::removeOutcomeSeries(const std::string &name)
181   {
182       OutcomeSeries series = outcomes[name].first;

184       plot->removeSeries(series.outcomeS);
185       delete series.outcomeS;
```

```
186     series.outcomeS = NULL;
187
188     plot->removeSeries(series.lowerBoundS);
189     delete series.lowerBoundS;
190     series.lowerBoundS = NULL;
191
192     plot->removeSeries(series.upperBoundS);
193     delete series.upperBoundS;
194     series.upperBoundS = NULL;
195
196     plot->removeSeries(series.meanS);
197     delete series.meanS;
198     series.meanS = NULL;
199
200     plot->removeSeries(series.qtaS);
201     delete series.qtaS;
202     series.qtaS = NULL;
203
204     outcomes.erase(name);
205 }
206
207 void DQPlotController::removeExpressionSeries(const std::string &name,
        bool isProbe)
208 {
209     ExpressionSeries series;
210     if (isProbe)
211         series = probes[name].first;
212     else
213         series = operators[name].first;
214
215     plot->removeSeries(series.obsLowerBoundS);
216     delete series.obsLowerBoundS;
217     series.obsLowerBoundS = NULL;
218
219     plot->removeSeries(series.obsUpperBoundS);
220     delete series.obsUpperBoundS;
221     series.obsUpperBoundS = NULL;
222
223     plot->removeSeries(series.obsS);
224     delete series.obsS;
225     series.obsS = NULL;
226
227     plot->removeSeries(series.obsMeanS);
228     delete series.obsMeanS;
229     series.obsMeanS = NULL;
230
231     plot->removeSeries(series.qtaS);
232     delete series.qtaS;
233     series.qtaS = NULL;
234
235     plot->removeSeries(series.calcS);
236     delete series.calcS;
237     series.calcS = NULL;
238
239     plot->removeSeries(series.calcLowerBoundS);
240     delete series.calcLowerBoundS;
```

```
241        series.calcLowerBoundS = NULL;

242

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

246

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

250

251        if (isProbe)
252            probes.erase(name);
253        else
254            operators.erase(name);
255 }

256

257 void DQPlotController::removeComponent(const std::string &name)
258 {
259        if (outcomes.count(name)) {
260            removeOutcomeSeries(name);
261        }
262        removeExpressionSeries(name, probes.count(name));
263 }

264

265 void DQPlotController::update(uint64_t timeLowerBound, uint64_t
       timeUpperBound)
266 {
267        std::lock_guard<std::mutex> lock(updateMutex);
268        double outcomeMax = 0;

269

270        for (auto &[name, seriesOutcome] : outcomes) {
271            double outcomeRange = updateOutcome(seriesOutcome.first,
       seriesOutcome.second, timeLowerBound, timeUpperBound);
272            if (outcomeRange > outcomeMax) {
273                outcomeMax = outcomeRange;
274            }
275        }

276

277        double probeMax = 0;
278        for (auto &[name, seriesProbe] : probes) {
279            if (seriesProbe.second) {
280                double probeRange = updateProbe(probes[name].first, probes
       [name].second, timeLowerBound, timeUpperBound);
281                if (probeRange > probeMax) {
282                    probeMax = probeRange;
283                }
284            }
285        }

286

287        double operatorMax = 0;
288        for (auto &[name, seriesOp] : operators) {
289            if (seriesOp.second) {
290                double opRange = updateOperator(operators[name].first,
       operators[name].second, timeLowerBound, timeUpperBound);
291                if (opRange > operatorMax) {
292                    operatorMax = opRange;
```

```cpp
293                 }
294             }
295         }
296         plot->updateXRange(std::max({outcomeMax, probeMax, operatorMax}));
297 }
298
299 double DQPlotController::updateOutcome(OutcomeSeries &series, const
        std::shared_ptr<Outcome> &outcome, uint64_t timeLowerBound,
        uint64_t timeUpperBound)
300 {
301     auto ret = QtConcurrent::run([=]() {
302         double maxDelay = outcome->getMaxDelay();
303         DeltaQRepr repr = outcome->getObservedDeltaQRepr(
        timeLowerBound, timeUpperBound);
304
305         DeltaQ deltaQ = repr.deltaQ;
306         std::vector<Bound> bounds = repr.bounds;
307         int size = deltaQ.getBins();
308         double binWidth = deltaQ.getBinWidth();
309
310         QVector<QPointF> deltaQData;
311         deltaQData.emplace_back(QPointF(0, 0));
312         deltaQData.reserve(size);
313
314         QVector<QPointF> lowerBoundData;
315         lowerBoundData.emplace_back(QPointF(0, 0));
316         lowerBoundData.reserve(size);
317
318         QVector<QPointF> upperBoundData;
319         upperBoundData.emplace_back(QPointF(0, 0));
320         upperBoundData.reserve(size);
321
322         QVector<QPointF> meanData;
323         meanData.emplace_back(QPointF(0, 0));
324         meanData.reserve(size);
325
326         QVector<QPointF> qtaData;
327         auto qta = outcome->getQTA();
328         for (int i = 0; i < size; ++i) {
329             double x = binWidth * (i + 1);
330
331             deltaQData.emplace_back(QPointF(x, deltaQ.cdfAt(i)));
332             lowerBoundData.emplace_back(QPointF(x, bounds[i].
        lowerBound));
333             upperBoundData.emplace_back(QPointF(x, bounds[i].
        upperBound));
334             meanData.emplace_back(QPointF(x, bounds[i].mean));
335         }
336
337         if (qta.defined) {
338             qtaData.reserve(8);
339             qtaData.emplace_back(qta.perc_25, 0);
340             qtaData.emplace_back(qta.perc_25, 0.25);
341             qtaData.emplace_back(qta.perc_50, 0.25);
342             qtaData.emplace_back(qta.perc_50, 0.5);
343             qtaData.emplace_back(qta.perc_75, 0.5);
```

```
344              qtaData.emplace_back(qta.perc_75, 0.75);
345              qtaData.emplace_back(maxDelay, 0.75);
346              qtaData.emplace_back(maxDelay, qta.cdfMax);
347          }
348          QMetaObject::invokeMethod(
349              plot,
350              [=]() {
351                  //        auto guiStart = high_resolution_clock::now()
    ;

352
353                  plot->setUpdatesEnabled(false);
354
355                  plot->updateSeries(series.outcomeS, deltaQData);
356                  plot->updateSeries(series.lowerBoundS, lowerBoundData)
    ;
357                  plot->updateSeries(series.upperBoundS, upperBoundData)
    ;
358                  plot->updateSeries(series.meanS, meanData);
359                  plot->updateSeries(series.qtaS, qtaData);
360
361                  plot->setUpdatesEnabled(true);
362
363                  //    auto guiEnd = high_resolution_clock::now();
364                  //      qDebug() << "GUI update took" << duration_cast
    <microseconds>(guiEnd - guiStart).count() << "";
365              },
366              Qt::QueuedConnection);
367      });
368      ret.waitForFinished();
369      return outcome->getMaxDelay();
370 }
371
372 double DQPlotController::updateOperator(ExpressionSeries &series, std
    ::shared_ptr<Operator> &op, uint64_t timeLowerBound, uint64_t
    timeUpperBound)
373 {
374      updateExpression(series, op->getObservedDeltaQRepr(timeLowerBound,
     timeUpperBound), op->getCalculatedDeltaQRepr(timeLowerBound,
    timeUpperBound),
375          op->getQTA(), op->getMaxDelay());
376
377      return op->getMaxDelay();
378 }
379
380 double DQPlotController::updateProbe(ExpressionSeries &series, std::
    shared_ptr<Probe> &probe, uint64_t timeLowerBound, uint64_t
    timeUpperBound)
381 {
382      updateExpression(series, probe->getObservedDeltaQRepr(
    timeLowerBound, timeUpperBound), probe->getCalculatedDeltaQRepr(
    timeLowerBound, timeUpperBound),
383          probe->getQTA(), probe->getMaxDelay());
384      return probe->getMaxDelay();
385 }
386
387 void DQPlotController::updateExpression(ExpressionSeries &series,
```

```cpp
       DeltaQRepr &&obsRepr, DeltaQRepr &&calcRepr, QTA &&qta, double
       maxDelay)
388 {
389     auto ret = QtConcurrent::run([=]() {
390         auto computeStart = high_resolution_clock::now();
391
392         DeltaQ obsDeltaQ = obsRepr.deltaQ;
393         std::vector<Bound> obsBounds = obsRepr.bounds;
394         DeltaQ calcDeltaQ = calcRepr.deltaQ;
395         std::vector<Bound> calcBounds = calcRepr.bounds;
396         int observedBins = obsDeltaQ.getBins();
397         double observedBinWidth = obsDeltaQ.getBinWidth();
398
399         // --- Prepare data ---
400         QVector<QPointF> obsDeltaQData;
401         obsDeltaQData.emplace_back(QPointF(0, 0));
402         obsDeltaQData.reserve(observedBins);
403
404         QVector<QPointF> obsLowerBoundData;
405         obsLowerBoundData.emplace_back(QPointF(0, 0));
406         obsLowerBoundData.reserve(observedBins);
407
408         QVector<QPointF> obsUpperBoundData;
409         obsUpperBoundData.emplace_back(QPointF(0, 0));
410         obsUpperBoundData.reserve(observedBins);
411
412         QVector<QPointF> obsMeanData;
413         obsMeanData.emplace_back(QPointF(0, 0));
414         obsMeanData.reserve(observedBins);
415
416         for (int i = 0; i < observedBins; ++i) {
417             double x = observedBinWidth * (i + 1);
418             obsDeltaQData.emplace_back(QPointF(x, obsDeltaQ.cdfAt(i)))
     ;
419             obsLowerBoundData.emplace_back(QPointF(x, obsBounds[i].
     lowerBound));
420             obsUpperBoundData.emplace_back(QPointF(x, obsBounds[i].
     upperBound));
421             obsMeanData.emplace_back(QPointF(x, obsBounds[i].mean));
422         }
423
424         QVector<QPointF> calcDeltaQData;
425         calcDeltaQData.emplace_back(QPointF(0, 0));
426         calcDeltaQData.reserve(observedBins);
427
428         QVector<QPointF> calcLowerBoundData;
429         calcLowerBoundData.emplace_back(QPointF(0, 0));
430         calcLowerBoundData.reserve(observedBins);
431
432         QVector<QPointF> calcUpperBoundData;
433         calcUpperBoundData.emplace_back(QPointF(0, 0));
434         calcUpperBoundData.reserve(observedBins);
435
436         QVector<QPointF> calcMeanData;
437         calcMeanData.emplace_back(QPointF(0, 0));
438         calcMeanData.reserve(observedBins);
```

```
439
440        QVector < QPointF > qtaData ;
441
442        // Prepare calculatedDeltaQ data
443        int calculatedBins = calcDeltaQ . getBins ();
444        double calculatedBinWidth = calcDeltaQ . getBinWidth ();
445        for ( int i = 0; i < calculatedBins ; ++i) {
446            double x = calculatedBinWidth * ( i + 1);
447            calcDeltaQData . emplace_back ( QPointF (x , calcDeltaQ . cdfAt (i )
    ));
448            calcLowerBoundData . emplace_back ( QPointF (x , calcBounds [i ].
    lowerBound ));
449            calcUpperBoundData . emplace_back ( QPointF (x , calcBounds [i ].
    upperBound ));
450            calcMeanData . emplace_back ( QPointF (x , calcBounds [i ]. mean ));
451        }
452        if ( qta . defined ) {
453            qtaData . reserve (8);
454            qtaData . emplace_back ( qta . perc_25 , 0);
455            qtaData . emplace_back ( qta . perc_25 , 0.25);
456            qtaData . emplace_back ( qta . perc_50 , 0.25);
457            qtaData . emplace_back ( qta . perc_50 , 0.5);
458            qtaData . emplace_back ( qta . perc_75 , 0.5);
459            qtaData . emplace_back ( qta . perc_75 , 0.75);
460            qtaData . emplace_back ( maxDelay , 0.75);
461            qtaData . emplace_back ( maxDelay , qta . cdfMax );
462        }
463        auto computeEnd = high_resolution_clock :: now ();
464        //   std :: cout << "comp ," << observedBins << "," <<
    calculatedBins << "," << duration_cast < microseconds >( computeEnd -
    computeStart ). count ();
465
466        // --- Push results back to GUI thread ---
467
468        QMetaObject :: invokeMethod (
469            plot ,
470            [=]() {
471                auto guiStart = high_resolution_clock :: now ();
472
473                plot ->setUpdatesEnabled (false );
474
475                plot ->updateSeries ( series . obsS , obsDeltaQData );
476                plot ->updateSeries ( series . obsLowerBoundS ,
    obsLowerBoundData );
477                plot ->updateSeries ( series . obsUpperBoundS ,
    obsUpperBoundData );
478                plot ->updateSeries ( series . obsMeanS , obsMeanData );
479
480                plot ->updateSeries ( series . calcS , calcDeltaQData );
481                plot ->updateSeries ( series . calcLowerBoundS ,
    calcLowerBoundData );
482                plot ->updateSeries ( series . calcUpperBoundS ,
    calcUpperBoundData );
483                plot ->updateSeries ( series . calcMeanS , calcMeanData );
484                plot ->updateSeries ( series . qtaS , qtaData );
485
```

```
486                plot->setUpdatesEnabled(true);
487
488                auto guiEnd = high_resolution_clock::now();
489                // std::cout << "gui," << observedBins << "," <<
      calculatedBins << "," << duration_cast<microseconds>(guiEnd -
      guiStart).count();
490            },
491            Qt::QueuedConnection);
492    });
493    ret.waitForFinished();
494 }
```

C.2.5   DQPlotList.cpp

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

```
1  #include "DQPlotList.h"
2  #include "../Application.h"
3  #include <QLabel>
4  #include <QListWidgetItem>
5  #include <qlistwidget.h>
6  DQPlotList::DQPlotList(DQPlotController *controller, QWidget *parent)
7      : QWidget(parent)
8      , controller(controller)
9  {
10     QVBoxLayout *layout = new QVBoxLayout(this);
11
12     // Selected items list
13     QLabel *selectedLabel = new QLabel("Selected probes:");
14     selectedList = new QListWidget(this);
15     layout->addWidget(selectedLabel);
16     layout->addWidget(selectedList);
17     selectedList->setSelectionMode(QAbstractItemView::MultiSelection);
18     removeButton = new QPushButton("Remove selected probes", this);
19     layout->addWidget(removeButton);
20
21     connect(removeButton, &QPushButton::clicked, this, &DQPlotList::
      onRemoveSelection);
22     // Available items list
23     QLabel *availableLabel = new QLabel("Available probes:");
24     availableList = new QListWidget(this);
25     layout->addWidget(availableLabel);
26     layout->addWidget(availableList);
27     availableList->setSelectionMode(QAbstractItemView::MultiSelection)
      ;
28     // Confirm button
29     addButton = new QPushButton("Add selected probes", this);
30     connect(addButton, &QPushButton::clicked, this, &DQPlotList::
      onConfirmSelection);
31     layout->addWidget(addButton);
32
33     updateLists();
34 }
35
36 bool DQPlotList::isEmptyAfterReset()
```

```
37  {
38      return controller ->isEmptyAfterReset ();
39  }
40
41
42  void DQPlotList :: updateLists ()
43  {
44      availableList ->clear ();
45      selectedList ->clear ();
46
47      auto plotComponents = controller ->getComponents ();
48      auto system = Application :: getInstance ().getSystem ();
49      // Add components selected in a DeltaQPlot
50      for (auto &component : plotComponents) {
51          new QListWidgetItem(QString :: fromStdString (component),
    selectedList );
52      }
53
54      // Select all components that are available to be chosen to add
55      auto allComponents = system ->getAllComponentsName ();
56
57      auto pred
58          = [& plotComponents ]( const std :: string &key) -> bool { return
    std :: find(plotComponents.begin (), plotComponents.end (), key) !=
    plotComponents.end (); };
59
60      allComponents.erase(std :: remove_if(allComponents.begin (),
    allComponents.end (), pred), allComponents.end ());
61      for (auto &component : allComponents) {
62          new QListWidgetItem(QString :: fromStdString (component),
    availableList );
63      }
64  }
65
66  void DQPlotList :: onConfirmSelection ()
67  {
68      QList <QListWidgetItem *> selected = availableList ->selectedItems ()
    ;
69      auto system = Application :: getInstance ().getSystem ();
70      for (QListWidgetItem *item : selected) {
71          controller ->addComponent(item ->text ().toStdString (), system ->
    hasOutcome(item ->text ().toStdString ()));
72      }
73      updateLists ();
74  }
75
76  void DQPlotList :: onRemoveSelection ()
77  {
78      QList <QListWidgetItem *> selected = selectedList ->selectedItems ();
79
80      for (QListWidgetItem *item : selected) {
81          controller ->removeComponent(item ->text ().toStdString ());
82      }
83      updateLists ();
84  }
```

C.2.6   DelaySettingsWidget.cpp

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

```cpp
/**
 * @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);
    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(); });
```

```cpp
48  }
49
50  /**
51   * @brief Populates the combo box with available observables from the
       system.
52   */
53  void DelaySettingsWidget::populateComboBox()
54  {
55      auto system = Application::getInstance().getSystem();
56      if (!system)
57          return;
58
59      observableComboBox->clear();
60      for (const auto &[name, obs] : system->getObservables()) {
61          if (obs)
62              observableComboBox->addItem(QString::fromStdString(name));
63      }
64  }
65
66  /**
67   * @brief Loads delay settings for the currently selected observable.
68   */
69  void DelaySettingsWidget::loadObservableSettings()
70  {
71      auto system = Application::getInstance().getSystem();
72      if (!system)
73          return;
74
75      QString observableName = observableComboBox->currentText();
76      if (observableName.isEmpty())
77          return;
78
79      auto observable = system->getObservable(observableName.toStdString
       ());
80
81      auto exponent = observable->getDeltaTExp();
82      auto bins = observable->getNBins();
83      delaySlider->setValue(exponent);
84      binSpinBox->setValue(bins);
85
86      updateMaxDelay();
87  }
88
89  /**
90   * @brief Computes the current maximum delay.
91   * @return Maximum delay value based on bin count and delay exponent.
92   */
93  double DelaySettingsWidget::getMaxDelayMs() const
94  {
95      int exponent = delaySlider->value();
96      int bins = binSpinBox->value();
97      return 1.0 * std::pow(2.0, exponent) * bins;
98  }
99
100 /**
101  * @brief Updates the label that shows the computed max delay value.
```

```
102  */
103 void DelaySettingsWidget::updateMaxDelay()
104 {
105     double delay = getMaxDelayMs();
106     maxDelayLabel->setText(QString("Max delay is: %1 ms").arg(delay,
     0, 'f', 2));
107 }
108
109 /**
110  * @brief Saves the current delay settings to the system and emits a
     change signal.
111  */
112 void DelaySettingsWidget::onSaveDelayClicked()
113 {
114     auto system = Application::getInstance().getSystem();
115     if (!system)
116         return;
117
118     QString name = observableComboBox->currentText();
119     if (name.isEmpty())
120         return;
121
122     int exponent = delaySlider->value();
123     int bins = binSpinBox->value();
124     std::string nameString = name.toStdString();
125     system->setObservableParameters(nameString, exponent, bins);
126
127     Q_EMIT delayParametersChanged();
128 }
```

C.2.7 DeltaQPlot.cpp

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

```
1
2 #include "DeltaQPlot.h"
3 #include "ColorRegistry.h"
4 #include "CustomLegendPanel.h"
5 #include "DQPlotList.h"
6 #include <QChartView>
7 #include <QDebug>
8 #include <QHBoxLayout>
9 #include <QLineSeries>
10 #include <QMouseEvent>
11 #include <QRandomGenerator>
12 #include <QToolButton>
13 #include <QVBoxLayout>
14 DeltaQPlot::DeltaQPlot(const std::vector<std::string> &selectedItems,
     QWidget *parent)
15     : QWidget(parent)
16 {
17
18     // Create legend panel and toggle
19     legendPanel = new CustomLegendPanel(this);
20     QToolButton *toggleButton = new QToolButton(this);
```

```
21      toggleButton ->setText(">  Legend");
22      toggleButton ->setCheckable(true);
23      toggleButton ->setChecked(true);
24      toggleButton ->setToolButtonStyle(Qt::ToolButtonTextBesideIcon);
25
26      connect(toggleButton, &QToolButton::toggled, legendPanel, &QWidget
        ::setVisible);
27      connect(toggleButton, &QToolButton::toggled, [toggleButton](bool
        checked) { toggleButton ->setText(checked ? "^ Legend" : ">  Legend"
        ); });
28
29      // Create chart and view
30      chart = new QChart();
31      chartView = new QChartView(chart, this);
32      chartView ->setRenderHint(QPainter::Antialiasing);
33      chart ->legend()->setVisible(false);
34
35      axisX = new QValueAxis();
36      axisY = new QValueAxis();
37      axisY ->setRange(-0.01, 1.0);
38      axisX ->setRange(0, 0.05);
39      chart ->addAxis(axisX, Qt::AlignBottom);
40      chart ->addAxis(axisY, Qt::AlignLeft);
41
42      controller = new DQPlotController(this, selectedItems);
43      plotList = new DQPlotList(controller, this);
44      // Right-side layout: toggle + legend
45      QVBoxLayout *rightLayout = new QVBoxLayout();
46      rightLayout ->addWidget(toggleButton);
47      rightLayout ->addWidget(legendPanel);
48      rightLayout ->addStretch();
49
50      // Main layout: chart + right side
51      QHBoxLayout *mainLayout = new QHBoxLayout(this);
52      mainLayout ->addWidget(chartView, 1);
53      mainLayout ->addLayout(rightLayout);
54      mainLayout ->setContentsMargins(0, 0, 0, 0);
55      mainLayout ->setSpacing(5);
56      setLayout(mainLayout);
57      chartView ->setRenderHint(QPainter::Antialiasing);
58 }
59
60 DeltaQPlot::~DeltaQPlot()
61 {
62      delete controller;
63      controller = nullptr;
64      delete plotList;
65      plotList = nullptr;
66 }
67
68 bool DeltaQPlot::isEmptyAfterReset()
69 {
70      if (!controller ->isEmptyAfterReset()) {
71          plotList ->updateLists();
72          return false;
73      }
```

```
74      return true;
75  }
76
77  void DeltaQPlot::setTitle(QString &&title)
78  {
79      chart->setTitle(title);
80  }
81
82  void DeltaQPlot::addSeries(QLineSeries *series, const std::string &
        name)
83  {
84      chart->addSeries(series);
85      series->setName(QString::fromStdString(name));
86      series->attachAxis(axisX);
87      series->attachAxis(axisY);
88      QColor color = ColorRegistry::getColorFor(name);
89      legendPanel->addEntry(QString::fromStdString(name), color);
90      series->setColor(color);
91      series->setVisible(true);
92  }
93
94  void DeltaQPlot::update(uint64_t timeLowerBound, uint64_t
        timeUpperBound)
95  {
96      controller->update(timeLowerBound, timeUpperBound);
97  }
98
99  void DeltaQPlot::removeSeries(QAbstractSeries *series)
100 {
101     chart->removeSeries(series);
102     auto name = series->name();
103     legendPanel->removeEntry(name);
104 }
105
106 void DeltaQPlot::editPlot(const std::vector<std::string> &
        selectedItems)
107 {
108     controller->editPlot(selectedItems);
109 }
110
111 void DeltaQPlot::updateSeries(QLineSeries *series, const QVector<
        QPointF> &data)
112 {
113     series->replace(data);
114 }
115
116 void DeltaQPlot::updateXRange(double xRange)
117 {
118     axisX->setRange(0, xRange);
119 }
120
121 std::vector<std::string> DeltaQPlot::getComponents()
122 {
123     return controller->getComponents();
124 }
125
```

```
126  DQPlotList *DeltaQPlot::getPlotList()
127  {
128      return plotList;
129  }
130
131  void DeltaQPlot::mousePressEvent(QMouseEvent *event)
132  {
133      Q_EMIT plotSelected(this);
134  }
```

C.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.

```
1   #include "MainWindow.h"
2   #include "../Application.h"
3   #include "../maths/DeltaQOperations.h"
4   #include "DQPlotList.h"
5   #include "DeltaQPlot.h"
6   #include "NewPlotList.h"
7   #include "ObservableSettings.h"
8   #include "Sidebar.h"
9   #include <QMenu>
10  #include <QMessageBox>
11  #include <QThread>
12  #include <QVBoxLayout>
13  #define MAX_P_ROW 2
14  #define MAX_P_COL 2
15
16  MainWindow::MainWindow(QWidget *parent)
17      : QMainWindow(parent)
18  {
19      // Set up central widget and main layout
20      centralWidget = new QWidget(this);
21      setCentralWidget(centralWidget);
22      mainLayout = new QHBoxLayout(centralWidget);
23
24      // Configure scroll area for plots
25      scrollArea = new QScrollArea(this);
26      scrollArea->setWidgetResizable(true);
27      scrollArea->setHorizontalScrollBarPolicy(Qt::ScrollBarAsNeeded);
28      scrollArea->setVerticalScrollBarPolicy(Qt::ScrollBarAsNeeded);
29      scrollArea->setBaseSize(800, 800);
30
31      // Set up plot container with grid layout
32      plotContainer = new QWidget();
33      plotLayout = new QGridLayout(plotContainer);
34      scrollArea->setWidget(plotContainer);
35      mainLayout->addWidget(scrollArea, 1);
36
37      // Initialize side panels
38      sidebar = new Sidebar(this);
39      triggersTab = new TriggersTab(this);
40      observableSettings = new ObservableSettings(this);
```

```
41      stubWidget = new StubControlWidget(this);
42
43      // Configure tabbed side panel
44      sideTabWidget = new QTabWidget(this);
45      sideTabWidget->addTab(sidebar, "System/Plots");
46      sideTabWidget->addTab(observableSettings, "Probes settings");
47      sideTabWidget->addTab(triggersTab, "Triggers");
48      sideTabWidget->addTab(stubWidget, "Connection controls");
49      sideTabWidget->setTabPosition(QTabWidget::West);
50      // Connect sidebar signals
51      connect(sidebar, &Sidebar::addPlotClicked, this, &MainWindow::
        onAddPlotClicked);
52
53      // Set up side container layout
54      sideContainer = new QWidget(this);
55      sideLayout = new QVBoxLayout(sideContainer);
56      sideLayout->addWidget(sideTabWidget);
57      sideLayout->setStretch(1, 0); // Make tabs take up most space
58      mainLayout->addWidget(sideContainer, 0);
59
60      // Set up update timer in separate thread
61      timerThread = new QThread(this);
62      updateTimer = new QTimer();
63      updateTimer->moveToThread(timerThread);
64      connect(updateTimer, &QTimer::timeout, this, &MainWindow::
        updatePlots, Qt::QueuedConnection);
65      connect(timerThread, &QThread::started, [this]() { updateTimer->
        start(200); });
66      timerThread->start();
67
68      // Register system reset observer
69      Application::getInstance().addObserver([this] { this->reset(); });
70
71      // Initialize time bounds
72      auto now = std::chrono::system_clock::now();
73      auto adjustedTime = now - std::chrono::milliseconds(200);
74      timeLowerBound = std::chrono::duration_cast<std::chrono::
        nanoseconds>(adjustedTime.time_since_epoch()).count();
75
76      // Connect polling rate changes
77      connect(sidebar, &Sidebar::onSamplingRateChanged, this, [this](int
        ms) {
78          qDebug() << "MainWindow received polling rate:" << ms;
79          pollingRate = ms;
80          QMetaObject::invokeMethod(updateTimer, [ms, this]() {
        updateTimer->setInterval(ms); }, Qt::QueuedConnection);
81      });
82  }
83
84  /**
85   * @brief Cleans up empty plots during reset.
86   */
87  void MainWindow::reset()
88  {
89      std::lock_guard<std::mutex> lock(plotDelMutex);
90      auto it = plotContainers.begin();
```

```
 91      while (it != plotContainers.end()) {
 92          DeltaQPlot *plot = it.key();
 93          QWidget *plotWidget = it.value();
 94
 95          if (plot->isEmptyAfterReset()) {
 96              it = plotContainers.erase(it);
 97              delete plot;
 98              if (plotWidget) {
 99                  delete plotWidget;
100              }
101              sidebar->hideCurrentPlot();
102          } else {
103              ++it;
104          }
105      }
106 }
107
108 /**
109  * @brief Updates all plots with new data from the system.
110  */
111 void MainWindow::updatePlots()
112 {
113     // Update time bounds
114     timeLowerBound += std::chrono::duration_cast<std::chrono::
    nanoseconds>(std::chrono::milliseconds(pollingRate)).count();
115     uint64_t timeUpperBound = timeLowerBound + std::chrono::
    duration_cast<std::chrono::nanoseconds>(std::chrono::milliseconds(
    pollingRate)).count();
116
117     auto system = Application::getInstance().getSystem();
118     std::lock_guard<std::mutex> lock(plotDelMutex);
119
120     // Update probes
121     for (auto &[name, probe] : system->getProbes()) {
122         if (probe) {
123             probe->getObservedDeltaQ(timeLowerBound, timeUpperBound);
124             probe->calculateCalculatedDeltaQ(timeLowerBound,
    timeUpperBound);
125         }
126     }
127
128     // Update operators
129     for (auto &[name, op] : system->getOperators()) {
130         if (op) {
131             op->getObservedDeltaQ(timeLowerBound, timeUpperBound);
132             op->calculateCalculatedDeltaQ(timeLowerBound,
    timeUpperBound);
133         }
134     }
135
136     // Update outcomes
137     for (auto &[name, outcome] : system->getOutcomes()) {
138         if (outcome) {
139             outcome->getObservedDeltaQ(timeLowerBound, timeUpperBound)
    ;
140         }
```

```
141         }
142
143         // Update all plots
144         for (auto [plot, _] : plotContainers.asKeyValueRange()) {
145             plot->update(timeLowerBound, timeUpperBound);
146         }
147 }
148
149 /**
150  * @brief Destructor cleans up timer thread and resources.
151  */
152 MainWindow::~MainWindow()
153 {
154     if (timerThread) {
155         timerThread->quit();
156         timerThread->wait();
157         delete timerThread;
158     }
159 }
160
161 /**
162  * @brief Adds a new plot based on sidebar selection.
163  */
164 void MainWindow::onAddPlotClicked()
165 {
166     auto selectedItems = sidebar->getPlotList()->getSelectedItems();
167
168     if (selectedItems.empty()) {
169         QMessageBox::warning(this, "No Selection", "Please select
    components before adding a plot.");
170         return;
171     }
172
173     // Create new plot container
174     auto *plotWidget = new QWidget(this);
175     auto *plotWidgetLayout = new QVBoxLayout(plotWidget);
176     auto *deltaQPlot = new DeltaQPlot(selectedItems, this);
177
178     // Set up connections and tracking
179     connect(deltaQPlot, &DeltaQPlot::plotSelected, this, &MainWindow::
    onPlotSelected);
180     plotContainers[deltaQPlot] = plotWidget;
181
182     // Configure layout
183     plotWidgetLayout->addWidget(deltaQPlot);
184     plotWidget->setMaximumWidth(scrollArea->width() / MAX_P_ROW);
185     plotWidget->setMaximumHeight(scrollArea->height() / 2);
186
187     // Position in grid
188     int plotCount = plotContainers.size();
189     int row = (plotCount - 1) / MAX_P_ROW;
190     int col = (plotCount - 1) % MAX_P_COL;
191     plotLayout->addWidget(plotWidget, row, col);
192
193     // Update UI state
194     onPlotSelected(deltaQPlot);
```

```
195        sidebar -> clearOnAdd ();
196 }
197
198 /**
199  * @brief Handles window resize events to adjust plot sizes.
200  * @param event The resize event.
201  */
202 void MainWindow :: resizeEvent ( QResizeEvent * event )
203 {
204     QMainWindow :: resizeEvent ( event );
205     for ( auto [plot , widget] : plotContainers . asKeyValueRange ()) {
206         widget -> setMaximumWidth ( scrollArea -> width () / MAX_P_COL );
207         widget -> setMaximumHeight ( scrollArea -> height () / 2);
208     }
209 }
210
211 /**
212  * @brief Shows context menu for plot management.
213  * @param event The context menu event.
214  */
215 void MainWindow :: contextMenuEvent ( QContextMenuEvent * event )
216 {
217     QWidget * child = childAt ( event -> pos ());
218     if (! child )
219         return ;
220
221     // Find which plot was right - clicked
222     DeltaQPlot * selectedPlot = nullptr ;
223     for ( auto it = plotContainers . begin (); it != plotContainers . end ();
       ++it) {
224         if (it . value () -> isAncestorOf ( child )) {
225             selectedPlot = it . key ();
226             break ;
227         }
228     }
229
230     if (! selectedPlot )
231         return ;
232
233     // Create and show context menu
234     QMenu contextMenu ( this );
235     QAction * removeAction = contextMenu . addAction ( "Remove Plot ");
236
237     QAction * selectedAction = contextMenu . exec ( event -> globalPos ());
238
239     if ( selectedAction == removeAction ) {
240         onRemovePlot ( selectedPlot );
241     }
242 }
243
244 /**
245  * @brief Removes a plot and cleans up resources.
246  * @param plot The plot to remove.
247  */
248 void MainWindow :: onRemovePlot ( DeltaQPlot * plot )
249 {
```

```
250     QWidget *plotWidget = plotContainers.value(plot, nullptr);
251     if (plotWidget) {
252         plotLayout->removeWidget(plotWidget);
253         plotWidget->deleteLater();
254     }
255     plotContainers.remove(plot);
256
257     // Reorganize remaining plots
258     int plotCount = 0;
259     for (auto it = plotContainers.begin(); it != plotContainers.end();
        ++it) {
260         int row = plotCount / MAX_P_ROW;
261         int col = plotCount % MAX_P_COL;
262         plotLayout->addWidget(it.value(), row, col);
263         plotLayout->setRowStretch(row, 1);
264         plotLayout->setColumnStretch(col, 1);
265         plotCount++;
266     }
267
268     sidebar->hideCurrentPlot();
269 }
270
271 /**
272  * @brief Updates sidebar when a plot is selected.
273  * @param plot The newly selected plot.
274  */
275 void MainWindow::onPlotSelected(DeltaQPlot *plot)
276 {
277     if (!plot)
278         return;
279     DQPlotList *plotList = plot->getPlotList();
280     if (!plotList) {
281         return;
282     }
283     sidebar->setCurrentPlotList(plotList);
284 }
```

C.2.9   NewPlotList.cpp

This widget is the widget to add a new plot to the plots panel.

```
1 /**
2  * @file NewPlotList.cpp
3  * @brief Implementation of the NewPlotList class, which provides a UI
      list for selecting observables to create a new plot.
4  */
5
6 #include "NewPlotList.h"
7 #include "../Application.h"
8 #include <iostream>
9 #include <qlistwidget.h>
10
11 NewPlotList::NewPlotList(QWidget *parent)
12     : QListWidget(parent)
13 {
```

```
14         setSelectionMode(QAbstractItemView::MultiSelection);
15         Application::getInstance().addObserver([this]() { this->reset();
     });
16  }
17
18  /**
19   * @brief Clears and repopulates the list with updated observables.
20   *
21   */
22  void NewPlotList::reset()
23  {
24      this->blockSignals(true);
25      while (count() != 0)
26          delete takeItem(0);
27      this->blockSignals(false);
28      addItems();
29  }
30
31  /**
32   * @brief Adds all observables from the system as items in the list.
33   */
34  void NewPlotList::addItems()
35  {
36      auto system = Application::getInstance().getSystem();
37      auto observables = system->getObservables();
38
39      for (const auto &obs : observables) {
40          if (obs.second) {
41              QListWidgetItem *item = new QListWidgetItem(QString::
     fromStdString(obs.first), this);
42              item->setFlags(Qt::ItemIsSelectable | Qt::ItemIsEnabled);
43          }
44      }
45  }
46
47  /**
48   * @brief Returns a list of selected observable names.
49   * @return A vector of strings corresponding to selected item labels.
50   */
51  std::vector<std::string> NewPlotList::getSelectedItems()
52  {
53      std::vector<std::string> selectedItems;
54      QList<QListWidgetItem *> selected = this->selectedItems();
55
56      for (QListWidgetItem *item : selected) {
57          selectedItems.push_back(item->text().toStdString());
58      }
59      return selectedItems;
60  }
61
62  /**
63   * @brief Deselects all currently selected items in the list.
64   */
65  void NewPlotList::deselectAll()
66  {
67      clearSelection();
```

```
68 }
```

C.2.10   ObservableSettings.cpp

This widget is a tab that contains the settings for the probes (Setting a QTA, setting
dMax).

```cpp
1  #include "ObservableSettings.h"
2  #include <qlabel.h>
3  #include <qwidget.h>
4
5  ObservableSettings::ObservableSettings(QWidget *parent)
6      : QWidget(parent)
7  {
8      layout = new QVBoxLayout(this);
9      layout->setAlignment(Qt::AlignTop);
10     layout->setSpacing(10);
11     layout->setContentsMargins(10, 10, 10, 10);
12
13     qtaInputWidget = new QTAInputWidget(this);
14     layout->addWidget(qtaInputWidget);
15
16     delaySettingsWidget = new DelaySettingsWidget(this);
17     layout->addWidget(delaySettingsWidget);
18     connect(delaySettingsWidget, &DelaySettingsWidget::
       delayParametersChanged, qtaInputWidget, &QTAInputWidget::
       loadObservableSettings);
19 }
```

C.2.11   SamplingRateWidget.cpp

This widget allows the sampling rate to be changed via a slider.

```cpp
1  #include "SamplingRateWidget.h"
2
3  SamplingRateWidget::SamplingRateWidget(QWidget *parent)
4      : QWidget(parent)
5  {
6      // Available polling rate options in milliseconds
7      pollingRates = {100, 200, 300, 400, 500, 600, 700, 800, 900,
8                      1000, 2000, 5000, 10000};
9
10     // Configure rate selection slider
11     slider = new QSlider(Qt::Horizontal);
12     slider->setMinimum(0);
13     slider->setMaximum(pollingRates.size() - 1);
14     slider->setValue(1);
15     slider->setTickInterval(1);
16     slider->setTickPosition(QSlider::TicksBelow);
17
18     valueLabel = new QLabel(QString("Polling Rate: %1 ms").arg(
       pollingRates[0]));
19
20     saveButton = new QPushButton("Save polling rate");
```

```cpp
21
22     // Set up layout
23     auto *layout = new QVBoxLayout();
24     layout->addWidget(slider);
25     layout->addWidget(valueLabel);
26     layout->addWidget(saveButton);
27     setLayout(layout);
28
29     // Connect signals
30     connect(slider, &QSlider::valueChanged,
31             this, &SamplingRateWidget::onSliderValueChanged);
32     connect(saveButton, &QPushButton::clicked,
33             this, &SamplingRateWidget::onSaveClicked);
34 }
35
36 /**
37  * @brief Updates the displayed rate when slider changes.
38  * @param value The current slider index (0-based).
39  *
40  * Converts slider index to actual milliseconds and updates the
      display label.
41  */
42 void SamplingRateWidget::onSliderValueChanged(int value)
43 {
44     int ms = pollingRates[value];
45     valueLabel->setText(QString("Polling Rate: %1 ms").arg(ms));
46 }
47
48 /**
49  * @brief Emits the selected polling rate when save is clicked.
50  *
51  * Gets the current slider value, converts it to milliseconds,
52  * and emits the onPollingRateChanged signal.
53  */
54 void SamplingRateWidget::onSaveClicked()
55 {
56     int ms = pollingRates[slider->value()];
57     Q_EMIT onSamplingRateChanged(ms);
58 }
```

C.2.12   QTAInputWidget.cpp

This widget allows the QTA to be set for a probe

```cpp
1 #include "QTAInputWidget.h"
2 #include "../Application.h"
3 #include <QDoubleValidator>
4 #include <QMessageBox>
5 #include <QPushButton>
6
7 QTAInputWidget::QTAInputWidget(QWidget *parent)
8     : QWidget(parent)
9 {
10     auto layout = new QFormLayout(this);
11
```

```
12      qtaLabel = new QLabel(this);
13      qtaLabel->setText("Set QTA for a probe:");
14      layout->addRow(qtaLabel);
15      observableComboBox = new QComboBox(this);
16      layout->addRow("Probe:", observableComboBox);
17
18      perc25Edit = new QLineEdit(this);
19      perc25Edit->setPlaceholderText("Seconds (s)");
20
21      perc50Edit = new QLineEdit(this);
22      perc50Edit->setPlaceholderText("Seconds (s)");
23
24      perc75Edit = new QLineEdit(this);
25      perc75Edit->setPlaceholderText("Seconds (s)");
26
27      cdfMaxEdit = new QLineEdit(this);
28      cdfMaxEdit->setPlaceholderText("Value between 0 and 1");
29
30      layout->addRow("25th Percentile (s):", perc25Edit);
31      layout->addRow("50th Percentile (s):", perc50Edit);
32      layout->addRow("75th Percentile (s):", perc75Edit);
33      layout->addRow("Max. allowed failure (0-1):", cdfMaxEdit);
34
35      connect(observableComboBox, &QComboBox::currentTextChanged, this,
        &QTAInputWidget::loadObservableSettings);
36
37      Application::getInstance().addObserver([this]() { this->
        populateComboBox(); });
38      saveButton = new QPushButton("Save QTA Settings", this);
39      layout->addRow(saveButton);
40      connect(saveButton, &QPushButton::clicked, this, &QTAInputWidget::
        onSaveButtonClicked);
41
42      setLayout(layout);
43
44      Application::getInstance().addObserver([this]() { this->
        populateComboBox(); });
45  }
46
47  void QTAInputWidget::populateComboBox()
48  {
49      auto system = Application::getInstance().getSystem();
50      if (!system)
51          return;
52
53      observableComboBox->clear();
54      for (const auto &[name, _] : system->getProbes()) {
55          observableComboBox->addItem(QString::fromStdString(name));
56      }
57      for (const auto &[name, _] : system->getOutcomes()) {
58          observableComboBox->addItem(QString::fromStdString(name));
59      }
60  }
61
62  void QTAInputWidget::loadObservableSettings()
63  {
```

```cpp
64        auto system = Application::getInstance().getSystem();
65        if (!system)
66            return;
67        std::string observableName = observableComboBox->currentText().
      toStdString();
68        auto observable = system->getObservable(observableName);
69        if (observable) {
70            auto qta = observable->getQTA();
71            perc25Edit->setText(QString::number(qta.perc_25, 'f', 6)); //
      6 decimal places
72            perc50Edit->setText(QString::number(qta.perc_50, 'f', 6));
73            perc75Edit->setText(QString::number(qta.perc_75, 'f', 6));
74            cdfMaxEdit->setText(QString::number(qta.cdfMax, 'f', 6));
75        }
76 }
77
78 void QTAInputWidget::onSaveButtonClicked()
79 {
80        auto system = Application::getInstance().getSystem();
81        if (!system)
82            return;
83
84        std::string observableName = observableComboBox->currentText().
      toStdString();
85        auto observable = system->getObservable(observableName);
86        if (!observable)
87            return;
88
89        try {
90            QTA newQTA = QTA::create(getPerc25(), getPerc50(), getPerc75()
      , getCdfMax(), true);
91            observable->setQTA(newQTA);
92        } catch (std::exception &e) {
93            QMessageBox::warning(this, "Error", e.what());
94        }
95 }
96
97 double QTAInputWidget::getPerc25() const
98 {
99        return perc25Edit->text().toDouble();
100 }
101 double QTAInputWidget::getPerc50() const
102 {
103        return perc50Edit->text().toDouble();
104 }
105 double QTAInputWidget::getPerc75() const
106 {
107        return perc75Edit->text().toDouble();
108 }
109 double QTAInputWidget::getCdfMax() const
110 {
111        return cdfMaxEdit->text().toDouble();
112 }
113
114 QString QTAInputWidget::getSelectedObservable() const
115 {
```

```
116     return observableComboBox ->currentText ();
117 }
```

C.2.13   Sidebar.cpp

This widget is a tab where the user can handle the system, add/remove plots and
change the sampling rate.

```cpp
1  #include "Sidebar.h"
2
3  #include "NewPlotList.h"
4  #include "SamplingRateWidget.h"
5  #include "SystemCreationWidget.h"
6  #include <QBoxLayout >
7  #include <QFileDialog >
8  #include <QLabel >
9  #include <QMessageBox >
10 #include <iostream >
11 #include <qboxlayout.h>
12 #include <qlabel.h>
13 #include <qlogging.h>
14 #include <qnamespace.h>
15 #include <qpushbutton.h>
16 #include <qsplitter.h>
17 #include <qtextedit.h>
18
19 Sidebar ::Sidebar (QWidget *parent)
20     : QWidget (parent)
21 {
22     // Main layout setup
23     layout = new QVBoxLayout (this);
24     layout ->setContentsMargins (0, 0, 0, 0);
25
26     mainSplitter = new QSplitter (Qt::Vertical , this);
27
28     // System creation section
29     systemCreationWidget = new SystemCreationWidget (this);
30     mainSplitter ->addWidget (systemCreationWidget );
31
32     // New plot section
33     newPlotListWidget = new QWidget (this);
34     newPlotListLayout = new QVBoxLayout (newPlotListWidget );
35     newPlotListLayout ->setContentsMargins (5, 5, 5, 5);
36
37     newPlotLabel = new QLabel ("Select probes for a new plot:", this);
38     newPlotLabel ->setSizePolicy (QSizePolicy ::Preferred , QSizePolicy ::
    Fixed);
39     addNewPlotButton = new QPushButton ("Add plot");
40     newPlotList = new NewPlotList (this);
41     connect (addNewPlotButton , &QPushButton ::clicked , this, &Sidebar ::
    onAddPlotClicked );
42
43     newPlotListLayout ->addWidget (newPlotLabel );
44     newPlotListLayout ->addWidget (newPlotList );
45     newPlotListLayout ->addWidget (addNewPlotButton );
```

```cpp
46
47      mainSplitter->addWidget(newPlotListWidget);
48
49      // Current plot section (initially hidden)
50      currentPlotWidget = new QWidget(this);
51      currentPlotLayout = new QVBoxLayout(currentPlotWidget);
52      currentPlotLabel = new QLabel("Modify current plot:", this);
53      currentPlotLabel->setSizePolicy(QSizePolicy::Preferred,
    QSizePolicy::Fixed);
54      currentPlotLabel->hide();
55
56      currentPlotLayout->addWidget(currentPlotLabel);
57      mainSplitter->addWidget(currentPlotWidget);
58
59      // Polling rate section
60      pollingRateWidget = new SamplingRateWidget(this);
61      mainSplitter->addWidget(pollingRateWidget);
62
63      connect(pollingRateWidget, &SamplingRateWidget::
    onSamplingRateChanged, this, &Sidebar::handleSamplingRateChanged);
64      layout->addWidget(mainSplitter);
65  }
66
67  /**
68   * @brief Sets the current plot list widget to display.
69   * @param plotList The DQPlotList widget to show in the current plot
    section.
70   */
71  void Sidebar::setCurrentPlotList(DQPlotList *plotList)
72  {
73      if (currentPlotList == plotList) {
74          return;
75      }
76
77      if (currentPlotList) {
78          layout->removeWidget(currentPlotList);
79          currentPlotList->hide();
80      }
81
82      if (plotList) {
83          currentPlotList = plotList;
84          layout->addWidget(currentPlotList);
85          currentPlotList->show();
86          currentPlotLabel->show();
87      }
88  }
89
90  /**
91   * @brief Handles polling rate change events from the
    PollingRateWidget.
92   * @param ms The new polling rate in milliseconds.
93   */
94  void Sidebar::handleSamplingRateChanged(int ms)
95  {
96      Q_EMIT onSamplingRateChanged(ms);
97  }
```

```
98
99  /**
100  * @brief Hides the current plot management section.
101  */
102 void Sidebar::hideCurrentPlot()
103 {
104     if (currentPlotList) {
105         layout->removeWidget(currentPlotList);
106         currentPlotList = nullptr;
107     }
108     currentPlotLabel->hide();
109 }
110
111 /**
112  * @brief Clears new plot selection after plot creation.
113  */
114 void Sidebar::clearOnAdd()
115 {
116     newPlotList->clearSelection();
117 }
118
119 /**
120  * @brief Handles the "Add plot" button click event.
121  * Validates selection and emits addPlotClicked() signal.
122  */
123 void Sidebar::onAddPlotClicked()
124 {
125     auto selectedItems = newPlotList->getSelectedItems();
126
127     if (selectedItems.empty()) {
128         QMessageBox::warning(this, "No Selection", "Please select
    probes before adding a plot.");
129         return;
130     }
131
132     Q_EMIT addPlotClicked();
133 }
```

C.2.14 SnapshotViewerWindow.cpp

This is a window to observe a snapshot from the triggers tab.

```
1  /**
2   * @file SnapshotViewerWindow.cpp
3   * @brief Implementation of the SnapshotViewerWindow class.
4   */
5
6  #include "SnapshotViewerWindow.h"
7
8  #include <QComboBox>
9  #include <QHBoxLayout>
10 #include <QLabel>
11 #include <QSlider>
12 #include <QVBoxLayout>
13 #include <QtCharts/QChartView>
```

```cpp
14  #include <QtCharts/QLineSeries>
15  #include <QtCharts/QValueAxis>
16  #include <QtConcurrent>
17  #include <qnamespace.h>
18
19  /**
20   * @brief Constructs a new SnapshotViewerWindow.
21   */
22  SnapshotViewerWindow::SnapshotViewerWindow(std::vector<Snapshot> &
        snapshotList, QWidget *parent)
23      : QWidget(parent)
24      , observableSelector(new QComboBox(this))
25      , timeSlider(new QSlider(Qt::Horizontal, this))
26      , timeLabel(new QLabel(this))
27      , chartView(new QChartView(new QChart(), this))
28  {
29      auto *mainLayout = new QHBoxLayout(this);
30
31      // Chart area
32      chartView->setRenderHint(QPainter::Antialiasing);
33      mainLayout->addWidget(chartView, 3); // 3/4 of space
34
35      // Controls
36      auto *controlLayout = new QVBoxLayout(this);
37      controlLayout->addWidget(new QLabel("Select Observable:"));
38      controlLayout->addWidget(observableSelector);
39
40      controlLayout->addWidget(new QLabel("Select Time:"));
41      controlLayout->addWidget(timeSlider);
42      controlLayout->addWidget(timeLabel);
43      controlLayout->addStretch();
44      mainLayout->addLayout(controlLayout, 1); // 1/4 of space
45
46      connect(observableSelector, &QComboBox::currentTextChanged, this,
        &SnapshotViewerWindow::onObservableChanged);
47      connect(timeSlider, &QSlider::valueChanged, this, &
        SnapshotViewerWindow::onTimeSliderChanged);
48
49      setSnapshots(snapshotList);
50  }
51
52  /**
53   * @brief Sets the snapshot data and updates the UI.
54   */
55  void SnapshotViewerWindow::setSnapshots(std::vector<Snapshot> &
        snapshotList)
56  {
57      snapshots.clear();
58      observableSelector->clear();
59
60      for (auto &s : snapshotList) {
61          snapshots[s.getName()] = s;
62          observableSelector->addItem(QString::fromStdString(s.getName()
        ));
63      }
64
```

```cpp
65      if (!snapshots.empty()) {
66          observableSelector->setCurrentIndex(0);
67          onObservableChanged(observableSelector->currentText());
68      }
69  }
70
71  /**
72   * @brief Handles the change of observable by updating the slider
73        range and triggering a plot update.
74   */
75  void SnapshotViewerWindow::onObservableChanged(const QString &name)
75  {
76      currentObservable = name.toStdString();
77
78      const auto &snapshot = snapshots.at(currentObservable);
79      int count = static_cast<int>(snapshot.getObservedSize());
80      timeSlider->setRange(0, std::max(0, count - 1));
81      timeSlider->setValue(0);
82      onTimeSliderChanged(0);
83  }
84
85  /**
86   * @brief Handles changes in the time slider by updating the chart.
87   */
88  void SnapshotViewerWindow::onTimeSliderChanged(int value)
89  {
90      if (snapshots.empty() || !snapshots.count(currentObservable))
91          return;
92
93      const auto &snapshot = snapshots.at(currentObservable);
94
95      if (snapshot.getObservedSize() == 0 || value >= static_cast<int>(
        snapshot.getObservedSize()))
96          return;
97
98      const auto obs = snapshot.getObservedDeltaQs()[value];
99      const auto calc = snapshot.getCalculatedSize() > value ? std::
        optional<DeltaQRepr>(snapshot.getCalculatedDeltaQs()[value]) : std
        ::nullopt;
100     const auto qta = snapshot.getQTAs()[value];
101     auto time = obs.time;
102
103     qint64 msTime = time / 1000000;
104     QDateTime timestamp = QDateTime::fromMSecsSinceEpoch(msTime);
105     timeLabel->setText(QString("Snapshot at: %1").arg(timestamp.
        toString()));
106
107     auto ret = QtConcurrent::run([=]() {
108         int bins = obs.deltaQ.getBins();
109         double binWidth = obs.deltaQ.getBinWidth();
110
111         QVector<QPointF> obsMean, obsLower, obsUpper, obsCdf, qtaData;
112         obsCdf.reserve(bins + 1);
113         obsMean.reserve(bins + 1);
114         obsLower.reserve(bins + 1);
115         obsUpper.reserve(bins + 1);
```

```
116
117         obsCdf.append(QPointF(0, 0));
118         obsMean.append(QPointF(0, 0));
119         obsLower.append(QPointF(0, 0));
120         obsUpper.append(QPointF(0, 0));
121
122         for (int i = 0; i < bins; ++i) {
123             double x = binWidth * (i + 1);
124             obsCdf.append(QPointF(x, obs.deltaQ.cdfAt(i)));
125             obsLower.append(QPointF(x, obs.bounds[i].lowerBound));
126             obsUpper.append(QPointF(x, obs.bounds[i].upperBound));
127             obsMean.append(QPointF(x, obs.bounds[i].mean));
128         }
129
130         if (qta.defined) {
131             double maxDelay = bins * binWidth;
132             qtaData = {
133                 {qta.perc_25, 0         },
134                 {qta.perc_25, 0.25      },
135                 {qta.perc_50, 0.25      },
136                 {qta.perc_50, 0.5       },
137                 {qta.perc_75, 0.5       },
138                 {qta.perc_75, 0.75      },
139                 {maxDelay,    0.75      },
140                 {maxDelay,    qta.cdfMax}
141             };
142         }
143
144         QVector<QPointF> calcCdf, calcMean, calcLower, calcUpper;
145         if (calc) {
146             int cbins = calc->deltaQ.getBins();
147             double cbinWidth = calc->deltaQ.getBinWidth();
148             calcCdf.reserve(cbins + 1);
149             calcMean.reserve(cbins + 1);
150             calcLower.reserve(cbins + 1);
151             calcUpper.reserve(cbins + 1);
152
153             calcCdf.append(QPointF(0, 0));
154             calcMean.append(QPointF(0, 0));
155             calcLower.append(QPointF(0, 0));
156             calcUpper.append(QPointF(0, 0));
157
158             for (int i = 0; i < cbins; ++i) {
159                 double x = cbinWidth * (i + 1);
160                 calcCdf.append(QPointF(x, calc->deltaQ.cdfAt(i)));
161                 calcLower.append(QPointF(x, calc->bounds[i].lowerBound
    ));
162                 calcUpper.append(QPointF(x, calc->bounds[i].upperBound
    ));
163                 calcMean.append(QPointF(x, calc->bounds[i].mean));
164             }
165         }
166
167         QMetaObject::invokeMethod(this, [=]() {
168             QChart *chart = chartView->chart();
169
```

```
170            // Clear previous content
171            chart->removeAllSeries();
172            const auto axes = chart->axes();
173            for (QAbstractAxis *axis : axes) {
174                chart->removeAxis(axis);
175                axis->deleteLater(); // Safe axis cleanup
176            }
177
178            chart->setTitle(QString::fromStdString(currentObservable)
     + QString(" - Time Index: %1").arg(value));
179
180            auto addSeries = [&](const QVector<QPointF> &data, const
     QString &name, const QColor &color) {
181                QLineSeries *series = new QLineSeries();
182                series->setName(name);
183                series->append(data);
184                series->setColor(color);
185                chart->addSeries(series);
186                return series;
187            };
188
189            addSeries(obsCdf, "Observed", Qt::blue);
190            addSeries(obsMean, "Obs Mean", Qt::yellow);
191            addSeries(obsLower, "Obs Lower", Qt::green);
192            addSeries(obsUpper, "Obs Upper", Qt::darkGreen);
193            addSeries(qtaData, "QTA", Qt::darkBlue);
194
195            if (!calcCdf.isEmpty()) {
196                addSeries(calcCdf, "Calculated", Qt::red);
197                addSeries(calcMean, "Calc Mean", Qt::darkYellow);
198                addSeries(calcLower, "Calc Lower", Qt::magenta);
199                addSeries(calcUpper, "Calc Upper", Qt::darkMagenta);
200            }
201
202            auto *axisX = new QValueAxis();
203            axisX->setTitleText("Delay (s)");
204            chart->addAxis(axisX, Qt::AlignBottom);
205
206            auto *axisY = new QValueAxis();
207            axisY->setTitleText("ΔQ(x)");
208            axisY->setRange(0, 1);
209            chart->addAxis(axisY, Qt::AlignLeft);
210
211            for (auto *series : chart->series()) {
212                series->attachAxis(axisX);
213                series->attachAxis(axisY);
214            }
215        });
216    });
217 }
```

C.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.

```cpp
#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 Erlang Endpoint",
    this);
    erlangMainLayout->addWidget(setErlangEndpointButton, 0, Qt::
    AlignLeft);

    // Third row: Start/Stop Wrapper
    QHBoxLayout *erlangButtonsLayout = new QHBoxLayout();
    stopErlangButton = new QPushButton("Stop Wrapper", this);
    startErlangButton = new QPushButton("Start Wrapper", 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);
```

```
48      // Second row: Start/Stop Server
49      QHBoxLayout *serverButtonsLayout = new QHBoxLayout();
50      startServerButton = new QPushButton("Start Oscilloscope Server",
    this);
51      stopServerButton = new QPushButton("Stop Oscilloscope Server",
    this);
52      serverButtonsLayout->addWidget(startServerButton);
53      serverButtonsLayout->addWidget(stopServerButton);
54      serverMainLayout->addLayout(serverButtonsLayout);
55
56      // Add to main layout
57      mainLayout->addWidget(erlangGroup);
58      mainLayout->addWidget(serverGroup);
59      mainLayout->addStretch();
60      setLayout(mainLayout);
61      setSizePolicy(QSizePolicy::Preferred, QSizePolicy::Maximum);
62
63      // Connect signals
64      connect(startErlangButton, &QPushButton::clicked, this, &
    StubControlWidget::onStartErlangClicked);
65      connect(stopErlangButton, &QPushButton::clicked, this, &
    StubControlWidget::onStopErlangClicked);
66      connect(startServerButton, &QPushButton::clicked, this, &
    StubControlWidget::onStartServerClicked);
67      connect(stopServerButton, &QPushButton::clicked, this, &
    StubControlWidget::onStopServerClicked);
68      connect(setErlangEndpointButton, &QPushButton::clicked, this, &
    StubControlWidget::onSetErlangEndpointClicked);
69  }
70
71  void StubControlWidget::onStartErlangClicked()
72  {
73      Application::getInstance().setStubRunning(true);
74  }
75
76  void StubControlWidget::onStopErlangClicked()
77  {
78      Application::getInstance().setStubRunning(false);
79  }
80
81  void StubControlWidget::onStartServerClicked()
82  {
83      QString ip = serverIpEdit->text();
84      int port = serverPortEdit->text().toInt();
85
86      if (Application::getInstance().startCppServer(ip.toStdString(),
    port)) {
87          startServerButton->setEnabled(false);
88          stopServerButton->setEnabled(true);
89      }
90  }
91
92  void StubControlWidget::onStopServerClicked()
93  {
94      Application::getInstance().stopCppServer();
95      startServerButton->setEnabled(true);
```

```
 96          stopServerButton->setEnabled(false);
 97 }
 98
 99 void StubControlWidget::onSetErlangEndpointClicked()
100 {
101      QString ip = erlangReceiverIpEdit->text();
102      int port = erlangReceiverPortEdit->text().toInt();
103
104      Application::getInstance().setErlangEndpoint(ip.toStdString(),
     port);
105 }
```

C.2.16   SystemCreationWidget.cpp

This widget allows the creation/update of a system, loading an already existing one or saving one.

```
 1 #include "SystemCreationWidget.h"
 2 #include "../Application.h"
 3 #include "../parser/SystemParserInterface.h"
 4 #include <QFileDialog>
 5 #include <QMessageBox>
 6 #include <fstream>
 7
 8 /**
 9  * @brief Constructs the SystemCreationWidget and sets up the UI.
10  * @param parent Parent QWidget.
11  */
12 SystemCreationWidget::SystemCreationWidget(QWidget *parent)
13      : QWidget(parent)
14 {
15      mainLayout = new QVBoxLayout(this);
16      mainLayout->setAlignment(Qt::AlignTop);
17      mainLayout->setSpacing(10);
18      mainLayout->setContentsMargins(0, 0, 0, 0);
19
20      systemLabel = new QLabel("Create or edit your system here");
21      systemTextEdit = new QTextEdit();
22
23      buttonLayout = new QHBoxLayout();
24      updateSystemButton = new QPushButton("Create or edit system");
25      saveSystemButton = new QPushButton("Save system to");
26      loadSystemButton = new QPushButton("Load system from");
27
28      buttonLayout->addWidget(updateSystemButton);
29      buttonLayout->addWidget(saveSystemButton);
30      buttonLayout->addWidget(loadSystemButton);
31
32      mainLayout->addWidget(systemLabel);
33      mainLayout->addWidget(systemTextEdit);
34      mainLayout->addLayout(buttonLayout);
35
36      connect(updateSystemButton, &QPushButton::clicked, this, &
     SystemCreationWidget::onUpdateSystem);
```

116

```
37      connect(saveSystemButton, &QPushButton::clicked, this, &
        SystemCreationWidget::saveSystemTo);
38      connect(loadSystemButton, &QPushButton::clicked, this, &
        SystemCreationWidget::loadSystem);
39  }
40
41  /**
42   * @brief Gets the text currently in the system text editor.
43   * @return System text as a std::string.
44   */
45  std::string SystemCreationWidget::getSystemText() const
46  {
47      return systemTextEdit->toPlainText().toStdString();
48  }
49
50  /**
51   * @brief Sets the content of the system text editor.
52   * @param text The new system text.
53   */
54  void SystemCreationWidget::setSystemText(const std::string &text)
55  {
56      systemTextEdit->setText(QString::fromStdString(text));
57  }
58
59  /**
60   * @brief Parses the system text and updates the application system if
         valid.
61   */
62  void SystemCreationWidget::onUpdateSystem()
63  {
64      std::string text = getSystemText();
65
66      try {
67          auto system = SystemParserInterface::parseString(text);
68          if (system.has_value()) {
69              Application::getInstance().setSystem(system.value());
70              system->setSystemDefinitionText(text);
71              Q_EMIT systemUpdated();
72          }
73      } catch (const std::exception &e) {
74          QMessageBox::critical(this, "Parsing error", e.what());
75      }
76  }
77
78  /**
79   * @brief Opens a dialog to save the current system to a file.
80   */
81  void SystemCreationWidget::saveSystemTo()
82  {
83      QFileDialog dialog(this);
84      dialog.setFileMode(QFileDialog::AnyFile);
85      QString filename = dialog.getSaveFileName(this, "Save file", "", "
        All files (* *.dq)");
86
87      if (!filename.isEmpty()) {
88          std::string systemText = getSystemText();
```

```
89          auto system = SystemParserInterface::parseString(systemText);
90
91          if (system.has_value()) {
92              std::ofstream outFile(filename.toStdString());
93              if (outFile.is_open()) {
94                  outFile << systemText;
95                  outFile.close();
96                  QMessageBox::information(this, "Success", "File saved
     successfully.");
97                  Q_EMIT systemSaved();
98              } else {
99                  QMessageBox::critical(this, "Error", "Could not open
     file for writing.");
100             }
101         } else {
102             QMessageBox::warning(this, "Error", "System parsing failed
     . File not saved.");
103         }
104     }
105 }
106
107 /**
108  * @brief Opens a dialog to load a system from a file, parses it, and
        updates the editor.
109  */
110 void SystemCreationWidget::loadSystem()
111 {
112     QFileDialog dialog(this);
113     std::string filename = dialog.getOpenFileName(this, "Select file",
        " ", "All files (* *.dq)").toStdString();
114
115     auto system = SystemParserInterface::parseFile(filename);
116     if (system.has_value()) {
117         Application::getInstance().setSystem(system.value());
118         std::ifstream file(filename);
119         std::string str;
120         std::string fileContents;
121         while (std::getline(file, str)) {
122             fileContents += str;
123             fileContents.push_back('\n');
124         }
125
126         system->setSystemDefinitionText(fileContents);
127         setSystemText(fileContents);
128         Q_EMIT systemLoaded();
129     }
130 }
```

C.2.17   TriggersTab.cpp

This tab holds the widgets to set/remove triggers and view fired ones.

```
1 #include "TriggersTab.h"
2 #include "SnapshotViewerWindow.h"
3 #include <QDateTime>
```

```
4  #include <QLabel>
5  #include <QString>
6  #include <QTimer>
7  #include <cstdlib>
8  #include <sstream>
9  #include <string>
10 #include <unordered_set>
11
12 TriggersTab::TriggersTab(QWidget *parent)
13     : QWidget(parent)
14 {
15     // Main layout setup
16     mainLayout = new QVBoxLayout(this);
17     formLayout = new QFormLayout();
18
19     // Observable selection dropdown
20     observableComboBox = new QComboBox(this);
21     connect(observableComboBox, &QComboBox::currentTextChanged, this,
    &TriggersTab::onObservableChanged);
22     formLayout->addRow("Probe:", observableComboBox);
23
24     // Sample limit controls
25     sampleLimitCheckBox = new QCheckBox("Sample Limit >", this);
26     sampleLimitSpinBox = new QSpinBox(this);
27     sampleLimitSpinBox->setRange(1, 100000);
28     sampleLimitSpinBox->setValue(sampleLimitThreshold);
29
30     sampleLimitLayout = new QHBoxLayout();
31     sampleLimitLayout->addWidget(sampleLimitCheckBox);
32     sampleLimitLayout->addWidget(sampleLimitSpinBox);
33
34     sampleLimitWidget = new QWidget(this);
35     sampleLimitWidget->setLayout(sampleLimitLayout);
36     formLayout->addRow(sampleLimitWidget);
37
38     // QTA bounds violation checkbox
39     qtaBoundsCheckBox = new QCheckBox("QTA Bound Violation", this);
40     formLayout->addRow(qtaBoundsCheckBox);
41
42     // Connect signals
43     connect(sampleLimitCheckBox, &QCheckBox::checkStateChanged, this,
    &TriggersTab::onTriggerChanged);
44     connect(sampleLimitSpinBox, QOverload<int>::of(&QSpinBox::
    valueChanged), this, &TriggersTab::onTriggerChanged);
45     connect(qtaBoundsCheckBox, &QCheckBox::checkStateChanged, this, &
    TriggersTab::onTriggerChanged);
46
47     mainLayout->addLayout(formLayout);
48
49     // Triggered events list
50     triggeredList = new QListWidget(this);
51     triggeredList->setSelectionMode(QAbstractItemView::SingleSelection
    );
52     triggeredList->setMinimumHeight(150);
53     connect(triggeredList, &QListWidget::itemClicked, this, &
    TriggersTab::onTriggeredItemClicked);
```

```
54
55       mainLayout->addWidget(new QLabel("Triggered Snapshots:"));
56       mainLayout->addWidget(triggeredList);
57
58       // Set up system observer
59       Application::getInstance().addObserver([this]() { this->
      populateObservables(); });
60
61       populateObservables();
62 }
63
64 TriggersTab::~TriggersTab()
65 {
66       delete mainLayout;
67 }
68
69 /**
70  * @brief Populates the observable dropdown with available probes and
      outcomes.
71  */
72 void TriggersTab::populateObservables()
73 {
74       try {
75           auto system = Application::getInstance().getSystem();
76           observableComboBox->clear();
77           for (const auto &[name, obs] : system->getObservables()) {
78               if (obs) {
79                   observableComboBox->addItem(QString::fromStdString(
      name));
80               }
81           }
82       } catch (std::exception &) {
83           return;
84       }
85 }
86
87 /**
88  * @brief Handles observable selection changes.
89  * @param name The newly selected observable name (unused, signal
      requires parameter).
90  */
91 void TriggersTab::onObservableChanged(const QString &)
92 {
93       updateCheckboxStates();
94 }
95
96 /**
97  * @brief Updates triggers when conditions change.
98  */
99 void TriggersTab::onTriggerChanged()
100 {
101       try {
102           auto observable = getCurrentObservable();
103           std::string name = observable->getName();
104
105           // Update sample limit trigger
```

```cpp
106            if (sampleLimitCheckBox->isChecked()) {
107                observable->addTrigger(
108                    TriggerType::SampleLimit,
109                    TriggerDefs::Conditions::SampleLimit(
       sampleLimitSpinBox->value()),
110                    [this, name](const DeltaQ &, const QTA &, std::
       uint64_t time) {
111                        this->captureSnapshots(time, name);
112                    },
113                    true,
114                    sampleLimitSpinBox->value());
115            } else {
116                observable->removeTrigger(TriggerType::SampleLimit);
117            }
118
119            // Update QTA bounds trigger
120            if (qtaBoundsCheckBox->isChecked()) {
121                observable->addTrigger(
122                    TriggerType::QTAViolation,
123                    TriggerDefs::Conditions::QTABounds(),
124                    [this, name](const DeltaQ &, const QTA &, std::
       uint64_t time) {
125                        this->captureSnapshots(time, name);
126                    },
127                    true,
128                    std::nullopt);
129            } else {
130                observable->removeTrigger(TriggerType::QTAViolation);
131            }
132        } catch (std::exception &) {
133            return;
134        }
135 }
136
137 /**
138  * @brief Captures snapshots when triggers are activated.
139  * @param time The timestamp of the trigger event.
140  * @param name The name of the observable that triggered.
141  */
142 void TriggersTab::captureSnapshots(std::uint64_t time, const std::
       string &name)
143 {
144     auto system = Application::getInstance().getSystem();
145     if (!system->isRecording()) {
146         system->setRecording(true);
147         QMetaObject::invokeMethod(
148             this,
149             [this, name, system, time]() {
150                 auto *timer = new QTimer(this);
151                 timer->setSingleShot(true);
152
153                 connect(timer, &QTimer::timeout, this, [=]() {
154                     system->getObservablesSnapshotAt(time);
155
156                     // Format timestamp for display
157                     qint64 msTime = time / 1000000;
```

```
158                        QDateTime timestamp = QDateTime ::
    fromMSecsSinceEpoch ( msTime );
159
160                        // Create display string
161                        QString timestampStr = timestamp . toString ();
162                        std :: ostringstream oss ;
163                        oss << "Snapshot at: " << timestampStr . toStdString
    () << " from " << name ;
164                        QString snapshotString = QString :: fromStdString (
    oss . str ());
165
166                        // Add to triggered list
167                        auto * item = new QListWidgetItem ( snapshotString );
168                        item -> setData ( Qt :: UserRole , static_cast < qulonglong
    >( time ));
169                        triggeredList -> addItem ( item );
170                        timer -> deleteLater ();
171
172                        system -> setRecording ( false );
173                    });
174
175                    timer -> start (5000);
176                },
177                Qt :: QueuedConnection );
178        }
179 }
180
181 /**
182  * @brief Updates checkbox states based on current triggers.
183  */
184 void TriggersTab :: updateCheckboxStates ()
185 {
186      try {
187          auto observable = getCurrentObservable ();
188          auto & manager = observable -> getTriggerManager ();
189
190          // Get active trigger types
191          auto all = manager . getAllTriggers ();
192          std :: unordered_set < TriggerType > activeTypes ;
193          for ( const auto & trigger : all ) {
194              if ( trigger . enabled )
195                  activeTypes . insert ( trigger . type );
196          }
197
198          // Update UI to match active triggers
199          sampleLimitCheckBox -> setChecked ( activeTypes . count ( TriggerType
    :: SampleLimit ));
200          qtaBoundsCheckBox -> setChecked ( activeTypes . count ( TriggerType ::
    QTAViolation ));
201
202          // Update sample limit value if trigger exists
203          for ( const auto & t : all ) {
204              if ( t . type == TriggerType :: SampleLimit && t .
    sampleLimitValue ) {
205                  sampleLimitSpinBox -> setValue (* t . sampleLimitValue );
206              }
```

```
207          }
208      } catch (std::exception &) {
209          sampleLimitCheckBox->setChecked(false);
210          qtaBoundsCheckBox->setChecked(false);
211      }
212 }
213
214 /**
215  * @brief Gets the currently selected observable.
216  * @return current observable.
217  * @throws std::runtime_error if system or observable doesn't exist.
218  */
219 std::shared_ptr<Observable> TriggersTab::getCurrentObservable()
220 {
221      auto system = Application::getInstance().getSystem();
222      if (!system)
223          throw std::runtime_error("System does not exist");
224
225      std::string name = observableComboBox->currentText().toStdString()
     ;
226      auto observable = system->getObservable(name);
227      if (!observable)
228          throw std::runtime_error("Observable does not exist");
229
230      return observable;
231 }
232
233 /**
234  * @brief Handles triggered item clicks to show snapshots.
235  * @param item The clicked list item containing snapshot data.
236  */
237 void TriggersTab::onTriggeredItemClicked(QListWidgetItem *item)
238 {
239      if (!item)
240          return;
241
242      // Retrieve timestamp from item data
243      qulonglong timestamp = item->data(Qt::UserRole).toULongLong();
244      if (timestamp == 0)
245          return;
246
247      // Find and display corresponding snapshot
248      auto system = Application::getInstance().getSystem();
249      const auto &snapshots = system->getAllSnapshots();
250      auto it = snapshots.find(timestamp);
251      if (it != snapshots.end()) {
252          const auto &snapshotList = it->second;
253
254          auto *viewer = new SnapshotViewerWindow(const_cast<std::vector
     <Snapshot> &>(snapshotList));
255          viewer->setAttribute(Qt::WA_DeleteOnClose);
256          viewer->resize(800, 600);
257          viewer->show();
258      }
259 }
```

C.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

C.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.

```cpp
#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; });
```

```
35        for (auto it = lower; it != upper; ++it) {
36            selectedSamples.emplace_back(*it);
37        }
38        samples.erase(std::remove_if(samples.begin(), samples.end(), [
      upperTime](const Sample &s) { return s.endTime < upperTime; }),
      samples.end());
39
40        return selectedSamples;
41    }
42
43    DeltaQ Observable::calculateObservedDeltaQ(std::uint64_t
      timeLowerBound, std::uint64_t timeUpperBound)
44    {
45        auto samplesInRange = getSamplesInRange(timeLowerBound,
      timeUpperBound);
46        if (samplesInRange.empty()) {
47            DeltaQ deltaQ = DeltaQ();
48            updateSnapshot(timeLowerBound, deltaQ);
49            return deltaQ;
50        }
51
52        DeltaQ deltaQ {getBinWidth(), samplesInRange, nBins};
53        std::cout << deltaQ.toString() << "\n";
54        updateSnapshot(timeLowerBound, deltaQ);
55
56        triggerManager.evaluate(deltaQ, qta, timeLowerBound);
57        return deltaQ;
58    }
59
60    void Observable::updateSnapshot(uint64_t timeLowerBound, DeltaQ &
      deltaQ)
61    {
62        if (!(deltaQ == DeltaQ())) {
63            observedInterval.addDeltaQ(deltaQ);
64            confidenceIntervalHistory.push_back(deltaQ);
65
66            if (confidenceIntervalHistory.size() > MAX_DQ) {
67                observedInterval.removeDeltaQ(confidenceIntervalHistory.
      front());
68                confidenceIntervalHistory.pop_front();
69            }
70        }
71        observableSnapshot.addObservedDeltaQ(timeLowerBound, deltaQ,
      observedInterval.getBounds());
72        observableSnapshot.addQTA(timeLowerBound, qta);
73        if (!recording && (confidenceIntervalHistory.size() > MAX_DQ)) {
74            observableSnapshot.resizeTo(MAX_DQ);
75        }
76    }
77
78    DeltaQ Observable::getObservedDeltaQ(uint64_t timeLowerBound, uint64_t
      timeUpperBound)
79    {
80        std::lock_guard<std::mutex> lock(observedMutex);
81        auto deltaQRepr = observableSnapshot.getObservedDeltaQAtTime(
      timeLowerBound);
```

```cpp
82      if (!deltaQRepr.has_value()) {
83          calculateObservedDeltaQ(timeLowerBound, timeUpperBound);
84          deltaQRepr = observableSnapshot.getObservedDeltaQAtTime(
    timeLowerBound);
85      }
86      return deltaQRepr.value().deltaQ;
87  }
88
89  DeltaQRepr Observable::getObservedDeltaQRepr(uint64_t timeLowerBound,
    uint64_t timeUpperBound)
90  {
91      std::lock_guard<std::mutex> lock(observedMutex);
92      auto deltaQRepr = observableSnapshot.getObservedDeltaQAtTime(
    timeLowerBound);
93      if (!deltaQRepr.has_value()) {
94          calculateObservedDeltaQ(timeLowerBound, timeUpperBound);
95          deltaQRepr = observableSnapshot.getObservedDeltaQAtTime(
    timeLowerBound);
96      }
97      return deltaQRepr.value();
98  }
99
100 void Observable::setQTA(const QTA &newQTA)
101 {
102     if (newQTA.perc_25 > maxDelay || newQTA.perc_50 > maxDelay ||
    newQTA.perc_75 > maxDelay)
103         throw std::invalid_argument("Percentages should not be bigger
    than maximum delay " + std::to_string(maxDelay));
104     qta = newQTA;
105 }
106
107 void Observable::addTrigger(TriggerType type, TriggerDefs::Condition
    condition, TriggerDefs::Action action, bool enabled, std::optional
    <int> sampleLimitVal)
108 {
109     auto &tm = triggerManager;
110     tm.addTrigger(type, condition, action, enabled);
111     auto &trigger = tm.getTriggersByType(type).back();
112     trigger.sampleLimitValue = sampleLimitVal;
113 }
114
115 void Observable::removeTrigger(TriggerType type)
116 {
117     triggerManager.removeTriggersByType(type);
118 }
119
120 void Observable::setRecording(bool isRecording)
121 {
122     if (recording && !isRecording) {
123         recording = isRecording;
124         observableSnapshot.resizeTo(30); // FIXME magic numb
125     }
126     recording = isRecording;
127 }
128
129 Snapshot Observable::getSnapshot()
```

```
130 {
131     return observableSnapshot;
132 }
133
134 double Observable::setNewParameters(int newExp, int newNBins)
135 {
136     std::lock_guard<decltype(paramMutex)> lock(paramMutex);
137
138     bool binsChanged = (newNBins != nBins);
139     bool expChanged = (newExp != deltaTExp);
140
141     nBins = newNBins;
142     deltaTExp = newExp;
143
144     maxDelay = DELTA_T_BASE * std::pow(2, deltaTExp) * nBins;
145
146     if (qta.perc_25 > maxDelay || qta.perc_50 > maxDelay || qta.
    perc_75 > maxDelay) {
147         qta = QTA::create(0, 0, 0, qta.cdfMax, false);
148     }
149
150     if (binsChanged || expChanged) {
151         observedInterval.setNumBins(nBins);
152         confidenceIntervalHistory.clear();
153     }
154
155     return maxDelay;
156 }
157
158 std::string Observable::getName() const &
159 {
160     return name;
161 }
```

C.3.2   Operator.cpp

This class represent a generic operator, it can be either a FTF, ATF or PC. It allows calculating the "calculated ΔQ".

```
1  #include "Operator.h"
2
3  #include <numeric>
4
5  #include "../maths/DeltaQOperations.h"
6  #include "Observable.h"
7  #include "OperatorType.h"
8  #include <cmath>
9  #include <limits>
10 #define OP_EPSILON std::numeric_limits<double>::epsilon()
11
12 Operator::Operator(const std::string &name, OperatorType type)
13     : Observable(name)
14     , type(type)
15     , calculatedInterval(50)
16 {
```

```cpp
17  }
18
19  Operator::~Operator() = default;
20
21  void Operator::setProbabilities(const std::vector<double> &probs)
22  {
23      if (type != OperatorType::PRB) {
24          throw std::invalid_argument("Only probabilistic operators
            accept probabilities");
25      }
26
27      double result = std::reduce(probs.begin(), probs.end());
28
29      if (std::fabs(1 - result) > OP_EPSILON)
30          throw std::logic_error("Result should approximate to 1");
31
32      probabilities = probs;
33  }
34
35  DeltaQ Operator::calculateCalculatedDeltaQ(uint64_t timeLowerBound,
        uint64_t timeUpperBound)
36  {
37      if (observableSnapshot.getCalculatedDeltaQAtTime(timeLowerBound).
        has_value()) {
38          return observableSnapshot.getCalculatedDeltaQAtTime(
            timeLowerBound).value().deltaQ;
39      }
40      std::vector<DeltaQ> deltaQs;
41
42      // Get DeltaQ for all children
43      for (auto &childrenLinks : causalLinks) {
44          std::vector<DeltaQ> childrenDeltaQs;
45
46          // For each children in causal link, get their DeltaQ
47          for (auto &component : childrenLinks) {
48              childrenDeltaQs.push_back(component->getObservedDeltaQ(
                timeLowerBound, timeUpperBound));
49          }
50          // Get the convolution of the components in a children
51          deltaQs.push_back(convolveN(childrenDeltaQs));
52      }
53      DeltaQ result;
54
55      // Choose appropriate operation
56      if (type == OperatorType::FTF)
57          result = firstToFinish(deltaQs);
58      else if (type == OperatorType::PRB)
59          result = probabilisticChoice(probabilities, deltaQs);
60      else
61          result = allToFinish(deltaQs);
62
63      int calculatedBins = result.getBins();
64
65      if (calculatedBins != calculatedInterval.getBins()) {
66          calculatedInterval.setNumBins(calculatedBins);
67          calculatedDeltaQHistory.clear();
```

```
68        }
69
70        calculatedInterval.addDeltaQ(result);
71        calculatedDeltaQHistory.push_back(result);
72
73        if (calculatedDeltaQHistory.size() > MAX_DQ) {
74            calculatedInterval.removeDeltaQ(calculatedDeltaQHistory.front
      ());
75            calculatedDeltaQHistory.pop_front();
76        }
77
78        observableSnapshot.addCalculatedDeltaQ(timeLowerBound, result,
      calculatedInterval.getBounds());
79
80        if (!recording && calculatedDeltaQHistory.size() > MAX_DQ) {
81            observableSnapshot.removeOldestCalculatedDeltaQ();
82        }
83        return result;
84 }
85
86 DeltaQRepr Operator::getCalculatedDeltaQRepr(uint64_t timeLowerBound,
      uint64_t timeUpperBound)
87 {
88        std::lock_guard<std::mutex> lock(calcMutex);
89        auto deltaQRepr = observableSnapshot.getCalculatedDeltaQAtTime(
      timeLowerBound);
90        if (!deltaQRepr.has_value()) {
91            calculateCalculatedDeltaQ(timeLowerBound, timeUpperBound);
92            deltaQRepr = observableSnapshot.getCalculatedDeltaQAtTime(
      timeLowerBound);
93        }
94        return deltaQRepr.value();
95 }
```

C.3.3   Outcome.cpp

This class represents a simple outcome.

```
1 #include "Outcome.h"
2
3 Outcome::Outcome(const std::string &name)
4      : Observable(name)
5 {
6 }
7
8 Outcome::~Outcome() = default;
```

C.3.4   Probe.cpp

This class represents what we described as "sub-outcome diagram". As the operator class, it allows calculating the "calculated ΔQ".

```
1 #include "Probe.h"
2 #include "../maths/ConfidenceInterval.h"
```

```cpp
#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
    ());
        calculatedDeltaQHistory.pop_front();
    }

    observableSnapshot.addCalculatedDeltaQ(timeLowerBound, result,
    calculatedInterval.getBounds());

    if (!recording && calculatedDeltaQHistory.size() > MAX_DQ) {
```

```cpp
            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();
}
```

C.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;
    }
```

```
17 }
18
19 void System::setProbes(std::unordered_map<std::string, std::shared_ptr
      <Probe>> probesMap)
20 {
21     probes = probesMap;
22     for (auto &[name, probe] : probes) {
23         observables[name] = probe;
24     }
25 }
26
27 void System::setOperators(std::unordered_map<std::string, std::
      shared_ptr<Operator>> operatorsMap)
28 {
29     operators = operatorsMap;
30     for (auto &[name, op] : operators) {
31         observables[name] = op;
32     }
33 }
34 std::shared_ptr<Outcome> System::getOutcome(const std::string &name)
35 {
36     return outcomes[name];
37 }
38
39 void System::setObservableParameters(std::string &componentName, int
      exponent, int numBins)
40 {
41     if (auto it = observables.find(componentName); it != observables.
      end()) {
42         double maxDelay = it->second->setNewParameters(exponent,
      numBins);
43         Application::getInstance().sendDelayChange(componentName,
      maxDelay * 1000);
44     }
45 }
46
47 void System::addSample(std::string &componentName, Sample &sample)
48 {
49     if (auto it = observables.find(componentName); it != observables.
      end()) {
50         it->second->addSample(sample);
51     }
52 }
53
54 DeltaQ System::calculateDeltaQ()
55 {
56     return DeltaQ();
57 }
58
59 [[nodiscard]] std::unordered_map<std::string, std::shared_ptr<Outcome
      >> &System::getOutcomes()
60 {
61     return outcomes;
62 }
63
64 [[nodiscard]] std::unordered_map<std::string, std::shared_ptr<Operator
```

```cpp
        >> &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)
```

```cpp
118 {
119     systemDefinitionText = text;
120 }
121
122 std::string System::getSystemDefinitionText()
123 {
124     return systemDefinitionText;
125 }
126
127 std::shared_ptr<Observable> System::getObservable(const std::string &
        name)
128 {
129     return observables[name];
130 }
131
132 void System::setRecording(bool isRecording)
133 {
134     if (recordingTrigger && isRecording) {
135         return;
136     }
137
138     recordingTrigger = isRecording;
139     for (auto &obs : observables) {
140         if (obs.second) {
141             obs.second->setRecording(isRecording);
142         }
143     }
144 }
145
146 bool System::isRecording() const
147 {
148     return recordingTrigger;
149 }
150
151 void System::getObservablesSnapshotAt(std::uint64_t time)
152 {
153     std::vector<Snapshot> result;
154     for (auto &[name, observable] : observables) {
155         if (observable)
156             result.push_back(observable->getSnapshot());
157     }
158
159     snapshots[time] = std::move(result); // store the snapshots by
        timestamp
160 }
161
162 std::map<std::uint64_t, std::vector<Snapshot>> System::getAllSnapshots
        ()
163 {
164     return snapshots;
165 }
```

C.4 ΔQ

The "maths" folder represents all the classes related to ΔQ, where mathematical operations are being done (hence the "maths" name).

C.4.1 ConfidenceInterval.cpp

This class represents the confidence bounds described earlier.

```cpp
#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()) {
```

```cpp
46          return;
47      }
48
49      const auto &cdf = deltaQ.getCdfValues();
50      if (cdf.size() != numBins) {
51          return; // Returning a previous DeltaQ which had different
    bins
52      }
53
54      for (size_t i = 0; i < cdf.size(); ++i) {
55          const double cdfValue = cdf[i];
56
57          cdfSum[i] -= cdfValue;
58          cdfSumSquares[i] -= cdfValue * cdfValue;
59
60          if (cdfSampleCounts[i] == 0) {
61              return; // Removing more than needed
62          }
63
64          cdfSampleCounts[i] -= 1;
65      }
66
67      updateConfidenceInterval();
68  }
69
70  void ConfidenceInterval::updateConfidenceInterval()
71  {
72      for (size_t i = 0; i < bounds.size(); ++i) {
73          unsigned int n = cdfSampleCounts[i];
74          if (cdfSampleCounts[i] == 0) {
75              bounds[i].lowerBound = 0.0;
76              bounds[i].upperBound = 0.0;
77              continue;
78          }
79
80          double mean = cdfSum[i] / n;
81          double meanSquare = cdfSumSquares[i] / n;
82          double variance = meanSquare - (mean * mean);
83          double stddev = std::sqrt(std::max(variance, 0.0));
84          double marginOfError = z * stddev / std::sqrt(n);
85          bounds[i].lowerBound = std::max(0.0, mean - marginOfError);
86          bounds[i].upperBound = std::min(1.0, mean + marginOfError);
87          bounds[i].mean = mean;
88      }
89  }
90
91  void ConfidenceInterval::reset()
92  {
93      bounds = std::vector<Bound>();
94      bounds.resize(numBins);
95      cdfSum = std::vector<double>();
96      cdfSum.resize(numBins);
97      cdfSumSquares = std::vector<double>();
98      cdfSumSquares.resize(numBins);
99      cdfSampleCounts = std::vector<unsigned int>();
100     cdfSampleCounts.resize(numBins);
```

```
101 }
102
103 void ConfidenceInterval::setNumBins(int newNumBins)
104 {
105     numBins = newNumBins;
106     reset();
107 }
108
109 std::vector<Bound> ConfidenceInterval::getBounds() const
110 {
111     return bounds;
112 }
113
114 unsigned int ConfidenceInterval::getBins()
115 {
116     return numBins;
117 }
```

C.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.

```
1
2
3 #include "DeltaQ.h"
4
5 #include <algorithm>
6 #include <cmath>
7 #include <functional>
8 #include <iomanip>
9 #include <iostream>
10
11 #include <chrono>
12 #include <iostream>
13 DeltaQ::DeltaQ(const double binWidth)
14     : binWidth(binWidth)
15     , bins(0)
16     , qta()
17 {
18 }
19
20 DeltaQ::DeltaQ(const double binWidth, const std::vector<double> &
    values, const bool isPdf)
21     : binWidth(binWidth)
22     , bins(values.size()) // Values is binned data
23     , qta()
24 {
25     if (isPdf) {
26         pdfValues = values;
27         calculateCDF();
28     } else {
29         cdfValues = values;
30         calculatePDF();
31     }
```

```
32 }
33 DeltaQ::DeltaQ(double binWidth, std::vector<Sample> &samples)
34     : binWidth(binWidth)
35     , bins {50}
36     , qta()
37 {
38     calculateDeltaQ(samples);
39 }
40
41 DeltaQ::DeltaQ(double binWidth, std::vector<Sample> &samples, int bins
       )
42     : binWidth(binWidth)
43     , bins(bins)
44     , qta()
45 {
46     calculateDeltaQ(samples);
47 }
48
49 void DeltaQ::calculateDeltaQ(std::vector<Sample> &outcomeSamples)
50 {
51
52     double size = binWidth * bins;
53     if (outcomeSamples.empty() || binWidth <= 0) {
54         bins = 0;
55         return;
56     }
57     std::vector<double> histogram(bins, 0.0);
58     totalSamples = outcomeSamples.size();
59     long long successfulSamples = 0;
60
61     std::sort(outcomeSamples.begin(), outcomeSamples.end(), [](const
       Sample &a, const Sample &b) { return a.elapsedTime < b.elapsedTime
       ; });
62
63     for (const auto &sample : outcomeSamples) {
64         if (sample.status != Status::SUCCESS) {
65             continue; // Exclude failed samples from histogram but
       count them
66         }
67
68         double elapsed = sample.elapsedTime;
69
70         if (elapsed < 0 || std::isnan(elapsed) || std::isinf(elapsed))
        {
71             std::cerr << "Warning: Invalid sample value: " << elapsed
       << std::endl;
72             continue;
73         }
74         double invBinWidth = 1.0 / binWidth;
75         int bin = static_cast<int>(elapsed * invBinWidth);
76         if (bin < 0) {
77             std::cerr << "Warning: Negative bin value: " << bin << std
       ::endl;
78             continue;
79         }
80         if (bin >= bins) {
```

```
81              if (elapsed > size) {
82                  continue;
83              }
84              bin = bins - 1;
85          }
86
87          successfulSamples++;
88          histogram[bin] += 1.0;
89      }
90      // Calculate PDF
91      for (double &val : histogram) {
92          val /= totalSamples;
93      }
94      pdfValues = std::move(histogram);
95
96      calculateCDF();
97      calculateQuartiles(outcomeSamples);
98  }
99
100 void DeltaQ::calculateQuartiles(std::vector<Sample> &outcomeSamples)
101 {
102     if (outcomeSamples.empty()) {
103         return;
104     }
105     const size_t n = outcomeSamples.size();
106
107     auto getElapsedAt = [&](size_t index) -> double { return
    outcomeSamples[index].elapsedTime; };
108
109     auto getPercentile = [&](double p) -> double {
110         double pos = p * (n - 1);
111         auto idx = static_cast<size_t>(pos);
112         double frac = pos - idx;
113
114         if (idx + 1 < n) {
115             // Linear interpolation between samples[idx] and samples[
    idx + 1]
116             return getElapsedAt(idx) * (1.0 - frac) + getElapsedAt(idx
     + 1) * frac;
117         }
118         return getElapsedAt(idx);
119     };
120     qta = QTA::create(getPercentile(0.25), getPercentile(0.50),
    getPercentile(0.75), ((cdfAt(bins - 1)) > 1) ? 1 : cdfAt(bins - 1)
    );
121 }
122
123 void DeltaQ::calculateCDF()
124 {
125     cdfValues.clear();
126     cdfValues.reserve(bins);
127     double cumulativeSum = 0;
128     for (const double &pdfValue : pdfValues) {
129         cumulativeSum += pdfValue;
130         cdfValues.push_back(cumulativeSum);
131     }
```

```cpp
132 }
133
134 void DeltaQ::calculatePDF()
135 {
136     pdfValues.clear();
137
138     double previous = 0;
139     for (const double &cdfValue : cdfValues) {
140         pdfValues.push_back(cdfValue - previous);
141         previous = cdfValue;
142     }
143 }
144
145 QTA DeltaQ::getQTA() const
146 {
147     return qta;
148 }
149
150 const std::vector<double> &DeltaQ::getPdfValues() const
151 {
152     return pdfValues;
153 }
154
155 const std::vector<double> &DeltaQ::getCdfValues() const
156 {
157     return cdfValues;
158 }
159
160 const unsigned int DeltaQ::getTotalSamples() const
161 {
162     return totalSamples;
163 }
164
165 void DeltaQ::setBinWidth(double newWidth)
166 {
167     binWidth = newWidth;
168 }
169
170 double DeltaQ::getBinWidth() const
171 {
172     return binWidth;
173 }
174
175 int DeltaQ::getBins() const
176 {
177     return bins;
178 }
179
180 double DeltaQ::pdfAt(int x) const
181 {
182     if (bins != 0) {
183         if (x >= bins) {
184             return 0.0;
185         }
186         return pdfValues.at(x);
187     }
```

```cpp
188     return 0;
189 }
190
191 double DeltaQ::cdfAt(int x) const
192 {
193     if (bins != 0) {
194         if (x >= bins) {
195             return cdfValues.at(bins - 1);
196         }
197         return cdfValues.at(x);
198     }
199     return 0;
200 }
201
202 bool DeltaQ::operator<(const DeltaQ &other) const
203 {
204     return this->bins < other.bins;
205 }
206
207 bool DeltaQ::operator>(const DeltaQ &other) const
208 {
209     return this->bins > other.bins;
210 }
211
212 bool DeltaQ::operator==(const DeltaQ &deltaQ) const
213 {
214     return pdfValues == deltaQ.getPdfValues();
215 }
216
217 DeltaQ operator*(const DeltaQ &deltaQ, double constant)
218 {
219     DeltaQ result(deltaQ.binWidth);
220     result.bins = deltaQ.bins;
221
222     result.pdfValues = deltaQ.pdfValues;
223     std::transform(result.pdfValues.begin(), result.pdfValues.end(),
    result.pdfValues.begin(), [constant](double value) { return value
    * constant; });
224
225     result.cdfValues = deltaQ.cdfValues;
226     std::transform(result.cdfValues.begin(), result.cdfValues.end(),
    result.cdfValues.begin(), [constant](double value) { return value
    * constant; });
227
228     return result;
229 }
230
231 template <typename BinaryOperation>
232 DeltaQ applyBinaryOperation(const DeltaQ &lhs, const DeltaQ &rhs,
    BinaryOperation op)
233 {
234     const DeltaQ &highestDeltaQ = (lhs > rhs) ? lhs : rhs;
235     const DeltaQ &otherDeltaQ = (lhs > rhs) ? rhs : lhs;
236
237     std::vector<double> resultingCdf;
238     resultingCdf.reserve(highestDeltaQ.getBins());
```

141

```
239
240      for (size_t i = 0; i < highestDeltaQ.getBins(); i++) {
241          double result = op(highestDeltaQ.cdfAt(i), otherDeltaQ.cdfAt(i
     ));
242          resultingCdf.push_back(result);
243      }
244
245      return {highestDeltaQ.getBinWidth(), resultingCdf, false};
246 }
247
248 DeltaQ operator*(double constant, const DeltaQ &deltaQ)
249 {
250      return deltaQ * constant;
251 }
252
253 DeltaQ operator+(const DeltaQ &lhs, const DeltaQ &rhs)
254 {
255      return applyBinaryOperation(lhs, rhs, std::plus<>());
256 }
257
258 DeltaQ operator-(const DeltaQ &lhs, const DeltaQ &rhs)
259 {
260      return applyBinaryOperation(lhs, rhs, std::minus<>());
261 }
262
263 DeltaQ operator*(const DeltaQ &lhs, const DeltaQ &rhs)
264 {
265      return applyBinaryOperation(lhs, rhs, std::multiplies<>());
266 }
267
268 std::string DeltaQ::toString() const
269 {
270      std::ostringstream oss;
271      oss << "<";
272
273      // Iterate through CDF values to construct the string
274      for (size_t i = 0; i < pdfValues.size(); ++i) {
275          const double bin = (i + 1) * binWidth;
276          oss << "(" << std::fixed << std::setprecision(7) << bin << ",
     " << std::setprecision(7) << pdfValues[i] << ")";
277          if (i < pdfValues.size() - 1) {
278              oss << ", ";
279          }
280      }
281
282      oss << ">";
283      return oss.str();
284 }
```

C.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.

```cpp
1  #include "DeltaQOperations.h"
2  #include "DeltaQ.h"
3  #include <fftw3.h>
4  #include <iostream>
5  #include <math.h>
6  #include <mutex>
7  #include <vector>
8  DeltaQ rebin(const DeltaQ &source, double targetBinWidth)
9  {
10     double originalBinWidth = source.getBinWidth();
11
12     if (std::abs(originalBinWidth - targetBinWidth) < 1e-9) {
13         return source; // Already same bin width
14     }
15
16     if (targetBinWidth < originalBinWidth) {
17         throw std::invalid_argument("Target bin width must be greater
   than or equal to original.");
18     }
19
20     int factor = static_cast<int>(std::round(targetBinWidth /
   originalBinWidth));
21     const auto &originalPdf = source.getPdfValues();
22
23     int newNumBins = static_cast<int>(std::ceil(originalPdf.size() /
   static_cast<double>(factor)));
24     std::vector<double> newPdf(newNumBins, 0.0);
25
26     for (size_t i = 0; i < originalPdf.size(); ++i) {
27         newPdf[i / factor] += originalPdf[i];
28     }
29
30     return DeltaQ(targetBinWidth, newPdf, true);
31 }
32
33 DeltaQ convolve(const DeltaQ &lhs, const DeltaQ &rhs)
34 {
35     double commonBinWidth = std::max(lhs.getBinWidth(), rhs.
   getBinWidth());
36
37     if (lhs == DeltaQ()) {
38         return rhs;
39     }
40     if (rhs == DeltaQ()) {
41         return lhs;
42     }
43
44     DeltaQ lhsRebinned = rebin(lhs, commonBinWidth);
45     DeltaQ rhsRebinned = rebin(rhs, commonBinWidth);
46
47     const auto &lhsPdf = lhsRebinned.getPdfValues();
48     const auto &rhsPdf = rhsRebinned.getPdfValues();
49
50     int resultSize = lhsPdf.size() + rhsPdf.size() - 1;
51     std::vector<double> resultPdf(resultSize, 0.0);
```

```cpp
52
53     for (size_t i = 0; i < lhsPdf.size(); ++i) {
54         for (size_t j = 0; j < rhsPdf.size(); ++j) {
55             resultPdf[i + j] += lhsPdf[i] * rhsPdf[j];
56         }
57     }
58
59     return DeltaQ(commonBinWidth, resultPdf, true);
60 }
61
62 // Inspired by https://github.com/jeremyfix/FFTConvolution/blob/master
       /Convolution/src/convolution_fftw.h
63 DeltaQ convolveFFT(const DeltaQ &lhs, const DeltaQ &rhs)
64 {
65     if (lhs == DeltaQ()) {
66         return rhs;
67     }
68     if (rhs == DeltaQ()) {
69         return lhs;
70     }
71
72     // Find a common bin width and rebin accordingly
73     double commonBinWidth = std::max(lhs.getBinWidth(), rhs.
       getBinWidth());
74
75     DeltaQ lhsRebinned = rebin(lhs, commonBinWidth);
76     DeltaQ rhsRebinned = rebin(rhs, commonBinWidth);
77
78     const auto &lhsPdf = lhsRebinned.getPdfValues();
79     const auto &rhsPdf = rhsRebinned.getPdfValues();
80
81     // Find the power of 2 nearest to the convolution size
82     size_t lhsSize = lhsPdf.size();
83     size_t rhsSize = rhsPdf.size();
84     size_t convSize = lhsSize + rhsSize - 1;
85     size_t fftSize = 1;
86     while (fftSize < convSize)
87         fftSize <<= 1;
88
89     // Pad pdf with zeroes until end of PDF
90     std::vector<double> lhsPadded(fftSize, 0.0);
91     std::vector<double> rhsPadded(fftSize, 0.0);
92     std::copy(lhsPdf.begin(), lhsPdf.end(), lhsPadded.begin());
93     std::copy(rhsPdf.begin(), rhsPdf.end(), rhsPadded.begin());
94     static std::mutex fftw_mutex;
95     {
96         std::lock_guard<std::mutex> lock(fftw_mutex);
97         fftw_complex *lhsFreq = (fftw_complex *)fftw_malloc(sizeof(
       fftw_complex) * (fftSize / 2 + 1));
98         fftw_complex *rhsFreq = (fftw_complex *)fftw_malloc(sizeof(
       fftw_complex) * (fftSize / 2 + 1));
99         double *lhsTime = lhsPadded.data();
100        double *rhsTime = rhsPadded.data();
101
102        // Transform real input to complex output
103        fftw_plan planLhs = fftw_plan_dft_r2c_1d(fftSize, lhsTime,
```

```cpp
          lhsFreq, FFTW_ESTIMATE);
104       fftw_plan planRhs = fftw_plan_dft_r2c_1d(fftSize, rhsTime,
          rhsFreq, FFTW_ESTIMATE);
105       fftw_execute(planLhs);
106       fftw_execute(planRhs);
107
108       // Do complex multiplication, r: ac - bd i : ad + bc
109       fftw_complex *resultFreq = (fftw_complex *)fftw_malloc(sizeof(
          fftw_complex) * (fftSize / 2 + 1));
110       for (size_t i = 0; i < fftSize / 2 + 1; ++i) {
111           double a = lhsFreq[i][0], b = lhsFreq[i][1];
112           double c = rhsFreq[i][0], d = rhsFreq[i][1];
113           resultFreq[i][0] = a * c - b * d;
114           resultFreq[i][1] = a * d + b * c;
115       }
116
117       // Invert from complex plane to real plane
118       std::vector<double> resultTime(fftSize);
119       fftw_plan planInv = fftw_plan_dft_c2r_1d(fftSize, resultFreq,
          resultTime.data(), FFTW_ESTIMATE);
120       fftw_execute(planInv);
121
122       for (auto &val : resultTime)
123           val /= fftSize;
124
125       resultTime.resize(convSize);
126
127       fftw_destroy_plan(planLhs);
128       fftw_destroy_plan(planRhs);
129       fftw_destroy_plan(planInv);
130       fftw_free(lhsFreq);
131       fftw_free(rhsFreq);
132       fftw_free(resultFreq);
133       return {lhsRebinned.getBinWidth(), resultTime, true};
134   }
135 }
136
137 DeltaQ convolveN(const std::vector<DeltaQ> &deltaQs)
138 {
139     if (deltaQs.empty())
140         return DeltaQ();
141
142     DeltaQ result = deltaQs[0];
143     for (size_t i = 1; i < deltaQs.size(); ++i) {
144         result = convolveFFT(result, deltaQs[i]);
145     }
146     return result;
147 }
148
149 DeltaQ probabilisticChoice(const std::vector<double> &probabilities,
        const std::vector<DeltaQ> &deltaQs)
150 {
151     std::vector<DeltaQ> nonEmpty;
152     std::vector<double> effectiveProbs;
153     double commonBinWidth = 0.0;
154
```

```
155     for (size_t i = 0; i < deltaQs.size(); ++i) {
156         if (deltaQs[i] == DeltaQ()) {
157             continue;
158         }
159         nonEmpty.push_back(deltaQs[i]);
160         effectiveProbs.push_back(probabilities[i]);
161         commonBinWidth = std::max(commonBinWidth, deltaQs[i].
    getBinWidth());
162     }
163
164     if (nonEmpty.empty()) {
165         return DeltaQ();
166     }
167
168     for (auto &dq : nonEmpty) {
169         dq = rebin(dq, commonBinWidth);
170     }
171
172     std::vector<DeltaQ> scaledDeltaQs;
173     for (size_t i = 0; i < nonEmpty.size(); ++i) {
174         scaledDeltaQs.push_back(nonEmpty[i] * effectiveProbs[i]);
175     }
176
177     DeltaQ result = scaledDeltaQs[0];
178     for (size_t i = 1; i < scaledDeltaQs.size(); ++i) {
179         result = result + scaledDeltaQs[i];
180     }
181
182     return result;
183 }
184
185 DeltaQ firstToFinish(const std::vector<DeltaQ> &deltaQs)
186 {
187     std::vector<DeltaQ> nonEmpty;
188     double commonBinWidth = 0.0;
189
190     for (const auto &dq : deltaQs) {
191         if (dq == DeltaQ()) {
192             continue;
193         }
194         nonEmpty.push_back(dq);
195         commonBinWidth = std::max(commonBinWidth, dq.getBinWidth());
196     }
197
198     if (nonEmpty.empty()) {
199         return DeltaQ();
200     }
201
202     for (auto &dq : nonEmpty) {
203         dq = rebin(dq, commonBinWidth);
204     }
205
206     const int largestSize = chooseLongestDeltaQSize(nonEmpty);
207     std::vector<double> resultingCdf;
208
209     for (int i = 0; i < largestSize; ++i) {
```

```
210        double sumAtI = 0;
211        double productAtI = 1;
212        for (const auto &dq : nonEmpty) {
213            const double cdfAtI = dq.cdfAt(i);
214            sumAtI += cdfAtI;
215            productAtI *= cdfAtI;
216        }
217        resultingCdf.push_back(sumAtI - productAtI);
218    }
219    return {commonBinWidth, resultingCdf, false};
220 }
221
222 DeltaQ allToFinish(const std::vector<DeltaQ> &deltaQs)
223 {
224     std::vector<DeltaQ> nonEmpty;
225     double commonBinWidth = 0.0;
226
227     for (const auto &dq : deltaQs) {
228         if (dq == DeltaQ()) {
229             continue;
230         }
231         nonEmpty.push_back(dq);
232         commonBinWidth = std::max(commonBinWidth, dq.getBinWidth());
233     }
234
235     if (nonEmpty.empty()) {
236         return DeltaQ();
237     }
238
239     for (auto &dq : nonEmpty) {
240         dq = rebin(dq, commonBinWidth);
241     }
242
243     DeltaQ result = nonEmpty[0];
244     for (size_t i = 1; i < nonEmpty.size(); ++i) {
245         result = result * nonEmpty[i];
246     }
247
248     return result;
249 }
250
251 int chooseLongestDeltaQSize(const std::vector<DeltaQ> &deltaQs)
252 {
253     int highestSize = 0;
254     for (const DeltaQ &deltaQ : deltaQs) {
255         if (deltaQ.getBins() > highestSize) {
256             highestSize = deltaQ.getBins();
257         }
258     }
259     return highestSize;
260 }
```

147

C.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) {
```

```
50          int toObserved = observedSize - newSize;
51          auto endIt = std::next(observedDeltaQs.begin(), toObserved);
52          observedDeltaQs.erase(observedDeltaQs.begin(), endIt);
53      }
54
55      if (calculatedSize > newSize) {
56          int toCalculated = calculatedSize - newSize;
57          auto endItC = std::next(calculatedDeltaQs.begin(),
    toCalculated);
58          calculatedDeltaQs.erase(calculatedDeltaQs.begin(), endItC);
59      }
60
61      if (QTAs.size() > newSize) {
62          int toSize = QTAs.size() - newSize;
63          auto endIt = std::next(QTAs.begin(), toSize);
64          QTAs.erase(QTAs.begin(), endIt);
65      }
66 }
67 void Snapshot::addQTA(uint64_t time, const QTA &qta)
68 {
69      QTAs[time] = qta;
70 }
71
72 std::optional<DeltaQRepr> Snapshot::getObservedDeltaQAtTime(std::
    uint64_t time)
73 {
74      auto it = observedDeltaQs.find(time);
75      if (it != observedDeltaQs.end()) {
76          return it->second;
77      }
78      return std::nullopt;
79 }
80
81 std::optional<DeltaQRepr> Snapshot::getCalculatedDeltaQAtTime(std::
    uint64_t time)
82 {
83      auto it = calculatedDeltaQs.find(time);
84      if (it != calculatedDeltaQs.end()) {
85          return it->second;
86      }
87      return std::nullopt;
88 }
89
90 std::size_t Snapshot::getObservedSize() const
91 {
92      return observedDeltaQs.size();
93 }
94
95 std::size_t Snapshot::getCalculatedSize() const
96 {
97      return calculatedDeltaQs.size();
98 }
99
100 std::vector<DeltaQRepr> Snapshot::getObservedDeltaQs() const &
101 {
102      auto obs = std::vector<DeltaQRepr>();
```

```
103      for (const auto &s : observedDeltaQs)
104          obs.emplace_back(s.second);
105      return obs;
106  }
107
108  std::vector<DeltaQRepr> Snapshot::getCalculatedDeltaQs() const &
109  {
110      auto calc = std::vector<DeltaQRepr>();
111      for (const auto &s : calculatedDeltaQs)
112          calc.emplace_back(s.second);
113      return calc;
114  }
115
116  std::vector<QTA> Snapshot::getQTAs() const &
117  {
118      auto QTAsVec = std::vector<QTA>();
119      for (const auto &q : QTAs) {
120          QTAsVec.emplace_back(q.second);
121      }
122      return QTAsVec;
123  }
124
125  void Snapshot::setName(const std::string &name)
126  {
127      observableName = name;
128  }
129
130  std::string Snapshot::getName() &
131  {
132      return observableName;
133  }
```

C.4.5   TriggerManager.cpp

This class is the manager of triggers for a probe. It can add/remove/evaluate triggers.

```
1  #include "TriggerManager.h"
2
3  #include <algorithm>
4
5  TriggerManager::Trigger::Trigger(TriggerType t, TriggerDefs::Condition
       c, TriggerDefs::Action a, bool e)
6      : type(t)
7      , condition(std::move(c))
8      , action(std::move(a))
9      , enabled(e)
10  {
11  }
12
13  void TriggerManager::addTrigger(TriggerType type, TriggerDefs::
       Condition condition, TriggerDefs::Action action, bool enabled)
14  {
15      triggers_.emplace_back(type, std::move(condition), std::move(
       action), enabled);
16  }
```

```cpp
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;
}
```

C.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.

```cpp
#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) { };
    }
}
}
```

C.5 parser

C.5.1 SystemBuilder.cpp

This class builds a new outcome diagram (system class) given an AST biult when parsing.

```cpp
#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;
}

```

```
47  std::any SystemBuilderVisitor::visitStart(parser::DQGrammarParser::
        StartContext *context)
48  {
49      for (const auto definition : context->definition()) {
50          visitDefinition(definition);
51      }
52
53      if (context->system()) {
54          visitSystem(context->system());
55      }
56
57      system.setOutcomes(outcomes);
58      system.setProbes(probes);
59      system.setOperators(operators);
60      /*
61      for (const auto &[name, link] : definitionLinks) {
62          std::cout << name << " [ ";
63          for (auto &name2 : link) {
64              std::cout << name2 << " ";
65          }
66          std::cout << "]\n";
67      }
68
69      for (const auto &[name, op] : operatorLinks) {
70          std::cout << name << " [ ";
71          for (auto link : op) {
72              std::cout << " [";
73              for (auto lill : link) {
74                  std::cout << lill << " ";
75              }
76              std::cout << "]";
77          }
78          std::cout << "]\n";
79      }
80      */
81      checkForCycles();
82      return nullptr;
83  }
84
85  std::any SystemBuilderVisitor::visitDefinition(parser::DQGrammarParser
        ::DefinitionContext *context)
86  {
87      std::string probeName = context->IDENTIFIER()->getText();
88      if (std::find(definedProbes.begin(), definedProbes.end(),
        probeName) != definedProbes.end()) {
89          throw std::invalid_argument("Probe has already been defined");
90      }
91      if (allNames.find(probeName) != allNames.end()) {
92          throw std::invalid_argument("Duplicate name detected: " +
        probeName);
93      }
94      allNames.insert(probeName);
95      currentlyBuildingProbe = probeName;
96
97      const auto chainComponents = std::any_cast<std::vector<std::::
        shared_ptr<Observable>>>(visitComponent_chain(context->
```

```
      component_chain ()));
 98    std::vector<std::shared_ptr<Observable>> probeCausalLinks;
 99    std::vector<std::string> links;
100    for (auto &comp : chainComponents) {
101        probeCausalLinks.push_back(comp);
102        links.push_back(comp->getName());
103    }
104
105    const auto probe = std::make_shared<Probe>(probeName,
      probeCausalLinks);
106
107    probes[probeName] = probe;
108    definedProbes.push_back(probeName);
109    definitionLinks[probeName] = links;
110    currentlyBuildingProbe = "";
111    return nullptr;
112 }
113
114 std::any SystemBuilderVisitor::visitSystem(parser::DQGrammarParser::
      SystemContext *context)
115 {
116    const auto chainComponents = std::any_cast<std::vector<std::
      shared_ptr<Observable>>>(visitComponent_chain(context->
      component_chain ()));
117
118    std::vector<std::string> links;
119    for (auto &comp : chainComponents) {
120        links.push_back(comp->getName());
121    }
122
123    systemLinks = links;
124
125    return nullptr;
126 }
127
128 std::any SystemBuilderVisitor::visitComponent(parser::DQGrammarParser
      ::ComponentContext *context)
129 {
130    if (context->behaviorComponent ()) {
131        return visitBehaviorComponent(context->behaviorComponent ());
132    } else if (context->probeComponent ()) {
133        return visitProbeComponent(context->probeComponent ());
134    } else if (context->outcome ()) {
135        return visitOutcome(context->outcome ());
136    }
137    return nullptr;
138 }
139
140 std::any SystemBuilderVisitor::visitBehaviorComponent(parser::
      DQGrammarParser::BehaviorComponentContext *context)
141 {
142    const std::string typeStr = context->BEHAVIOR_TYPE ()->getText();
143    std::string name = context->IDENTIFIER ()->getText();
144
145    if (operators.find(name) != operators.end()) {
146        return std::dynamic_pointer_cast<Observable>(operators[name]);
```

```
147         }
148         if (allNames.find(name) != allNames.end()) {
149             throw std::invalid_argument("Duplicate name detected: " + name
    );
150         }
151
152         allNames.insert(name);
153         OperatorType type;
154         if (typeStr == "a")
155             type = OperatorType::ATF;
156         else if (typeStr == "f")
157             type = OperatorType::FTF;
158         else if (typeStr == "p")
159             type = OperatorType::PRB;
160         else
161             throw std::invalid_argument("Unknown operator type: " +
    typeStr);
162
163         const auto op = std::make_shared<Operator>(name, type);
164         if (type == OperatorType::PRB && context->probability_list()) {
165             const auto probabilities = std::any_cast<std::vector<double>>(
    visitProbability_list(context->probability_list()));
166             op->setProbabilities(probabilities);
167         } else if (type == OperatorType::PRB && !context->probability_list
    ()) {
168             throw std::invalid_argument("A probabilistic operator must
    have probabilities");
169         } else if (type != OperatorType::PRB && context->probability_list
    ()) {
170             throw std::invalid_argument("A non probabilistic operator
    cannot have probabilities");
171         }
172
173         std::vector<std::vector<std::shared_ptr<Observable>>>
    operatorPtrLinks;
174
175         if (context->component_list()) {
176             auto childrenChains = std::any_cast<std::vector<std::vector<
    std::shared_ptr<Observable>>>>(visitComponent_list(context->
    component_list()));
177
178             for (auto &chain : childrenChains) {
179                 if (!chain.empty()) {
180                     operatorPtrLinks.push_back(chain);
181                 }
182             }
183         }
184
185         op->setCausalLinks(operatorPtrLinks);
186
187         if (context->component_list()) {
188             auto childrenChains = std::any_cast<std::vector<std::vector<
    std::shared_ptr<Observable>>>>(visitComponent_list(context->
    component_list()));
189
190             std::vector<std::vector<std::string>> childrenLinks;
```

```
191
192         for (auto &chain : childrenChains) {
193             if (!chain.empty()) {
194                 std::vector<std::string> chainNames;
195                 for (auto &comp : chain) {
196                     chainNames.push_back(comp->getName());
197                 }
198                 childrenLinks.push_back(chainNames);
199             }
200         }
201
202         operatorLinks[name] = childrenLinks;
203     }
204
205     operators[name] = op;
206     return std::dynamic_pointer_cast<Observable>(op);
207 }
208 std::any SystemBuilderVisitor::visitProbeComponent(parser::
     DQGrammarParser::ProbeComponentContext *context)
209 {
210     std::string name = context->IDENTIFIER()->getText();
211
212     if (!currentlyBuildingProbe.empty()) {
213         dependencies[currentlyBuildingProbe].push_back(name);
214     }
215
216     if (probes.find(name) != probes.end()) {
217         return std::dynamic_pointer_cast<Observable>(probes[name]);
218     }
219
220     // Create a stub probe (may be fleshed out later)
221     auto probe = std::make_shared<Probe>(name);
222     probes[name] = probe;
223     return std::dynamic_pointer_cast<Observable>(probe);
224 }
225
226 std::any SystemBuilderVisitor::visitProbability_list(parser::
     DQGrammarParser::Probability_listContext *context)
227 {
228     std::locale::global(std::locale("C"));
229
230     std::vector<double> probabilities;
231     for (auto num : context->NUMBER()) {
232         probabilities.push_back(std::stod(num->getText()));
233     }
234     return probabilities;
235 }
236
237 std::any SystemBuilderVisitor::visitComponent_list(parser::
     DQGrammarParser::Component_listContext *context)
238 {
239     std::vector<std::vector<std::shared_ptr<Observable>>>
     componentsChains;
240
241     for (auto chainCtx : context->component_chain()) {
242         auto chain = std::any_cast<std::vector<std::shared_ptr<
```

```
          Observable >>>( visitComponent_chain ( chainCtx ));
243           componentsChains.push_back ( chain );
244       }
245
246       return componentsChains;
247  }
248
249  std::any SystemBuilderVisitor::visitComponent_chain ( parser::
         DQGrammarParser::Component_chainContext *context )
250  {
251       std::vector<std::shared_ptr<Observable>> components;
252
253       for (auto compCtx : context->component()) {
254           auto component = std::any_cast<std::shared_ptr<Observable>>(
         visitComponent(compCtx));
255           components.push_back (component);
256       }
257       return components;
258  }
259
260  std::any SystemBuilderVisitor::visitOutcome ( parser::DQGrammarParser::
         OutcomeContext *context )
261  {
262       std::string name = context->IDENTIFIER()->getText();
263
264       if (outcomes.find(name) != outcomes.end()) {
265           return std::dynamic_pointer_cast<Observable>(outcomes[name]);
266       }
267       if (allNames.find(name) != allNames.end()) {
268           throw std::invalid_argument("Duplicate name detected: " + name
         );
269       }
270       allNames.insert(name);
271
272       auto outcome = std::make_shared<Outcome>(name);
273       outcomes[name] = outcome;
274       return std::dynamic_pointer_cast<Observable>(outcome);
275  }
```

C.5.2 SystemParserInterface.cpp

This class is an interface 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.

```
1  #include "SystemParserInterface.h"
2  #include "SystemErrorListener.h"
3  #include <exception>
4  #include <fstream>
5  #include <iostream>
6  #include <sstream>
7  #include <stdexcept>
8
9  /**
10  * @brief Parses a system definition from a file.
11  * @param filename Path to the file containing system definition.
```

```cpp
12  * @return Optional containing the parsed System if successful ,
       nullopt on error .
13  * @throws std :: invalid_argument if parsing fails .
14  */
15 std :: optional < System > SystemParserInterface :: parseFile ( const std ::
       string & filename )
16 {
17     std :: ifstream file ( filename );
18     if (! file ) {
19         std :: cerr << "Error: Could not open file: " << filename << std
       :: endl ;
20         return std :: nullopt ;
21     }
22
23     // Read entire file content
24     std :: stringstream buffer ;
25     buffer << file . rdbuf ();
26     std :: string content = buffer . str ();
27
28     antlr4 :: ANTLRInputStream input ( content );
29     try {
30         return parseInternal ( input );
31     } catch ( std :: exception & e ) {
32         throw std :: invalid_argument ( e . what ());
33     }
34 }
35
36 /**
37  * @brief Parses a system definition from a string .
38  * @param inputStr String containing system definition .
39  * @return Optional containing the parsed System if successful ,
       nullopt on error .
40  * @throws std :: invalid_argument if parsing fails .
41  */
42 std :: optional < System > SystemParserInterface :: parseString ( const std ::
       string & inputStr )
43 {
44     antlr4 :: ANTLRInputStream input ( inputStr );
45     try {
46         return parseInternal ( input );
47     } catch ( std :: exception & e ) {
48         throw std :: invalid_argument ( e . what ());
49     }
50 }
51
52 /**
53  * @brief Internal parsing implementation using ANTLR .
54  * @param input ANTLR input stream containing system definition .
55  * @return Optional containing the parsed System if successful ,
       nullopt on error .
56  * @throws std :: invalid_argument if parsing fails .
57  */
58 std :: optional < System > SystemParserInterface :: parseInternal ( antlr4 ::
       ANTLRInputStream & input )
59 {
60     // Initialize lexer and parser
```

```
61    parser::DQGrammarLexer lexer(&input);
62    antlr4::CommonTokenStream tokens(&lexer);
63    parser::DQGrammarParser parser(&tokens);
64
65    // Configure error handling
66    SystemErrorListener errorListener;
67    lexer.removeErrorListeners();
68    parser.removeErrorListeners();
69    lexer.addErrorListener(&errorListener);
70    parser.addErrorListener(&errorListener);
71
72    try {
73        // Parse and build system
74        auto tree = parser.start();
75        SystemBuilderVisitor visitor;
76        visitor.visitStart(tree);
77        return visitor.getSystem();
78    } catch (std::exception &e) {
79        throw std::invalid_argument(e.what());
80    }
81 }
```

C.6   server

C.6.1   Server.cpp

This class represents the server which receives and sends messages from Erlang.

```
1  #include "Server.h"
2  #include "../Application.h"
3  #include <arpa/inet.h>
4  #include <cstdint>
5  #include <cstring>
6  #include <fcntl.h>
7  #include <iostream>
8  #include <regex>
9  #include <signal.h>
10 #include <sys/socket.h>
11 #include <unistd.h>
12 #define TIMEOUT "to"
13 #define EXEC_OK "ok"
14 #define FAIL "fa"
15
16 /**
17  * @brief Constructs the Server and registers system observer.
18  * @param port The TCP port to listen on.
19  */
20 Server::Server(int port)
21     : port(port)
22     , server_fd(0)
23     , new_socket(0)
24     , server_started(false)
25 {
```

```
26      Application :: getInstance ().addObserver ([this]() { this ->
    updateSystem (); });
27
28      // Start worker thread (this can run independently)
29      workerThread = std :: thread ([this]() {
30          while (!shutdownWorker) {
31              std :: unique_lock lock ( queueMutex );
32              queueCond.wait (lock, [this] { return !sampleQueue.empty ()
    || shutdownWorker; });
33
34              while (!sampleQueue.empty ()) {
35                  auto [name, sample] = sampleQueue.front ();
36                  sampleQueue.pop ();
37                  lock.unlock ();
38
39                  if (system) {
40                      system ->addSample (name, sample );
41                  }
42
43                  lock.lock ();
44              }
45          }
46      });
47  }
48
49  /**
50   * @brief Destructor cleans up sockets and joins threads.
51   */
52  Server :: ~ Server ()
53  {
54      std :: lock_guard < std :: mutex > lock ( erlangMutex );
55      if ( erlang_socket > 0) {
56          close ( erlang_socket );
57          erlang_socket = -1;
58      }
59      close ( new_socket );
60      close ( server_fd );
61      if ( serverThread.joinable ())
62          serverThread.join ();
63  }
64  /**
65   * @brief Updates the system reference from Application.
66   */
67  void Server :: updateSystem ()
68  {
69      system = Application :: getInstance ().getSystem ();
70  }
71
72  /**
73   * @brief Main server loop handling client connections.
74   */
75  void Server :: run ()
76  {
77      server_fd = socket ( AF_INET, SOCK_STREAM, 0);
78      if ( server_fd == 0) {
79          perror (" Socket failed ");
```

```cpp
80         return;
81     }
82
83     // Configure socket options
84     int opt = 1;
85     setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt))
       ;
86     setsockopt(server_fd, SOL_SOCKET, SO_REUSEPORT, &opt, sizeof(opt))
       ;
87
88     // Set large buffer sizes
89     int bufSize = 1 << 20; // 1MB
90     setsockopt(server_fd, SOL_SOCKET, SO_RCVBUF, &bufSize, sizeof(
       bufSize));
91     setsockopt(server_fd, SOL_SOCKET, SO_SNDBUF, &bufSize, sizeof(
       bufSize));
92
93     // Bind socket to specified IP
94     address.sin_family = AF_INET;
95     address.sin_port = htons(port);
96
97     // Parse IP address
98     if (server_ip == "0.0.0.0" || server_ip.empty()) {
99         address.sin_addr.s_addr = INADDR_ANY;
100    } else {
101        if (inet_pton(AF_INET, server_ip.c_str(), &address.sin_addr)
       <= 0) {
102            std::cerr << "Invalid IP address: " << server_ip << std::
       endl;
103            close(server_fd);
104            return;
105        }
106    }
107
108    if (bind(server_fd, (struct sockaddr *)&address, sizeof(address))
       < 0) {
109        perror("Bind failed");
110        close(server_fd);
111        return;
112    }
113
114    // Set non-blocking mode
115    fcntl(server_fd, F_SETFL, O_NONBLOCK);
116
117    if (listen(server_fd, SOMAXCONN) < 0) {
118        perror("Listen failed");
119        close(server_fd);
120        return;
121    }
122
123    std::cout << "Server running on " << server_ip << ":" << port <<
       std::endl;
124    running = true;
125    server_started = true;
126
127    while (running) {
```

```
128          int addrlen = sizeof(address);
129          client_socket = accept(server_fd, (struct sockaddr *)&address,
     (socklen_t *)&addrlen);
130
131          if (client_socket < 0) {
132              if (errno == EWOULDBLOCK || errno == EAGAIN) {
133                  std::this_thread::sleep_for(std::chrono::milliseconds
     (100));
134                  cleanupThreads();
135                  continue;
136              }
137              if (!running)
138                  break; // Server was stopped
139              perror("Accept failed");
140              continue;
141          }
142
143          std::lock_guard<std::mutex> lock(clientsMutex);
144          clientThreads.emplace_back(&Server::handleClient, this,
     client_socket);
145      }
146
147      // Cleanup
148      close(server_fd);
149      server_fd = 0;
150      cleanupThreads();
151      server_started = false;
152 }
153
154 /**
155  * @brief Starts the server on specified IP and port.
156  * @param ip The IP address to bind to (default: "0.0.0.0" for all
     interfaces)
157  * @param port The port to listen on
158  * @return true if server started successfully
159  */
160 bool Server::startServer(const std::string &ip, int port)
161 {
162      if (server_started) {
163          std::cerr << "Server already running. Stop it first." << std::
     endl;
164          return false;
165      }
166
167      this->server_ip = ip;
168      this->port = port;
169
170      serverThread = std::thread(&Server::run, this);
171
172      // Wait a bit to see if server started successfully
173      std::this_thread::sleep_for(std::chrono::milliseconds(100));
174      return server_started;
175 }
176
177 /**
178  * @brief Stops the server and closes all sockets.
```

```cpp
179    */
180  void Server::stopServer()
181  {
182      if (!server_started) {
183          std::cout << "Server not running." << std::endl;
184          return;
185      }
186
187      running = false;
188      server_started = false;
189
190      // Close server socket to break accept loop
191      if (server_fd > 0) {
192          close(server_fd);
193          server_fd = 0;
194      }
195
196      // Join server thread
197      if (serverThread.joinable()) {
198          serverThread.join();
199      }
200
201      // Cleanup client threads
202      cleanupThreads();
203
204      std::cout << "Server stopped." << std::endl;
205  }
206
207  /**
208   * @brief Sets the Erlang endpoint for connections.
209   * @param ip Erlang server IP address
210   * @param port Erlang server port
211   * @return true if endpoint set successfully
212   */
213  bool Server::setErlangEndpoint(const std::string &ip, int port)
214  {
215      std::lock_guard<std::mutex> lock(erlangMutex);
216
217      // Close existing connection if any
218      if (erlang_socket > 0) {
219          close(erlang_socket);
220          erlang_socket = -1;
221      }
222
223      erlang_ip = ip;
224      erlang_port = port;
225
226      std::cout << "Erlang endpoint set to " << ip << ":" << port << std
      ::endl;
227      return true;
228  }
229
230  /**
231   * @brief Connects to the Erlang process.
232   * @return true if connection succeeded.
233   */
```

```cpp
234 bool Server::connectToErlang()
235 {
236     std::lock_guard<std::mutex> lock(erlangMutex);
237
238     if (erlang_socket > 0)
239         return true; // Already connected
240
241     erlang_socket = socket(AF_INET, SOCK_STREAM, 0);
242     if (erlang_socket < 0) {
243         perror("Erlang socket creation failed");
244         return false;
245     }
246
247     sockaddr_in erlang_addr {};
248     erlang_addr.sin_family = AF_INET;
249     erlang_addr.sin_port = htons(erlang_port);
250
251     if (inet_pton(AF_INET, erlang_ip.c_str(), &erlang_addr.sin_addr)
        <= 0) {
252         std::cerr << "Invalid Erlang IP: " << erlang_ip << std::endl;
253         close(erlang_socket);
254         erlang_socket = -1;
255         return false;
256     }
257
258     if (connect(erlang_socket, (struct sockaddr *)&erlang_addr, sizeof
        (erlang_addr)) < 0) {
259         perror("Failed to connect to Erlang");
260         close(erlang_socket);
261         erlang_socket = -1;
262         return false;
263     }
264
265     std::cout << "Connected to Erlang on " << erlang_ip << ":" <<
        erlang_port << std::endl;
266     return true;
267 }
268
269 /**
270  * @brief Sends a command to the Erlang process.
271  * @param command The command string to send.
272  */
273 void Server::sendToErlang(const std::string &command)
274 {
275     signal(SIGPIPE, SIG_IGN); // Ignore SIGPIPE to prevent crashes on
        disconnect
276
277     if (!connectToErlang()) {
278         std::cerr << "Unable to send to Erlang: not connected.\n";
279         return;
280     }
281
282     std::lock_guard<std::mutex> lock(erlangMutex);
283
284     std::string msgWithNewline = command + "\n";
285     ssize_t sent = send(erlang_socket, msgWithNewline.c_str(),
```

```
     msgWithNewline.size(), 0);

286
287     if (sent == -1) {
288         perror("send failed");

289
290         if (errno == EPIPE) {
291             std::cerr << "Broken pipe: Erlang side likely disconnected
    ." << std::endl;
292             close(erlang_socket);
293             erlang_socket = -1;
294         }

295
296         return;
297     }
298     std::cout << "Sent to Erlang: " << command << std::endl;
299 }

300
301 /**
302  * @brief Handles communication with a client.
303  * @param clientSocket The client socket file descriptor.
304  */
305 void Server::handleClient(int clientSocket)
306 {
307     std::string buffer;
308     char tempBuf[4096];

309
310     while (running) {
311         int valread = read(clientSocket, tempBuf, sizeof(tempBuf));
312         if (valread <= 0) {
313             if (valread == 0 || errno == ECONNRESET)
314                 break;
315             if (errno == EWOULDBLOCK || errno == EAGAIN) {
316                 std::this_thread::sleep_for(std::chrono::milliseconds
    (10));
317                 continue;
318             }
319             perror("Read failed");
320             break;
321         }

322
323         buffer.append(tempBuf, valread);

324
325         size_t pos;
326         size_t offset = 0;
327         while ((pos = buffer.find('\n', offset)) != std::string::npos)
    {
328             std::string_view message(buffer.data() + offset, pos -
    offset);
329             parseErlangMessage(message.data(), message.size());
330             offset = pos + 1;
331         }
332         buffer.erase(0, offset);
333     }

334
335     close(clientSocket);
336 }
```

```
337
338 /**
339  * @brief Cleans up finished client threads.
340  */
341 void Server::cleanupThreads()
342 {
343     std::lock_guard<std::mutex> lock(clientsMutex);
344     auto it = clientThreads.begin();
345     while (it != clientThreads.end()) {
346         if (it->joinable()) {
347             it->join();
348             it = clientThreads.erase(it);
349         } else {
350             ++it;
351         }
352     }
353 }
354
355 /**
356  * @brief Stops the server and worker threads.
357  */
358 void Server::stop()
359 {
360     running = false;
361     {
362         std::lock_guard lock(queueMutex);
363         shutdownWorker = true;
364     }
365     queueCond.notify_all();
366     if (workerThread.joinable())
367         workerThread.join();
368 }
369
370 /**
371  * @brief Parses messages from Erlang and adds samples to queue.
372  * @param buffer The message buffer.
373  * @param len Length of the message.
374  */
375 void Server::parseErlangMessage(const char *buffer, int len)
376 {
377     if (buffer == nullptr || len <= 0 || len >= 1024) {
378         std::cerr << "error" << std::endl;
379         return;
380     }
381
382     std::string message(buffer, len);
383
384     // Parse message components
385     size_t nPos = message.find("n:");
386     size_t bPos = message.find(";b:");
387     size_t ePos = message.find(";e:");
388     size_t sPos = message.find(";s:");
389
390     if (nPos != 0 || bPos == std::string::npos || ePos == std::string
    ::npos || sPos == std::string::npos) {
391         std::cerr << "Failed to parse message: " << message << std::
```

```
      endl;
392           return;
393       }
394
395       // Extract message fields
396       std::string name = message.substr(2, bPos - 2);
397       std::string bStr = message.substr(bPos + 3, ePos - (bPos + 3));
398       std::string eStr = message.substr(ePos + 3, sPos - (ePos + 3));
399       std::string statusStr = message.substr(sPos + 3);
400
401       // Convert to sample data
402       uint64_t startTime = std::stoull(bStr);
403       uint64_t endTime = std::stoull(eStr);
404       Sample sample;
405       Status status = Status::SUCCESS;
406
407       if (statusStr == TIMEOUT || statusStr == FAIL) {
408           status = (statusStr == TIMEOUT) ? Status::TIMEDOUT : Status::
      FAILED;
409       } else if (statusStr != EXEC_OK) {
410           std::cerr << "Unknown status: " << statusStr << std::endl;
411           return;
412       }
413
414       double long elapsed = (endTime - startTime) / 1'000'000'000.0L;
415       sample = {startTime, endTime, elapsed, status};
416
417       // Add to processing queue
418       {
419           std::lock_guard lock(queueMutex);
420           sampleQueue.emplace(name, sample);
421       }
422       queueCond.notify_one();
423 }
```

Appendix D

C++ Header Files

D.1 Root

D.1.1 Application.h

```
1
2 #pragma once
3 #include "diagram/Observable.h"
4 #include "diagram/System.h"
5 #include "server/Server.h"
6 #include <functional>
7 #include <iostream>
8 #include <memory>
9 #include <mutex>
10 #include <vector>
11
12 struct SystemDiff {
13     std::vector<std::string> addedProbes;
14     std::vector<std::string> removedProbes;
15     std::vector<std::string> changedProbes;
16
17     std::vector<std::string> addedOutcomes;
18     std::vector<std::string> removedOutcomes;
19     std::vector<std::string> changedOutcomes;
20
21     std::vector<std::string> addedOperators;
22     std::vector<std::string> removedOperators;
23     std::vector<std::string> changedOperators;
24 };
25
26 class Application
27 {
28
29     std::shared_ptr<System> system = nullptr;
30     std::vector<std::function<void()>> observers; // List of functions
        to notify
31     Server *server = nullptr;
32     bool componentsDiffer(const std::shared_ptr<Observable> &a, const
    std::shared_ptr<Observable> &b);
```

169

```
33
34     SystemDiff diffWith(System &newSystem);
35
36     Application();
37     void notifyObservers();
38
39 public:
40     static Application &getInstance();
41     void setServer(Server *);
42     void setSystem(System newSystem);
43     std::shared_ptr<System> getSystem();
44     void addObserver(std::function<void()> callback);
45     void sendDelayChange(std::string &, double);
46
47     bool startCppServer(const std::string &&, int);
48     void stopCppServer();
49
50     void setErlangEndpoint(const std::string &&, int);
51     void setStubRunning(bool running);
52 };
```

D.2   dashboard

D.2.1   ColorRegistry.h

```
1 #pragma once
2 #include <QColor>
3 #include <string>
4 #include <unordered_map>
5
6 class ColorRegistry
7 {
8 public:
9     static QColor getColorFor(const std::string &name);
10
11 private:
12     static QColor generateDistinctColor(int index);
13     static std::unordered_map<std::string, QColor> colorMap;
14 };
```

D.2.2   CustomLegendEntry.h

```
1
2
3 #pragma once
4
5 #include <QLabel>
6 #include <QWidget>
7 #include <qboxlayout.h>
8
9 class CustomLegendEntry : public QWidget
10 {
11     Q_OBJECT
```

```
12      QHBoxLayout *layout;
13      QLabel *colorBox;
14      QLabel *nameLabel;
15
16 public:
17      CustomLegendEntry(const QString &name, const QColor &color,
        QWidget *parent = nullptr);
18 };
```

D.2.3   CustomLegendPanel.h

```
1  #pragma once
2
3  #include <QVBoxLayout>
4  #include <QWidget>
5  #include <map>
6  #include <qscrollarea.h>
7
8  /**
9   * @class CustomLegendPanel
10  * @brief A scrollable widget that displays a legend for a plot.
11  */
12 class CustomLegendPanel : public QWidget
13 {
14     Q_OBJECT
15
16 public:
17     explicit CustomLegendPanel(QWidget *parent = nullptr);
18
19     /**
20      * @brief Adds a new entry to the legend.
21      * @param name The display name for the entry.
22      * @param color The color for the entry.
23      */
24     void addEntry(const QString &name, const QColor &color);
25
26     /**
27      * @brief Removes an entry from the legend by name.
28      * @param name The name of the entry to remove.
29      */
30     void removeEntry(const QString &name);
31
32     /**
33      * @brief Clears all entries from the legend.
34      */
35     void clear();
36
37 private:
38     std::map<QString, QWidget *> legendEntries;  ///< Map of legend
     entries by name
39     QVBoxLayout *mainLayout;                      ///< Main layout of
     the panel
40     QVBoxLayout *legendLayout;                    ///< Layout containing
      the legend entries
```

```
41      QScrollArea *scrollArea;                    ///< Scroll area for
    the legend content
42      QWidget *scrollContent;                     ///< Widget containing
     the scrollable content
43 };
```

D.2.4   DQPlotController.h

```
1
2  #ifndef DQPLOTCONTROLLER_H
3  #define DQPLOTCONTROLLER_H
4
5  #include <qlineseries.h>
6  #pragma once
7
8  // Project includes
9  #include "../diagram/System.h"
10 #include "DeltaQPlot.h"
11
12 // Qt includes
13 #include <QString>
14 #include <QtCharts/QLineSeries>
15
16 // C++ includes
17 #include <map>
18 #include <memory>
19 #include <string>
20 #include <vector>
21
22 // All series pertaining to an outcome
23 struct OutcomeSeries {
24      QLineSeries *outcomeS;
25      QLineSeries *lowerBoundS;
26      QLineSeries *upperBoundS;
27      QLineSeries *meanS;
28      QLineSeries *qtaS;
29 };
30
31 // All series pertaining to a probe
32 struct ExpressionSeries {
33      QLineSeries *obsS;
34      QLineSeries *obsLowerBoundS;
35      QLineSeries *obsUpperBoundS;
36      QLineSeries *obsMeanS;
37      QLineSeries *calcS;
38      QLineSeries *calcLowerBoundS;
39      QLineSeries *calcUpperBoundS;
40      QLineSeries *calcMeanS;
41      QLineSeries *qtaS;
42 };
43
44 class DeltaQPlot;
45
46 /**
47  * @class DQPlotController
```

```
48  * @brief Controls the data management and update logic of DeltaQPlot.
49  */
50 class DQPlotController
51 {
52 public:
53     /**
54      * @brief Constructor with associated plot and selected components
     .
55      */
56     DQPlotController(DeltaQPlot *plot, const std::vector<std::string>
    &selectedItems);
57
58     /**
59      * @brief Destructor.
60      */
61     ~DQPlotController();
62
63     /**
64      * @brief Checks if a component is already being plotted.
65      */
66     bool containsComponent(std::string name);
67
68     /**
69      * @brief Updates the plot according to a new list of selected
    components.
70      */
71     void editPlot(const std::vector<std::string> &selectedItems);
72
73     /**
74      * @brief Adds a new component (probe or outcome) to the plot.
75      */
76     void addComponent(const std::string &name, bool isOutcome);
77
78     QLineSeries *createAndAddLineSeries(const std::string &legendName)
    ;
79
80     void addOutcomeSeries(const std::string &name);
81
82     void removeOutcomeSeries(const std::string &name);
83
84     void addExpressionSeries(const std::string &name, bool isProbe);
85
86     void removeExpressionSeries(const std::string &name, bool isProbe)
    ;
87     /**
88      * @brief Returns a list of names of all currently plotted
    components.
89      */
90     std::vector<std::string> getComponents();
91     /**
92      * @brief Removes a plotted component by name (const reference).
93      */
94     void removeComponent(const std::string &name);
95
96     /**
```

```
 97      * @brief Updates the data of all series based on provided time
     range and bin width.
 98      */
 99     void update(uint64_t timeLowerBound, uint64_t timeUpperBound);
100
101     void setTitle();
102
103     bool isEmptyAfterReset();
104
105 private:
106     double updateOutcome(OutcomeSeries &, const std::shared_ptr<
     Outcome> &, uint64_t, uint64_t);
107
108     double updateProbe(ExpressionSeries &, std::shared_ptr<Probe> &,
     uint64_t, uint64_t);
109
110     double updateOperator(ExpressionSeries &, std::shared_ptr<Operator
     > &, uint64_t, uint64_t);
111
112     void updateExpression(ExpressionSeries &, DeltaQRepr &&,
     DeltaQRepr &&, QTA &&, double maxDelay);
113
114     DeltaQPlot *plot;
115
116     std::mutex updateMutex;
117     std::mutex resetMutex;
118
119     std::map<std::string, std::pair<OutcomeSeries, std::shared_ptr<
     Outcome>>> outcomes;
120     std::map<std::string, std::pair<ExpressionSeries, std::shared_ptr<
     Probe>>> probes;
121     std::map<std::string, std::pair<ExpressionSeries, std::shared_ptr<
     Operator>>> operators;
122 };
123
124 #endif // DQPLOTCONTROLLER_H
```

D.2.5   DQPlotList.h

```
 1 #pragma once
 2
 3 #ifndef DQ_PLOT_LIST_H
 4 #define DQ_PLOT_LIST_H
 5
 6 #include "DQPlotController.h"
 7 #include <QCheckBox>
 8 #include <QListWidget>
 9 #include <QPushButton>
10 #include <QVBoxLayout>
11 #include <QWidget>
12
13 class DQPlotController;
14
15 /**
16  * @class DQPlotList
```

```
17   * @brief A widget that displays and manages lists of available and
         selected plot components for a selected plot.
18   *
19   */
20  class DQPlotList : public QWidget
21  {
22      Q_OBJECT
23
24  public:
25      explicit DQPlotList(DQPlotController *controller, QWidget *parent
     = nullptr);
26
27      /**
28       * @brief Resets the widget's state.
29       */
30      void reset();
31
32      /**
33       * @brief Checks if the widget is empty after reset.
34       * @return true if no components are selected, false otherwise.
35       */
36      bool isEmptyAfterReset();
37
38      /**
39       * @brief Updates both the available and selected lists.
40       */
41      void updateLists();
42
43      /**
44       * @brief Default destructor.
45       */
46      ~DQPlotList() = default;
47
48  private Q_SLOTS:
49      /**
50       * @brief Handles confirmation of selected components to add.
51       */
52      void onConfirmSelection();
53
54      /**
55       * @brief Handles removal of selected components.
56       */
57      void onRemoveSelection();
58
59  private:
60      /**
61       * @brief Adds an item to either the available or selected list.
62       * @param name The name of the component.
63       * @param isSelected Whether the item should be in the selected
     list.
64       * @param category The category for the item (used for grouping).
65       */
66      void addItemToList(const std::string &name, bool isSelected, const
     QString &category);
67
```

175

```
68     DQPlotController *controller; ///< The controller managing plot
     components.
69
70     QListWidget *selectedList;     ///< List widget showing currently
     selected components.
71     QListWidget *availableList;    ///< List widget showing available
     components.
72
73     QPushButton *addButton;        ///< Button to add selected
     components.
74     QPushButton *removeButton;     ///< Button to remove selected
     components.
75 };
76
77 #endif // DQ_PLOT_LIST_H
```

D.2.6   DelaySettingsWidget.h

```
1 #pragma once
2
3 #include <QComboBox>
4 #include <QHBoxLayout>
5 #include <QLabel>
6 #include <QPushButton>
7 #include <QSlider>
8 #include <QSpinBox>
9 #include <QVBoxLayout>
10 #include <QWidget>
11 #include <cmath>
12
13 /**
14  * @brief A widget for configuring delay parameters for a selected
     observable.
15  */
16 class DelaySettingsWidget : public QWidget
17 {
18     Q_OBJECT
19
20 public:
21     explicit DelaySettingsWidget(QWidget *parent = nullptr);
22
23     /**
24      * @brief Populates the observable combo box using the system's
     observable list.
25      */
26     void populateComboBox();
27
28     /**
29      * @brief Computes the maximum delay in milliseconds based on the
     slider and spinbox values.
30      * @return Maximum delay in milliseconds.
31      */
32     double getMaxDelayMs() const;
33
34 Q_SIGNALS:
```

```cpp
35      /**
36       * @brief Signal emitted when delay parameters have been changed
     and saved.
37       */
38      void delayParametersChanged();
39
40 private Q_SLOTS:
41      /**
42       * @brief Updates the label showing the current maximum delay
     based on UI values.
43       */
44      void updateMaxDelay();
45
46      /**
47       * @brief Handles saving the currently selected delay parameters
     to the system.
48       */
49      void onSaveDelayClicked();
50
51      /**
52       * @brief Loads the saved settings for the currently selected
     observable.
53       */
54      void loadObservableSettings();
55
56 private:
57      QVBoxLayout *mainLayout;            ///< Main layout container.
58      QLabel *settingsLabel;             ///< Label describing the
     purpose of the widget.
59      QLabel *maxDelayLabel;             ///< Label showing the computed
     max delay.
60
61      QHBoxLayout *settingsLayout;       ///< Layout for parameter
     controls.
62      QComboBox *observableComboBox;     ///< Combo box for selecting
     observables.
63      QSlider *delaySlider;              ///< Slider to set delay
     exponent.
64      QSpinBox *binSpinBox;              ///< Spin box to set number of
     bins.
65
66      QPushButton *saveDelayButton;      ///< Button to save the delay
     configuration.
67 };
```

D.2.7   DeltaQPlot.h

```cpp
1
2 #ifndef DELTAQPLOT_H
3 #define DELTAQPLOT_H
4
5 #include "CustomLegendPanel.h"
6 #include <qboxlayout.h>
7 #pragma once
8
```

```
 9  // Qt includes
10  #include <QChartView>
11  #include <QLineSeries>
12  #include <QToolButton>
13  #include <QValueAxis>
14  // C++ includes
15  #include <string>
16  #include <vector>
17
18  class DQPlotController;
19  class DQPlotList;
20
21  /**
22   * @class DeltaQPlot
23   * @brief A class representing a DeltaQ chart view that allows
          visualization of probes over time.
24   */
25  class DeltaQPlot : public QWidget
26  {
27      Q_OBJECT
28
29  public:
30      /**
31       * @brief Constructs a DeltaQPlot with selected components.
32       * @param selectedItems List of selected component names.
33       * @param parent Parent widget.
34       */
35      explicit DeltaQPlot(const std::vector<std::string> &selectedItems,
        QWidget *parent = nullptr);
36
37      /**
38       * @brief Destructor.
39       */
40      ~DeltaQPlot();
41
42      /**
43       * @brief Adds a QLineSeries to the chart.
44       * @param series Pointer to the line series.
45       * @param name Name of the series.
46       */
47      void addSeries(QLineSeries *series, const std::string &name);
48
49      /**
50       * @brief Updates the plot data using provided time range and bin
        width.
51       */
52      void update(uint64_t timeLowerBound, uint64_t timeUpperBound);
53
54      /**
55       * @brief Removes a series from the chart.
56       * @param series Series to remove.
57       */
58      void removeSeries(QAbstractSeries *series);
59
60      /**
61       * @brief Updates plot with new set of selected components.
```

```
 62       */
 63      void editPlot(const std::vector<std::string> &selectedItems);
 64
 65      /**
 66       * @brief Gets list of currently plotted component names.
 67       */
 68      std::vector<std::string> getComponents();
 69
 70      bool isEmptyAfterReset();
 71
 72      /**
 73       * @brief Updates an existing series with new data points.
 74       */
 75      void updateSeries(QLineSeries *series, const QVector<QPointF> &
     data);
 76
 77      void updateXRange(double xRange);
 78      /**
 79       * @brief Returns the associated plot list.
 80       */
 81      DQPlotList *getPlotList();
 82
 83      void setTitle(QString &&);
 84
 85 protected:
 86      /**
 87       * @brief Handles mouse press events on the chart.
 88       */
 89      void mousePressEvent(QMouseEvent *event) override;
 90
 91 Q_SIGNALS:
 92      /**
 93       * @brief Emitted whenout
 94      QHBox this plot is selected by the user.
 95       */
 96      void plotSelected(DeltaQPlot *plot);
 97
 98 private:
 99      QHBoxLayout *layout;
100      QToolButton *toggleButton;
101      QChartView *chartView;
102      QChart *chart;
103
104      QValueAxis *axisX;
105      QValueAxis *axisY;
106
107      QLineSeries *operationSeries;
108      DQPlotController *controller;
109      DQPlotList *plotList;
110
111      CustomLegendPanel *legendPanel;
112 };
113
114 #endif // DELTAQPLOT_H
```

179

D.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
```

```
48     QMap<DeltaQPlot *, QWidget *> plotContainers; ///< Map of plots to
       their containers
49     uint64_t timeLowerBound;        ///< Lower time bound for data
     updates
50
51     std::mutex plotDelMutex;        ///< Mutex for plot deletion
     safety
52     std::mutex updateMutex;         ///< Mutex for update operations
53
54     int pollingRate {200};          ///< Current polling rate in
     milliseconds
55
56 public:
57     MainWindow(QWidget *parent = nullptr);
58
59     ~MainWindow();
60
61     /**
62      * @brief Resets the window state, cleaning up empty plots.
63      */
64     void reset();
65
66 private Q_SLOTS:
67     /**
68      * @brief Updates all plots with new data.
69      */
70     void updatePlots();
71
72     /**
73      * @brief Handles adding new plots from sidebar selection.
74      */
75     void onAddPlotClicked();
76
77     /**
78      * @brief Removes a specific plot.
79      * @param plot The plot to remove.
80      */
81     void onRemovePlot(DeltaQPlot *plot);
82
83     /**
84      * @brief Handles plot selection changes.
85      * @param plot The newly selected plot.
86      */
87     void onPlotSelected(DeltaQPlot *plot);
88
89 protected:
90     /**
91      * @brief Handles context menu events for plot management.
92      * @param event The context menu event.
93      */
94     void contextMenuEvent(QContextMenuEvent *event) override;
95
96     /**
97      * @brief Handles window resize events to adjust plot sizes.
98      * @param event The resize event.
99      */
```

```
100    void resizeEvent(QResizeEvent *event) override;
101 };
```

D.2.9   NewPlotList.h

```cpp
1  #pragma once
2
3  #ifndef NEW_PLOT_LIST_H
4  #define NEW_PLOT_LIST_H
5
6  #include "../diagram/System.h"
7  #include <QCheckBox>
8  #include <QListWidget>
9  #include <qlistwidget.h>
10 #include <qwidget.h>
11
12 /**
13  * @brief A list widget for selecting observables to create a new plot
     .
14  *
15  */
16 class NewPlotList : public QListWidget
17 {
18     Q_OBJECT
19
20 public:
21     explicit NewPlotList(QWidget *parent = nullptr);
22
23     /**
24      * @brief Gets the names of all currently selected observables.
25      * @return A vector of strings representing selected observables.
26      */
27     std::vector<std::string> getSelectedItems();
28
29     /**
30      * @brief Deselects all currently selected items in the list.
31      */
32     void deselectAll();
33
34     /**
35      * @brief Clears the list and repopulates it with updated
     observables from the system.
36      */
37     void reset();
38
39 private:
40     /**
41      * @brief Adds observable items to the list from the current
     system.
42      */
43     void addItems();
44 };
45
46 #endif // NEW_PLOT_LIST_H
```

D.2.10   ObservableSettings.h

```cpp
#ifndef OBS_SETTINGS_H
#define OBS_SETTINGS_H

#include <qlabel.h>
#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
```

D.2.11   SamplingRateWidget.h

```cpp
#pragma once

#include <QLabel>
#include <QPushButton>
#include <QSlider>
#include <QVBoxLayout>
#include <QWidget>

/**
 * @class SamplingRateWidget
 * @brief A widget for selecting and saving polling rate intervals.
 *
 * Provides a slider interface to select from predefined polling 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:
```

```
24      /**
25       * @brief Handles slider value changes to update the displayed
         rate.
26       * @param value The current slider index (0-based).
27       */
28      void onSliderValueChanged(int value);
29
30      /**
31       * @brief Handles save button clicks to emit the selected rate.
32       */
33      void onSaveClicked();
34
35 Q_SIGNALS:
36      /**
37       * @brief Emitted when a new polling rate is saved.
38       * @param milliseconds The selected polling rate in milliseconds.
39       */
40      void onSamplingRateChanged(int milliseconds);
41
42 private:
43      QSlider *slider;              ///< Slider for selecting polling rate
44      QLabel *valueLabel;           ///< Displays the current polling rate
45      QPushButton *saveButton;      ///< Button to save the selected rate
46      QVector<int> pollingRates;    ///< Available polling rate options (
         ms)
47 };
```

D.2.12   QTAInputWidget.h

```
1 #ifndef QTAINPUTWIDGET_H
2 #define QTAINPUTWIDGET_H
3
4 #include <QWidget>
5 #include <QLineEdit>
6 #include <QComboBox>
7 #include <QFormLayout>
8 #include <QLabel>
9 #include <QPushButton>
10
11 /**
12  * @class QTAInputWidget
13  * @brief A Qt widget for configuring QTAs for observables.
14  */
15 class QTAInputWidget : public QWidget
16 {
17      Q_OBJECT
18
19 public:
20      explicit QTAInputWidget(QWidget *parent = nullptr);
21
22      /**
23       * @brief Gets the 25th percentile value (in seconds).
24       * @return The value entered in the 25th percentile field.
25       */
26      double getPerc25() const;
```

```
27
28      /**
29       * @brief Gets the 50th percentile (median) value (in seconds).
30       * @return The value entered in the 50th percentile field.
31       */
32      double getPerc50() const;
33
34      /**
35       * @brief Gets the 75th percentile value (in seconds).
36       * @return The value entered in the 75th percentile field.
37       */
38      double getPerc75() const;
39
40      /**
41       * @brief Gets the maximum allowed CDF value (0 to 1).
42       * @return The value entered in the CDF max field.
43       */
44      double getCdfMax() const;
45
46      /**
47       * @brief Gets the currently selected observable name.
48       * @return The name of the selected observable.
49       */
50      QString getSelectedObservable() const;
51
52 public Q_SLOTS:
53      /**
54       * @brief Populates the observable dropdown with available
     observables.
55       * @note Called automatically when the system updates.
56       */
57      void populateComboBox();
58
59      /**
60       * @brief Loads QTA settings for the selected observable into the
     UI fields.
61       * @note Triggered when the dropdown selection changes.
62       */
63      void loadObservableSettings();
64
65      /**
66       * @brief Saves the current QTA settings to the system.
67       * @note Called when the "Save" button is clicked.
68       * @throws std::exception if validation fails (e.g., invalid CDF
     value).
69       */
70      void onSaveButtonClicked();
71
72 private:
73      QComboBox *observableComboBox;    ///< Dropdown to select an
     observable (probe/outcome).
74      QLineEdit *perc25Edit;            ///< Input field for the 25th
     percentile (seconds).
75      QLineEdit *perc50Edit;            ///< Input field for the 50th
     percentile (seconds).
```

```
76      QLineEdit *perc75Edit;              ///< Input field for the 75th
     percentile (seconds).
77      QLineEdit *cdfMaxEdit;              ///< Input field for the max CDF
     value (0-1).
78      QPushButton *saveButton;           ///< Button to save QTA settings.
79      QLabel *qtaLabel;                  ///< Label describing the widget'
     s purpose.
80 };
81
82 #endif // QTAINPUTWIDGET_H
```

D.2.13   Sidebar.h

```
1  #ifndef SIDEBAR_H
2  #define SIDEBAR_H
3
4  #include "DQPlotList.h"
5  #include "NewPlotList.h"
6  #include "SamplingRateWidget.h"
7  #include "SystemCreationWidget.h"
8  #include <QComboBox>
9  #include <QLabel>
10 #include <QPushButton>
11 #include <QSpinBox>
12 #include <QSplitter>
13 #include <QTextEdit>
14 #include <QVBoxLayout>
15 #include <QWidget>
16 #include <qboxlayout.h>
17
18 /**
19  * @class Sidebar
20  * @brief Main sidebar widget containing plot management controls and
     system configuration.
21  *
22  * The Sidebar provides UI components for:
23  * - Creating new plots
24  * - Managing existing plots
25  * - System configuration
26  * - Polling rate adjustment
27  */
28 class Sidebar : public QWidget
29 {
30      Q_OBJECT
31
32      QVBoxLayout *newPlotListLayout;      ///< Layout for new plot
     selection components
33      QWidget *newPlotListWidget;          ///< Container widget for new
      plot controls
34      QLabel *newPlotLabel;                ///< Label for new plot
     section
35      NewPlotList *newPlotList;            ///< List widget for
     selecting probes for new plots
36      QPushButton *addNewPlotButton;       ///< Button to create new
     plot
```

```cpp
37
38     QWidget *currentPlotWidget;          ///< Container widget for
       current plot controls
39     QVBoxLayout *currentPlotLayout;      ///< Layout for current plot
       components
40     QLabel *currentPlotLabel;            ///< Label for current plot
       section
41     DQPlotList *currentPlotList = nullptr; ///< List widget for
       managing current plot's probes
42
43     QSplitter *mainSplitter;             ///< Main splitter organizing
       sections vertically
44     QVBoxLayout *layout;                 ///< Main layout of the
       sidebar
45
46     SystemCreationWidget *systemCreationWidget; ///< Widget for system
       creation/configuration
47     SamplingRateWidget *pollingRateWidget; ///< Widget for adjusting
       polling rate
48
49 Q_SIGNALS:
50     /**
51      * @brief Emitted when the "Add plot" button is clicked.
52      */
53     void addPlotClicked();
54
55     /**
56      * @brief Emitted when polling rate is changed.
57      * @param milliseconds The new polling rate in milliseconds.
58      */
59     void onSamplingRateChanged(int milliseconds);
60
61 private Q_SLOTS:
62     /**
63      * @brief Handles "Add plot" button click event.
64      */
65     void onAddPlotClicked();
66
67     /**
68      * @brief Handles polling rate change events.
69      * @param ms The new polling rate in milliseconds.
70      */
71     void handleSamplingRateChanged(int ms);
72
73 public:
74     /**
75      * @brief Constructs a Sidebar widget.
76      * @param parent The parent widget (optional).
77      */
78     explicit Sidebar(QWidget *parent = nullptr);
79
80     /**
81      * @brief Sets the current plot list widget.
82      * @param currentPlotList The DQPlotList widget to display.
83      */
84     void setCurrentPlotList(DQPlotList *currentPlotList);
```

```
85
86      /**
87       * @brief Hides the current plot management section.
88       */
89      void hideCurrentPlot();
90
91      /**
92       * @brief Gets the new plot list widget.
93       * @return Pointer to the NewPlotList widget.
94       */
95      NewPlotList *getPlotList() const
96      {
97          return newPlotList;
98      }
99
100     /**
101      * @brief Clears new plot selection after plot creation.
102      */
103     void clearOnAdd();
104 };
105
106 #endif
```

D.2.14   SnapshotViewerWindow.h

```
1  #pragma once
2
3  #include <QChartView>
4  #include <QComboBox>
5  #include <QLabel>
6  #include <QSlider>
7  #include <QWidget>
8
9  #include "../maths/Snapshot.h"
10 #include <map>
11
12 /**
13  * @brief A QWidget-based window for visualizing snapshots from fired
     triggers.
14  */
15 class SnapshotViewerWindow : public QWidget
16 {
17     Q_OBJECT
18
19 public:
20     explicit SnapshotViewerWindow(std::vector<Snapshot> &snapshotList,
       QWidget *parent = nullptr);
21
22     /**
23      * @brief Sets the snapshots to display in the viewer.
24      * @param snapshotList A vector of Snapshots.
25      */
26     void setSnapshots(std::vector<Snapshot> &snapshotList);
27
28 private Q_SLOTS:
```

```
29      /**
30       * @brief Slot triggered when the observable selection changes.
31       * @param name Name of the newly selected observable.
32       */
33      void onObservableChanged(const QString &name);
34
35      /**
36       * @brief Slot triggered when the time slider is moved.
37       * @param value Index of the snapshot in the selected observable.
38       */
39      void onTimeSliderChanged(int value);
40
41  private:
42      /**
43       * @brief Updates the chart view based on the current observable
       and time index.
44       */
45      void updatePlot();
46
47      QChartView *chartView;                      ///< Chart view for
       plotting the snapshot data.
48      QComboBox *observableSelector;              ///< Dropdown for
       selecting an observable.
49      QSlider *timeSlider;                        ///< Slider for
       selecting time index.
50      QLabel *timeLabel;                          ///< Label showing the
       current time.
51
52      std::map<std::string, Snapshot> snapshots;///< Map of observable
       name to snapshot.
53      std::string currentObservable;             ///< Currently selected
       observable.
54  };
```

D.2.15   StubControlWidget.h

```
1  #pragma once
2  #include "src/Application.h"
3  #include <QApplication>
4  #include <QGridLayout>
5  #include <QGroupBox>
6  #include <QHBoxLayout>
7  #include <QLabel>
8  #include <QLineEdit>
9  #include <QPushButton>
10 #include <QVBoxLayout>
11 #include <QWidget>
12
13 class StubControlWidget : public QWidget
14 {
15     Q_OBJECT
16 public:
17     StubControlWidget(QWidget *parent = nullptr);
18
19 private:
```

```
20      // Erlang controls
21      QPushButton *startErlangButton;
22      QPushButton *stopErlangButton;
23
24      // Server controls
25      QPushButton *startServerButton;
26      QPushButton *stopServerButton;
27      QLineEdit *serverIpEdit;
28      QLineEdit *serverPortEdit;
29
30      // Erlang receiver settings
31      QLineEdit *erlangReceiverIpEdit;
32      QLineEdit *erlangReceiverPortEdit;
33      QPushButton *setErlangEndpointButton;
34
35      QVBoxLayout *mainLayout;
36
37  private Q_SLOTS:
38      // Erlang slots
39      void onStartErlangClicked();
40      void onStopErlangClicked();
41
42      // Server slots
43      void onStartServerClicked();
44      void onStopServerClicked();
45
46      // Erlang endpoint slot
47      void onSetErlangEndpointClicked();
48  };
```

D.2.16   SystemCreationWidget.h

```
1  #ifndef SYSTEMCREATIONWIDGET_H
2  #define SYSTEMCREATIONWIDGET_H
3
4  #include <QHBoxLayout>
5  #include <QLabel>
6  #include <QPushButton>
7  #include <QTextEdit>
8  #include <QVBoxLayout>
9  #include <QWidget>
10
11  /**
12   * @class SystemCreationWidget
13   * @brief A QWidget that allows users to create, edit, load, and save
14      system definitions.
15   */
16  class SystemCreationWidget : public QWidget
17  {
18      Q_OBJECT
19
20  public:
21      explicit SystemCreationWidget(QWidget *parent = nullptr);
22
23      /**
```

```
23       * @brief Retrieves the current system text from the editor.
24       * @return The system text as a std::string.
25       */
26      std::string getSystemText() const;
27
28      /**
29       * @brief Sets the system text in the editor.
30       * @param text The new system definition text.
31       */
32      void setSystemText(const std::string &text);
33
34 Q_SIGNALS:
35      /**
36       * @brief Emitted when the system is successfully updated.
37       */
38      void systemUpdated();
39
40      /**
41       * @brief Emitted when the system is successfully saved.
42       */
43      void systemSaved();
44
45      /**
46       * @brief Emitted when a system is successfully loaded.
47       */
48      void systemLoaded();
49
50 private Q_SLOTS:
51      /**
52       * @brief Parses the text and updates the system instance.
53       */
54      void onUpdateSystem();
55
56      /**
57       * @brief Saves the current system text to a file.
58       */
59      void saveSystemTo();
60
61      /**
62       * @brief Loads a system from a file and updates the editor.
63       */
64      void loadSystem();
65
66 private:
67      QTextEdit *systemTextEdit;          ///< Editor widget for system
     text.
68      QPushButton *updateSystemButton;   ///< Button to update system.
69      QPushButton *saveSystemButton;     ///< Button to save system.
70      QPushButton *loadSystemButton;     ///< Button to load system.
71      QLabel *systemLabel;               ///< Label describing the
     editor.
72
73      QVBoxLayout *mainLayout;           ///< Layout for the main
     components.
74      QHBoxLayout *buttonLayout;         ///< Layout for the buttons.
75 };
```

```
76
77 #endif // SYSTEMCREATIONWIDGET_H
```

D.2.17 TriggersTab.h

```
1 #pragma once
2
3 #include <QCheckBox>
4 #include <QComboBox>
5 #include <QFormLayout>
6 #include <QListWidget>
7 #include <QMap>
8 #include <QSpinBox>
9 #include <QVBoxLayout>
10 #include <QWidget>
11
12 #include "../Application.h"
13 #include "src/maths/TriggerManager.h"
14
15 /**
16  * @class TriggersTab
17  * @brief Widget for managing and monitoring trigger conditions on
     observables.
18  *
19  * Provides UI for:
20  * - Setting up trigger conditions (sample limits, QTA violations)
21  * - Displaying triggered events
22  * - Viewing snapshots of triggered states
23  */
24 class TriggersTab : public QWidget
25 {
26     Q_OBJECT
27
28 public:
29     explicit TriggersTab(QWidget *parent = nullptr);
30
31     ~TriggersTab();
32
33     /**
34      * @brief Adds a triggered message to the display list.
35      * @param msg The message to display.
36      */
37     void addTriggeredMessage(const QString &msg);
38
39 private Q_SLOTS:
40     /**
41      * @brief Handles observable selection changes.
42      * @param name The newly selected observable name.
43      */
44     void onObservableChanged(const QString &name);
45
46     /**
47      * @brief Handles trigger condition changes.
48      */
49     void onTriggerChanged();
```

```
50
51     /**
52      * @brief Handles triggered item clicks to show snapshots.
53      * @param item The clicked list item.
54      */
55     void onTriggeredItemClicked(QListWidgetItem *item);
56
57 private:
58     /**
59      * @brief Captures snapshots when triggers are activated.
60      * @param time The timestamp of the trigger event.
61      * @param name The name of the observable that triggered.
62      */
63     void captureSnapshots(std::uint64_t time, const std::string &name)
       ;
64
65     /**
66      * @brief Populates the observable dropdown list.
67      */
68     void populateObservables();
69
70     /**
71      * @brief Updates checkbox states based on current triggers.
72      */
73     void updateCheckboxStates();
74
75     QVBoxLayout *mainLayout;              ///< Main vertical layout
76     QFormLayout *formLayout;             ///< Form layout for controls
77     QComboBox *observableComboBox;       ///< Dropdown for observable
       selection
78
79     QWidget *sampleLimitWidget;          ///< Container for sample
       limit controls
80     QHBoxLayout *sampleLimitLayout;      ///< Layout for sample limit
       controls
81     QCheckBox *sampleLimitCheckBox;      ///< Checkbox to enable
       sample limit trigger
82     QSpinBox *sampleLimitSpinBox;        ///< Spinbox for sample limit
        threshold
83
84     QCheckBox *qtaBoundsCheckBox;        ///< Checkbox for QTA bounds
       violation trigger
85
86     QListWidget *triggeredList;          ///< List widget for
       triggered events
87
88     /**
89      * @brief Gets the currently selected observable.
90      * @return Shared pointer to the current observable.
91      * @throws std::runtime_error if system or observable doesn't
       exist.
92      */
93     std::shared_ptr<Observable> getCurrentObservable();
94
95     static constexpr int sampleLimitThreshold = 500;      ///< Default
        sample limit threshold
```

```
96    static constexpr double failureRateThreshold = 0.95;  ///< Failure
      rate threshold constant
97 };
```

D.3   diagram

D.3.1   Observable.h

```
1  #pragma once
2
3  #include "../maths/Snapshot.h"
4  #include "Sample.h"
5  #include "src/maths/TriggerManager.h"
6  #include <deque>
7  #include <math.h>
8  #include <mutex>
9
10 #define DELTA_T_BASE 0.001
11 #define MAX_DQ 30
12 class Observable
13 {
14 protected:
15     std::string name;
16     std::deque<Sample> samples;
17     mutable bool sorted;
18
19     std::deque<DeltaQ> confidenceIntervalHistory;
20     double maxDelay {0.05};
21     int deltaTExp {0}; // Exponent for dynamic binning
22     int nBins {50}; // Number of bins
23
24     TriggerManager triggerManager;
25
26     ConfidenceInterval observedInterval;
27     QTA qta;
28
29     Snapshot observableSnapshot;
30
31     std::mutex observedMutex;
32     std::mutex samplesMutex;
33     std::mutex paramMutex;
34
35     bool recording = false;
36
37 private:
38     void updateSnapshot(uint64_t timeLowerBound, DeltaQ &deltaQ);
39
40 public:
41     /**
42      * @brief Constructor an observable with its name
43      */
44     Observable(const std::string &name);
45
46     virtual ~Observable() { };
```

194

```
47      /**
48       * Add a sample (outcome instance) to an observable
49       */
50      void addSample(const Sample &sample);
51      /**
52       * @brief Get all sample with endTime timeLowerBound -
        timeUpperBound
53       * @return The sample in range timeLowerBound - timeUpperBound
54       */
55      std::vector<Sample> getSamplesInRange(std::uint64_t timeLowerBound
        , std::uint64_t timeUpperBound);
56      /**
57       * @brief Get observed DeltaQ in range timeLowerBound -
        timeUpperBound from snapshot, if it has not been calculated,
        calculate it
58       * @return DeltaQ
59       */
60      DeltaQ getObservedDeltaQ(uint64_t, uint64_t);
61      /**
62       * @brief Calculate the observed DeltaQ in range timeLowerBound -
        timeUpperBound, add it to snapshot and ConfidenceInterval
63       * @return The calculated DeltaQ
64       */
65      DeltaQ calculateObservedDeltaQ(uint64_t, uint64_t);
66
67      /**
68       * @brief Get DeltaQ representation for graphical plotting
69       */
70      DeltaQRepr getObservedDeltaQRepr(uint64_t, uint64_t);
71
72      /**
73       * @brief Set new parameters for a DeltaQ
74       * @return new dMax
75       */
76      double setNewParameters(int newExp, int newNBins);
77
78      double getBinWidth() const
79      {
80          return DELTA_T_BASE * std::pow(2, deltaTExp);
81      }
82
83      int getNBins() const
84      {
85          return nBins;
86      }
87
88      double getMaxDelay() const
89      {
90          return maxDelay;
91      }
92
93      QTA getQTA() const
94      {
95          return qta;
96      }
97
```

```
 98        int getDeltaTExp() const
 99        {
100            return deltaTExp;
101        }
102
103        const TriggerManager &getTriggerManager() const
104        {
105            return triggerManager;
106        }
107
108        /**
109         * @brief Set recoding snapshot
110         * @param bool is recording
111         */
112        void setRecording(bool);
113
114        /**
115         * @brief Set QTA for an observable
116         * @param qta new QTA
117         */
118        void setQTA(const QTA &);
119        /**
120         * @brief add a trigger to an observable
121         * @param type the observable type
122         * @param condition the condition to evalute
123         * @param action action to perform on trigger fired
124         * @param enabled
125         * @param sampleLimit sampleLimit sample limit for sample limit
    trigger
126         */
127        void addTrigger(TriggerType type, TriggerDefs::Condition condition
    , TriggerDefs::Action action, bool enabled, std::optional<int>
    sampleLimit);
128         /**
129         * @brief Remove observable trigger
130         */
131        void removeTrigger(TriggerType type);
132
133        /**
134         * @brief Get snapshot of observable
135         */
136        Snapshot getSnapshot();
137        /**
138         * @brief Get observable name
139         * @return its name
140         */
141        [[nodiscard]] std::string getName() const &;
142 };
```

D.3.2   Operator.h

```
1 #pragma once
2
3 #include "../maths/DeltaQ.h"
4 #include "Observable.h"
```

```cpp
#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;
    }

    /**
```

```cpp
54      * @brief Get the links for a children
55      * @return The links for a children
56      */
57     std::vector<std::shared_ptr<Observable>> getChildren();
58
59     void setCausalLinks(std::vector<std::vector<std::shared_ptr<
       Observable>>> links)
60     {
61         causalLinks = links;
62     }
63
64     std::vector<std::vector<std::shared_ptr<Observable>>>
       getCausalLinks()
65     {
66         return causalLinks;
67     }
68
69     OperatorType getType()
70     {
71         return type;
72     }
73 };
```

D.3.3 OperatorType.h

```cpp
1 #pragma once
2 // AllToFinish, FirstToFinish, Probabilistic Choice
3 enum class OperatorType { ATF, FTF, PRB };
```

D.3.4 Outcome.h

```cpp
1 /**
2  * @author Francesco Nieri
3  * @date 25/10/2024
4  * Class representing an outcome O_n in a system
5  */
6 #pragma once
7
8 #include "Observable.h"
9
10 class Outcome : public Observable
11 {
12 public:
13     Outcome(const std::string &name);
14     ~Outcome();
15 };
```

D.3.5 Probe.h

```cpp
1 #pragma once
2 #include <string>
3 #include <vector>
```

```cpp
#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;

```

```
55    void setCausalLinks ( std :: vector < std :: shared_ptr < Observable > >
    newCausalLinks )
56    {
57        causalLinks = newCausalLinks ;
58    }
59
60    std :: vector < std :: shared_ptr < Observable > > getCausalLinks ()
61    {
62        return causalLinks ;
63    }
64 };
```

D.3.6 Sample.h

```
1 #pragma once
2
3 #include < cstdint >
4
5 enum Status { SUCCESS , TIMEDOUT , FAILED };
6
7 /**
8  * @struct Sample represent an outcome instance
9  */
10 struct Sample {
11    std :: uint64_t startTime ;
12    std :: uint64_t endTime ;
13    double long elapsedTime ;
14    Status status ;
15 };
```

D.3.7 System.h

```
1 /**
2  * @author Francesco Nieri
3  * @date 26/10/2024
4  * Class representing a DeltaQ system
5  */
6 #pragma once
7
8 #include "../ maths / DeltaQ .h"
9 #include " Observable .h"
10 #include " Operator .h"
11 #include " Outcome .h"
12 #include " Probe .h"
13 #include < memory >
14 #include < unordered_map >
15
16 class System
17 {
18    std :: unordered_map < std :: string , std :: shared_ptr < Outcome > > outcomes
    {}; ///< All outcome
19    std :: unordered_map < std :: string , std :: shared_ptr < Operator > >
    operators {}; ///< All operators
```

```
20      std::unordered_map<std::string, std::shared_ptr<Probe>> probes {};
        /// < All probes
21      std::unordered_map<std::string, std::shared_ptr<Observable>>
        observables {}; ///< The above grouped together
22
23      std::string systemDefinitionText; ///< The definition of the
        system
24
25      bool recordingTrigger = false;
26      std::map<uint64_t, std::vector<Snapshot>> snapshots;
27
28 public:
29      System() = default;
30
31      [[nodiscard]] std::unordered_map<std::string, std::shared_ptr<
        Outcome>> &getOutcomes();
32
33      [[nodiscard]] std::unordered_map<std::string, std::shared_ptr<
        Probe>> &getProbes();
34
35      [[nodiscard]] std::unordered_map<std::string, std::shared_ptr<
        Operator>> &getOperators();
36
37      [[nodiscard]] std::unordered_map<std::string, std::shared_ptr<
        Observable>> &getObservables();
38
39      void setOutcomes(std::unordered_map<std::string, std::shared_ptr<
        Outcome>> outcomesMap);
40
41      void setOperators(std::unordered_map<std::string, std::shared_ptr<
        Operator>> operatorsMap);
42
43      void setProbes(std::unordered_map<std::string, std::shared_ptr<
        Probe>> probesMap);
44
45      bool hasOutcome(const std::string &name);
46
47      bool hasOperator(const std::string &name);
48
49      std::shared_ptr<Outcome> getOutcome(const std::string &outcomeName
        );
50
51      std::shared_ptr<Operator> getOperator(const std::string &);
52
53      bool hasProbe(const std::string &name);
54
55      std::shared_ptr<Probe> getProbe(const std::string &name);
56
57      std::shared_ptr<Observable> getObservable(const std::string &
        observableName);
58
59      void setSystemDefinitionText(std::string &text);
60
61      std::string getSystemDefinitionText();
62
63      void setObservableParameters(std::string &, int, int);
```

```
64
65    /**
66     * @brief Add outcome instance for an observable, if it exists
67     */
68    void addSample(std::string &componentName, Sample &sample);
69
70    /**
71     * @brief Set all observables to record snapshots
72     */
73    void setRecording(bool);
74
75    bool isRecording() const;
76
77    /**
78     * @brief Add the snapshots of all observables for a trigger at
      time t
79     */
80    void getObservablesSnapshotAt(std::uint64_t);
81
82    /**
83     * Get the snapshots of all observables for a trigger at time t
84     */
85    std::map<std::uint64_t, std::vector<Snapshot>> getAllSnapshots();
86
87    /**
88     * @brief Get the name of all components
89     */
90    std::vector<std::string> getAllComponentsName();
91
92    /**
93     * @deprecated This may be used in the future
94     * Calculate the resulting DeltaQ for the whole system
95     */
96    DeltaQ calculateDeltaQ();
97 };
```

D.4   maths

D.4.1   ConfidenceInterval.h

```
1  #pragma once
2
3  #include "DeltaQ.h"
4  #include <vector>
5  // Upper and lower confidence bounds of a DeltaQ's CDF
6  struct Bound {
7      double lowerBound {0};
8      double upperBound {1};
9      double mean {0};
10 };
11
12 /**
13  * @class ConfidenceInterval
```

```cpp
14  * @brief Class representing the confidence interval of a window of
       DeltaQs
15  */
16 class ConfidenceInterval
17 {
18 private:
19     std::vector<Bound> bounds; ///< The bounds at each point
20     unsigned int numBins;
21     unsigned int size {0};
22     std::vector<double> cdfSum; ///< The sum of the cdf at each bin
23     std::vector<double> cdfSumSquares; ///< Variance at each bin
24     std::vector<unsigned int> cdfSampleCounts; ///< Sample per bins
25     double z {1}; ///< Confidence interval
26     void updateConfidenceInterval();
27
28 public:
29     /**
30      * @brief Default constructor, set to 0 bins
31      */
32     ConfidenceInterval();
33     /**
34      * @brief Constructor for ConfidenceInterval with number of bins
35      * @param numBins number of bins
36      */
37     ConfidenceInterval(int numBins);
38
39     void setNumBins(int newNumBins);
40     /**
41      * @brief Add DeltaQ to intervals
42      * @param DeltaQ DeltaQ to add
43      */
44     void addDeltaQ(const DeltaQ &);
45     /**
46      * @brief Remove DeltaQ from intervals
47      * @param DeltaQ DeltaQ to remove
48      */
49     void removeDeltaQ(const DeltaQ &);
50
51     /**
52      * @brief Get current confidence bounds
53      */
54     std::vector<Bound> getBounds() const;
55     /**
56      * @brief Get number of bins
57      * @return Number of bins
58      */
59     unsigned int getBins();
60     /**
61      * @brief Zero the confidence intervals
62      */
63     void reset();
64 };
```

D.4.2   DeltaQ.h

```cpp
/**
 * @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>
#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;
```

```
55    [[nodiscard]] double pdfAt(int x) const;
56    [[nodiscard]] double cdfAt(int x) const;
57    [[nodiscard]] const unsigned int getTotalSamples() const;
58    [[nodiscard]] QTA getQTA() const;
59    void calculateQuartiles(std::vector<Sample> &);
60    void setBinWidth(double newWidth);
61    /**
62     * Operator Overloads
63     */
64    friend DeltaQ operator*(const DeltaQ &deltaQ, double constant);
65    friend DeltaQ operator*(double constant, const DeltaQ &deltaQ);
66    friend DeltaQ operator*(const DeltaQ &lhs, const DeltaQ &rhs);
67    friend DeltaQ operator+(const DeltaQ &lhs, const DeltaQ &rhs);
68    friend DeltaQ operator-(const DeltaQ &lhs, const DeltaQ &rhs);
69
70    friend std::ostream &operator<<(std::ostream &os, const DeltaQ &
      deltaQ);
71
72    /**
73     * Comparison Operators
74     */
75    bool operator<(const DeltaQ &other) const;
76    bool operator>(const DeltaQ &other) const;
77
78    bool operator==(const DeltaQ &deltaQ) const;
79    [[nodiscard]] std::string toString() const;
80 };
81
82 #endif // DELTAQ_H
```

D.4.3   DeltaQOperations.h

```
1  #pragma once
2
3  #include "DeltaQ.h"
4
5  /**
6   * Perform discrete convolution between two DeltaQs
7   */
8  DeltaQ convolve(const DeltaQ &lhs, const DeltaQ &rhs);
9  /**
10  * Perform Fast Fourier Transform on two DeltaQs
11  */
12 DeltaQ convolveFFT(const DeltaQ &lhs, const DeltaQ &rhs);
13 DeltaQ convolveN(const std::vector<DeltaQ> &deltaQs);
14
15 /**
16  * Assume two independent outcomes with the same start event
17  * All-to-finish outcome occurs when both end events occur
18  * All-to-finish is defined as ΔQ_{LTF(A,B)} ΔQ_A * ΔQ_B
19  */
20 DeltaQ allToFinish(const std::vector<DeltaQ> &deltaQs);
21
22 /**
23  * Assume two independent outcomes with the same start event
```

```
24  * First-to-finish outcome occurs when at least one end event occurs
25  * We compute the probability that there are zero end events
26  * First-to-finish is defined as
27  * DeltaQ_{FTF(A, B)} = DeltaQ_A + DeltaQ_B - DeltaQ_A * DeltaQ_B
28  */
29 DeltaQ firstToFinish(const std::vector<DeltaQ> &deltaQs);
30
31 /**
32  *  Assume there are two possible outcomes OA and OB and
33  * exactly one outcome is chosen during each occurrence of a start
       event
34  * O_A occurs with probability p/(p+q)
35  * O_B occurs with probability q/(p+q)
36  * Therefore:
37  * ΔQ_{PC(A,B)} = p/(p+q) ΔQ_A + q/(p+q) ΔQ_B
38  */
39 DeltaQ probabilisticChoice(const std::vector<double> &probabilities,
       const std::vector<DeltaQ> &deltaQs);
40
41 DeltaQ rebin(const DeltaQ &source, double targetBinWidth);
42
43 /**
44  * Choose the highest size from a list of DeltaQs
45  */
46 int chooseLongestDeltaQSize(const std::vector<DeltaQ> &deltaQs);
```

D.4.4   DeltaQRepr.h

```
1
2  #pragma once
3
4  #include "ConfidenceInterval.h"
5  #include "DeltaQ.h"
6  #include <cstdint>
7  #include <vector>
8
9  /**
10  * Class storing a DeltaQ representation for graphical plotting
11  */
12
13 struct DeltaQRepr {
14     std::uint64_t time;
15     DeltaQ deltaQ;
16     std::vector<Bound> bounds;
17
18     DeltaQRepr()
19         : time(0)
20         , deltaQ()
21         , bounds()
22     {
23     }
24
25     // DeltaQRepr copy constructor
26     DeltaQRepr(const DeltaQRepr &other)
27         : time(other.time)
```

```cpp
28          , deltaQ(other.deltaQ)
29          , bounds(other.bounds)
30      {
31      }
32
33      // DeltaQRepr move semantics
34      DeltaQRepr(DeltaQRepr &&other) noexcept
35          : time(other.time)
36          , deltaQ(std::move(other.deltaQ))
37          , bounds(std::move(other.bounds))
38      {
39      }
40
41      // DeltaQRepr assignment operators
42      DeltaQRepr &operator=(const DeltaQRepr &other)
43      {
44          if (this != &other) {
45              time = other.time;
46              deltaQ = other.deltaQ;
47              bounds = other.bounds;
48          }
49          return *this;
50      }
51
52      DeltaQRepr &operator=(DeltaQRepr &&other) noexcept
53      {
54          if (this != &other) {
55              time = other.time;
56              deltaQ = std::move(other.deltaQ);
57              bounds = std::move(other.bounds);
58          }
59          return *this;
60      }
61
62      // Constructor with parameters
63      DeltaQRepr(std::uint64_t t, const DeltaQ &dq, const std::vector<
    Bound> &b)
64          : time(t)
65          , deltaQ(dq)
66          , bounds(b)
67      {
68      }
69 };
```

D.4.5   QTA.h

```cpp
1 #ifndef QTA_H
2 #define QTA_H
3
4 #include <iostream>
5 #include <limits>
6 #define QTA_EPSILON std::numeric_limits<double>::epsilon()
7 /**
8  * @struct QTA Quantitative Timeliness Agreement for an observable
9  */
```

```cpp
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);
    }

    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
```

D.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

```

```cpp
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.
     */
    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;
```

209

```
71
72      /// @return The number of calculated DeltaQs stored.
73      std::size_t getCalculatedSize() const;
74
75      /**
76       * @brief Truncates observed/calculated DeltaQs to a specified
     size (removes oldest entries).
77       * @param size The maximum number of entries to retain.
78       */
79      void resizeTo(size_t size);
80
81      // --- Name Management ---
82      /**
83       * @brief Sets the name of the observable.
84       * @param name New name for the observable.
85       */
86      void setName(const std::string &name);
87
88      /// @return The current observable name.
89      std::string getName() &;
90
91      // --- Data Retrieval ---
92      /// @return A vector of all observed DeltaQReprs (sorted by
     timestamp).
93      std::vector<DeltaQRepr> getObservedDeltaQs() const &;
94
95      /// @return A vector of all calculated DeltaQReprs (sorted by
     timestamp).
96      std::vector<DeltaQRepr> getCalculatedDeltaQs() const &;
97
98      /**
99       * @brief Retrieves an observed DeltaQ at a specific timestamp.
100       * @param timestamp The time to query.
101       * @return The DeltaQRepr if found, or `std::nullopt` otherwise.
102       */
103      std::optional<DeltaQRepr> getObservedDeltaQAtTime(std::uint64_t
     timestamp);
104
105      /**
106       * @brief Retrieves a calculated DeltaQ at a specific timestamp.
107       * @param timestamp The time to query.
108       * @return The DeltaQRepr if found, or `std::nullopt` otherwise.
109       */
110      std::optional<DeltaQRepr> getCalculatedDeltaQAtTime(std::uint64_t
     timestamp);
111
112      /// @return A vector of all QTAs (sorted by timestamp).
113      std::vector<QTA> getQTAs() const &;
114 };
```

D.4.7   TriggerManager.h

```
1 #pragma once
2
3 #include "TriggerTypes.h"
```

```cpp
#include "Triggers.h"
#include <memory>
#include <optional>
#include <vector>

/**
 * @class TriggerManager
 * @brief Manages triggers for an observable
 */
class TriggerManager
{
public:
    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);
```

```
56
57     /**
58      * @brief Remove all triggers
59      */
60     void clearAllTriggers();
61
62     /**
63      * @brief Enable/Disable all triggers with a type
64      * @param type The type of trigger to enable/disable
65      * @param enabled
66      */
67     void setTriggersEnabled(TriggerType type, bool enabled);
68
69     /**
70      * @brief Get all triggers
71      */
72     std::vector<Trigger> getAllTriggers() const;
73
74 private:
75     std::vector<Trigger> triggers_;
76 };
```

D.4.8 TriggerTypes.h

```
1 #ifndef TRIGGERTYPE_H
2 #define TRIGGERTYPE_H
3 #pragma once
4
5 enum class TriggerType {
6     SampleLimit,
7     QTAViolation,
8     Failure,
9     Hazard
10 };
11
12 #endif //TRIGGERTYPE_H
```

D.4.9 Triggers.h

```
1 #pragma once
2
3 #include "DeltaQ.h"
4 #include "QTA.h"
5 #include "TriggerTypes.h"
6 #include <functional>
7 #include <iostream>
8 #include <string>
9
10 namespace TriggerDefs
11 {
12 using Condition = std::function<bool(const DeltaQ &, const QTA &)>;
13 using Action = std::function<void(const DeltaQ &, const QTA &, std::
    uint64_t)>;
```

```
14
15 namespace Conditions
16 {
17     Condition SampleLimit(int maxSamples);
18     Condition QTABounds();
19     Condition FailureRate(double threshold);
20 }
21
22 namespace Actions
23 {
24     Action LogToConsole(const std::string &message);
25     Action notify();
26     Action SaveSnapshot(const std::string &filename);
27 }
28 }
```

D.5   parser

D.5.1   SystemBuilder.h

```
1 #ifndef SYSTEMBUILDER_H
2 #define SYSTEMBUILDER_H
3
4 #include "../diagram/System.h"
5 #include "DQGrammarVisitor.h"
6 #include <memory>
7 #include <unordered_map>
8
9 /**
10  * @brief Visitor class that builds a System from a parsed grammar
     tree.
11  */
12 class SystemBuilderVisitor : public parser::DQGrammarVisitor
13 {
14 private:
15     std::unordered_map<std::string, std::shared_ptr<Outcome>> outcomes
     ;   ///< Map of outcome names to Outcome objects.
16     std::unordered_map<std::string, std::shared_ptr<Operator>>
     operators; ///< Map of operator names to Operator objects.
17     std::unordered_map<std::string, std::shared_ptr<Probe>> probes;
         ///< Map of probe names to Probe objects.
18
19     std::vector<std::string> definedProbes; ///< List of defined probe
      names.
20     System system; ///< The final system constructed by the visitor.
21
22     std::string currentlyBuildingProbe; ///< Tracks the probe
     currently being built for dependency management.
23     std::map<std::string, std::vector<std::string>> dependencies; ///<
      Graph of probe dependencies.
24
25     std::unordered_map<std::string, std::vector<std::string>>
     definitionLinks; ///< For debugging: links between definitions.
```

```
26      std::unordered_map<std::string, std::vector<std::vector<std::
      string>>> operatorLinks; ///< For debugging: operator chains.
27      std::vector<std::string> systemLinks; ///< Top-level system
      observable links.
28      std::unordered_set<std::string> allNames; ///< Tracks all used
      names to detect duplicates.
29
30      /**
31       * @brief Checks the dependency graph for cycles.
32       * @throws std::invalid_argument if a cycle is detected.
33       */
34      void checkForCycles() const;
35
36      /**
37       * @brief Recursive utility to detect cycles in a graph.
38       * @param node Current node.
39       * @param visited Set of visited nodes.
40       * @param recursionStack Stack of nodes in the current DFS path.
41       * @return True if a cycle is found.
42       */
43      bool hasCycle(const std::string& node,
44                    std::set<std::string>& visited,
45                    std::set<std::string>& recursionStack) const;
46
47  public:
48      /**
49       * @brief Returns the constructed System.
50       * @return The system object.
51       */
52      System getSystem() const;
53
54      // Visitor overrides from DQGrammarVisitor:
55      std::any visitStart(parser::DQGrammarParser::StartContext *context
      ) override;
56      std::any visitDefinition(parser::DQGrammarParser::
      DefinitionContext *context) override;
57      std::any visitSystem(parser::DQGrammarParser::SystemContext *
      context) override;
58      std::any visitComponent(parser::DQGrammarParser::ComponentContext
      *context) override;
59      std::any visitBehaviorComponent(parser::DQGrammarParser::
      BehaviorComponentContext *context) override;
60      std::any visitProbeComponent(parser::DQGrammarParser::
      ProbeComponentContext *context) override;
61      std::any visitProbability_list(parser::DQGrammarParser::
      Probability_listContext *context) override;
62      std::any visitComponent_list(parser::DQGrammarParser::
      Component_listContext *context) override;
63      std::any visitOutcome(parser::DQGrammarParser::OutcomeContext *
      context) override;
64      std::any visitComponent_chain(parser::DQGrammarParser::
      Component_chainContext *context) override;
65  };
66
67  #endif // SYSTEMBUILDER_H
```

D.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);
    }
};
```

D.5.3 SystemParserInterface.h

```cpp
#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.
```

```
29      * @return Optional containing the parsed System if successful,
    nullopt otherwise.
30      */
31      static std::optional<System> parseString(const std::string &input)
    ;
32
33  private:
34      /**
35       * @brief Internal parsing method using ANTLR input stream.
36       * @param input ANTLR input stream containing system definition.
37       * @return Optional containing the parsed System if successful,
    nullopt otherwise.
38       */
39      static std::optional<System> parseInternal(antlr4::
    ANTLRInputStream &input);
40  };
```

D.6   server

D.6.1   Server.h

```
1  #ifndef SERVER_H
2  #define SERVER_H
3
4  #include "../diagram/System.h"
5  #include <atomic>
6  #include <condition_variable>
7  #include <memory>
8  #include <mutex>
9  #include <netinet/in.h>
10 #include <queue>
11 #include <sys/socket.h>
12 #include <thread>
13
14 /**
15  * @class Server
16  * @brief TCP server for handling client connections and Erlang
    communication.
17  */
18 class Server
19 {
20 public:
21     /**
22      * @brief Constructs a Server instance.
23      * @param port The TCP port to listen on.
24      */
25     Server(int port);
26
27     /**
28      * @brief Destructor cleans up sockets and threads.
29      */
30     ~Server();
31
32     /**
```

```
33      * @brief Sends a command to the Erlang process.
34      * @param command The command string to send.
35      */
36     void sendToErlang(const std::string &command);
37
38     bool startServer(const std::string &ip = "0.0.0.0", int port =
       8080);
39
40     void stopServer();
41
42     bool setErlangEndpoint(const std::string &ip, int port);
43
44     bool isServerRunning() const
45     {
46         return server_started;
47     }
48
49     /**
50      * @brief Stops the server and worker threads.
51      */
52     void stop();
53
54 private:
55     /**
56      * @brief Main server loop running in a separate thread.
57      */
58     void run();
59
60     int server_fd; ///< Server socket file descriptor
61     int new_socket; ///< Client socket file descriptor
62     struct sockaddr_in address; ///< Server address structure
63     int port; ///< Listening port number
64
65     std::thread serverThread; ///< Thread for server operations
66
67     /**
68      * @brief Updates the system reference from Application.
69      */
70     void updateSystem();
71
72     /**
73      * @brief Parses messages from Erlang.
74      * @param buffer The message buffer.
75      * @param len Length of the message.
76      */
77     void parseErlangMessage(const char *buffer, int len);
78
79     std::shared_ptr<System> system; ///< Reference to the system being
       monitored
80
81     std::vector<std::thread> clientThreads; ///< Active client handler
       threads
82     std::mutex clientsMutex; ///< Mutex for client threads access
83     std::atomic<bool> running {false}; ///< Server running state flag
84
85     /**
```

```cpp
86         * @brief Handles communication with a client.
87         * @param clientSocket The client socket file descriptor.
88         */
89        void handleClient(int clientSocket);
90
91        /**
92         * @brief Cleans up finished client threads.
93         */
94        void cleanupThreads();
95
96        int erlang_socket = -1; ///< Socket for Erlang communication
97        std::mutex erlangMutex; ///< Mutex for Erlang socket operations
98
99        /**
100        * @brief Establishes connection to Erlang.
101        * @return true if connection succeeded.
102        */
103       bool connectToErlang();
104
105       int client_socket; ///< Current client socket
106
107       // Asynchronous sample processing
108       std::queue<std::pair<std::string, Sample>> sampleQueue; ///<
          Sample processing queue
109       std::mutex queueMutex; ///< Mutex for queue access
110       std::condition_variable queueCond; ///< Condition variable for
          queue notifications
111       std::thread workerThread; ///< Worker thread for sample processing
112       bool shutdownWorker = false; ///< Flag to signal worker thread
          shutdown
113
114       std::string erlang_ip = "127.0.0.1"; ///< Erlang server IP
115       int erlang_port = 8081; ///< Erlang server port
116       std::string server_ip = "0.0.0.0"; ///< Current C++ server IP
117       bool server_started = false;
118  };
119
120  #endif
```

Appendix E

Build Configuration Files

E.0.1 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)
```

E.1   dashboard/CMakeLists.txt

```
add_library(${PREFIX}_dashboard
    ColorRegistry.cpp
    ColorRegistry.h
```

```
 4      CustomLegendEntry.h
 5      CustomLegendEntry.cpp
 6      CustomLegendPanel.h
 7      CustomLegendPanel.cpp
 8      DelaySettingsWidget.h
 9      DelaySettingsWidget.cpp
10      DQPlotController.h
11      DQPlotController.cpp
12      DQPlotList.h
13      DQPlotList.cpp
14      DeltaQPlot.h
15      DeltaQPlot.cpp
16      MainWindow.h
17      MainWindow.cpp
18      NewPlotList.h
19      NewPlotList.cpp
20      ObservableSettings.h
21      ObservableSettings.cpp
22      SamplingRateWidget.h
23      SamplingRateWidget.cpp
24      Sidebar.h
25      Sidebar.cpp
26      SnapshotViewerWindow.h
27      SnapshotViewerWindow.cpp
28      StubControlWidget.h
29      StubControlWidget.cpp
30      SystemCreationWidget.h
31      SystemCreationWidget.cpp
32      QTAInputWidget.cpp
33      QTAInputWidget.h
34      TriggersTab.cpp
35      TriggersTab.h
36  )
37
38
39  find_package(QT NAMES Qt6 Qt5 REQUIRED COMPONENTS Core Gyu Widgets
        Charts Graphs)
40  find_package(Qt6 REQUIRED COMPONENTS Core Gui Charts Widgets Graphs)
41
42  target_link_libraries(${PREFIX}_dashboard
43          PUBLIC Qt6::Core Qt6::Gui Qt6::Widgets Qt6::Charts Qt6::Graphs
44          ${PREFIX}_parser
45      )
46
47  target_include_directories(${PREFIX}_dashboard
48          PUBLIC
49          ${CMAKE_SOURCE_DIR}
50          ${PREFIX}_parser
51  )
```

E.2   diagram/CMakeLists.txt

```
1  add_library(${PREFIX}_diagram
2      Outcome.h
```

```
3      Outcome.cpp
4      Operator.h
5      Operator.cpp
6      Probe.h
7      Probe.cpp
8      System.h
9      System.cpp
10     Sample.h
11     Observable.h
12     Observable.cpp
13 )
14
15 target_link_libraries(${PREFIX}_diagram
16     PUBLIC
17         ${PREFIX}_maths
18         ${PREFIX}_application
19 )
20
21
22 target_include_directories(${PREFIX}_diagram
23         PUBLIC
24         ${CMAKE_SOURCE_DIR}
25 )
26
27 add_executable(diagram
28         main.cpp
29 )
30
31 target_link_libraries(diagram
32         PRIVATE
33         ${PREFIX}_diagram
34 )
```

E.3   maths/CMakeLists.txt

```
1 add_library(${PREFIX}_maths
2     ConfidenceInterval.h
3     ConfidenceInterval.cpp
4     DeltaQ.h
5     DeltaQ.cpp
6     DeltaQRepr.h
7     DeltaQOperations.h
8     DeltaQOperations.cpp
9     QTA.h
10     Triggers.h
11     Triggers.cpp
12     TriggerTypes.h
13     TriggerManager.cpp
14     TriggerManager.h
15     Snapshot.h
16     Snapshot.cpp
17 )
18
19 target_include_directories(${PREFIX}_maths
```

```
20          PUBLIC
21          ${CMAKE_SOURCE_DIR}
22 )
23
24 target_link_libraries(${PREFIX}_maths
25      PRIVATE
26      fftw3
27 )
28
29 add_executable(maths
30          main.cpp
31 )
32
33
34 target_link_libraries(maths
35          PRIVATE
36          ${PREFIX}_maths
37 )
```

E.4 parser/CMakeLists.txt

```
1
2 find_package(antlr4-runtime REQUIRED CONFIG)
3
4 if(NOT antlr4-runtime_FOUND)
5      # Manually specify paths (adjust according to your installation)
6      set(ANTLR4_INCLUDE_DIR "/usr/local/include")
7      set(ANTLR4_LIB_DIR "/usr/local/lib")
8
9      find_library(ANTLR4_RUNTIME_LIB antlr4-runtime PATHS ${
   ANTLR4_LIB_DIR})
10     if(NOT ANTLR4_RUNTIME_LIB)
11         message(FATAL_ERROR "ANTLR4 runtime library not found")
12     endif()
13
14     add_library(antlr4-runtime SHARED IMPORTED)
15     set_target_properties(antlr4-runtime PROPERTIES
16         IMPORTED_LOCATION ${ANTLR4_RUNTIME_LIB}
17         INTERFACE_INCLUDE_DIRECTORIES ${ANTLR4_INCLUDE_DIR}
18     )
19 endif()
20
21 set(ANTLR_GENERATED_DIR ${CMAKE_CURRENT_BINARY_DIR}/generated)
22 file(MAKE_DIRECTORY ${ANTLR_GENERATED_DIR})
23
24 # Generate ANTLR files
25 find_program(ANTLR_EXECUTABLE antlr4)
26
27 execute_process(
28     COMMAND ${ANTLR_EXECUTABLE}
29         -Dlanguage=Cpp
30         -visitor -no-listener
31         -o ${ANTLR_GENERATED_DIR}
32         -package parser
```

```
33          ${CMAKE_CURRENT_SOURCE_DIR}/DQGrammar.g4
34      WORKING_DIRECTORY ${CMAKE_CURRENT_SOURCE_DIR}
35      OUTPUT_VARIABLE ANTLR_OUTPUT
36      ERROR_VARIABLE ANTLR_ERROR
37      RESULT_VARIABLE ANTLR_RESULT
38  )
39  if(NOT ANTLR_RESULT EQUAL 0)
40      message(FATAL_ERROR "ANTLR generation failed: ${ANTLR_ERROR}")
41  endif()
42
43  message(STATUS "ANTLR generated: ${ANTLR_OUTPUT}")
44
45  add_library(${PREFIX}_parser
46      ${ANTLR_GENERATED_DIR}/DQGrammarLexer.h
47      ${ANTLR_GENERATED_DIR}/DQGrammarLexer.cpp
48      ${ANTLR_GENERATED_DIR}/DQGrammarParser.h
49      ${ANTLR_GENERATED_DIR}/DQGrammarParser.cpp
50      ${ANTLR_GENERATED_DIR}/DQGrammarBaseVisitor.cpp
51      ${ANTLR_GENERATED_DIR}/DQGrammarVisitor.cpp
52      ${ANTLR_GENERATED_DIR}/DQGrammarBaseVisitor.h
53      ${ANTLR_GENERATED_DIR}/DQGrammarVisitor.h
54       SystemBuilder.h
55       SystemBuilder.cpp
56       SystemErrorListener.h
57       SystemParserInterface.h
58       SystemParserInterface.cpp
59  )
60  target_link_libraries(${PREFIX}_parser
61      PUBLIC antlr4-runtime
62  )
63
64  target_include_directories(${PREFIX}_parser
65      PUBLIC
66      ${CMAKE_SOURCE_DIR}
67      ${ANTLR_GENERATED_DIR}
68      ${ANTLR4_INCLUDE_DIR}
69  )
70
71  add_executable(parser
72      main.cpp
73  )
74
75  target_link_libraries(parser
76      PRIVATE
77      ${PREFIX}_parser
78      antlr4-runtime
79   ${PREFIX}_diagram
80  )
```

E.5   server/CMakeLists.txt

```
1  add_library(${PREFIX}_server
2      Server.cpp
3      Server.h
```

```
4  )
5
6  target_link_libraries(${PREFIX}_server
7      PUBLIC
8          ${PREFIX}_diagram
9      )
10
11 target_include_directories(${PREFIX}_server
12     PUBLIC
13     ${CMAKE_SOURCE_DIR}
14 )
```

Appendix F

Erlang Source Files

F.1 Root

F.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
30 %%%======================
31 %%% Application Callbacks
32 %%%======================
33
34
35 start(_Type, _Args) ->
36     dqsd_otel_sup:start_link().
37
38 stop(_State) ->
39     ok.
40
41 init_ets() ->
42     ets:new(timeout_registry, [named_table, public, set]),
43     ets:new(otel_state, [named_table, public, set]),
44     ets:insert(otel_state, {stub_running, false}),
45     {ok, self()}.
46
47
48 %%%======================
49 %%% For testing purposes
50 %%% ======================
51
52 set_stub_running(Bool) when is_boolean(Bool) ->
53     ets:insert(otel_state, {stub_running, Bool}),
54     io:format("Stub running set to: ~p~n", [Bool]),
55     ok.
56
57 %%%======================
58 %%% Public API
59 %%%======================
60 %% @doc Starts the otel_wrapper application and all dependencies.
61 -spec start() -> {ok, [atom()]} | {error, term()}.
62 start() ->
63     application:ensure_all_started(dqsd_otel).
64
65 %% @doc Starts a span with the given name, if the stub is running.
66 %% Returns a tuple of SpanContext and the internal span process PID or
       `ignore`.
67 -spec start_span(binary()) -> {opentelemetry:span_ctx(), pid() |
      ignore}.
68 start_span(Name) ->
69     SpanCtx = ?start_span(Name),
70     case ets:lookup(otel_state, stub_running) of
71         [{_, true}] ->
72             case ets:lookup(timeout_registry, Name) of
73                 [{_, T}] ->
74                     StartTime = erlang:system_time(nanosecond),
75                     Pid = spawn(?MODULE, span_process, [Name,
    StartTime, T]),
76                     {SpanCtx, Pid};
77                 [] ->
78                     {SpanCtx, ignore}
79             end;
80         _ ->
81             {SpanCtx, ignore}
82     end.
```

```
83
84 %% @doc Starts a span with attributes.
85 -spec start_span(binary(), map()) -> {opentelemetry:span_ctx(), pid()
      | ignore}.
86 start_span(Name, Attrs) when is_map(Attrs) ->
87     SpanCtx = ?start_span(Name, Attrs),
88     case ets:lookup(otel_state, stub_running) of
89         [{_, true}] ->
90             case ets:lookup(timeout_registry, Name) of
91                 [{_, T}] ->
92                     StartTime = erlang:system_time(nanosecond),
93                     Pid = spawn(?MODULE, span_process, [Name,
    StartTime, T]),
94                     {SpanCtx, Pid};
95                 [] ->
96                     {SpanCtx, ignore}
97             end;
98         _ ->
99             {SpanCtx, ignore}
100    end.
101
102 %% @doc Ends the span and reports it, unless stub is disabled or Pid
      is `ignore`.
103 -spec end_span(opentelemetry:span_ctx(), pid() | ignore) -> ok | term
      ().
104 end_span(Ctx, Pid) ->
105    ?end_span(Ctx),
106        case Pid of
107            ignore -> ok;
108        _ when is_pid(Pid) ->
109            Pid ! {<<"end_span">>, erlang:system_time(nanosecond)}
110    end.
111
112
113 %% @doc Fail the span and reports it to the oscilloscope, unless stub
      is disabled or Pid is `ignore`.
114 -spec fail_span( pid() | ignore) -> ok | term().
115 fail_span(Pid) ->
116    case Pid of
117        ignore -> ok;
118    _ when is_pid(Pid) ->
119        Pid ! {<<"fail_span">>, erlang:system_time(nanosecond)}
120    end.
121
122
123
124 %% @doc Executes Fun inside a span with attributes.
125 -spec with_span(binary(), fun(() -> any())) -> any().
126 with_span(Name, Fun) ->
127    ?with_span(Name, #{},
128        fun(_SpanCtx) ->
129            Pid = start_with_span(Name),
130            Result = Fun(),
131            end_with_span(Pid),
132            Result
133        end).
```

```erlang
134
135 %% @doc Executes Fun inside a span with attributes.
136 -spec with_span(binary(), fun(() -> any()), map()) -> any().
137 with_span(Name, Fun, Attrs) when is_map(Attrs), is_function(Fun, 0) ->
138     ?with_span(Name, Attrs,
139         fun(_SpanCtx) ->
140             Pid = start_with_span(Name),
141             Result = Fun(),
142             end_with_span(Pid),
143             Result
144         end).
145
146 start_with_span(Name) ->
147     case ets:lookup(otel_state, stub_running) of
148         [{_, true}] ->
149             case ets:lookup(timeout_registry, Name) of
150                 [{_, T}] ->
151                     StartTime = erlang:system_time(nanosecond),
152                     Pid = spawn(?MODULE, span_process, [Name,
    StartTime, T]),
153                     Pid;
154                 [] ->
155                     ignore
156             end;
157         _ ->
158             ignore
159     end.
160
161 end_with_span(Pid) ->
162         case Pid of
163             ignore -> ok;
164         _ when is_pid(Pid) ->
165             Pid ! {<<"end_span">>, erlang:system_time(nanosecond)}
166     end.
167
168
169
170 %%%=======================
171 %%% Span Worker
172 %%%=======================
173
174 span_process(NameBin, StartTime, Timeout) ->
175     Deadline = StartTime + (Timeout * 1000000),
176     Timer = erlang:send_after(Timeout, self(), {<<"timeout">>,
    Deadline}),
177     receive
178         {<<"fail_span">>, EndTime} ->
179             io:format("failure"),
180             erlang:cancel_timer(Timer),
181             send_span(NameBin, StartTime, EndTime, <<"fa">>);
182         {<<"end_span">>, EndTime} ->
183             erlang:cancel_timer(Timer),
184             send_span(NameBin, StartTime, EndTime, <<"ok">>);
185         {<<"timeout">>, Deadline} ->
186             send_span(NameBin, StartTime, Deadline, <<"to">>)
187     end.
```

```erlang
188
189
190  %%%=======================
191  %%% Handle Incoming Messages from C
192  %%%=======================
193
194  handle_c_message(Bin) when is_binary(Bin) ->
195      case binary:split(Bin, <<";">>, [global]) of
196          [<<"set_timeout">>, Name, TimeoutBin] ->
197              case string:to_integer(binary_to_list(TimeoutBin)) of
198                  {Timeout, _} when is_integer(Timeout) ->
199                      ets:insert(timeout_registry, {Name, Timeout}),
200                      io:format("dqsd_otel: Timeout set: ~p = ~p~n", [
      Name, Timeout]);
201                  _ ->
202                      io:format("dqsd_otel: Invalid timeout: ~p~n", [
      TimeoutBin])
203              end;
204          [<<"start_stub">>] ->
205              ets:insert(otel_state, {stub_running, true}),
206              io:format("dqsd_otel: Stub enabled~n");
207          [<<"stop_stub">>] ->
208              ets:insert(otel_state, {stub_running, false}),
209              io:format("dqsd_otel: Stub stopped~n");
210          _ ->
211              io:format("dqsd_otel: Unknown command: ~p~n", [Bin])
212      end.
213
214  %%%=======================
215  %%% Sending Span Data to C
216  %%%=======================
217
218  send_span(NameBin, Start, End, StatusBin) ->
219      Data = io_lib:format("n:~s;b:~p;e:~p;s:~s~n", [
220          NameBin,
221          Start,
222          End,
223          StatusBin
224      ]),
225      dqsd_otel_tcp_client:send_span(lists:flatten(Data)).
```

F.1.2 dqsd_otel_app.erl

```erlang
1  %%%-------------------------------------------------------------------
2  %% @doc dqsd_otel public API
3  %% @end
4  %%%-------------------------------------------------------------------
5
6  -module(dqsd_otel_app).
7
8  -behaviour(application).
9
10  -export([start/2, stop/1]).
11
12  start(_StartType, _StartArgs) ->
```

```
13        dqsd_otel_sup:start_link().
14
15  stop(_State) ->
16        ok.
```

F.1.3 dqsd_otel_sup.erl

The supervisor of the adapter. It start the TCP server, client and adapter.

```
1
2  -module(dqsd_otel_sup).
3  -behaviour(supervisor).
4  -export([start_link/0, init/1]).
5
6  start_link() ->
7        supervisor:start_link({local, ?MODULE}, ?MODULE, []).
8
9  init([]) ->
10        ChildSpecs = [
11            #{id => ets_init,
12              start => {dqsd_otel, init_ets, []},
13              restart => temporary,
14              shutdown => brutal_kill,
15              type => worker,
16              modules => [dqsd_otel]},
17
18            #{id => tcp_server,
19              start => {dqsd_otel_tcp_server, start_link, []},
20              restart => permanent,
21              shutdown => brutal_kill,
22              type => worker,
23              modules => [dqsd_otel_tcp_server]},
24
25            #{id => tcp_client,
26              start => {dqsd_otel_tcp_client, start_link, []},
27              restart => permanent,
28              shutdown => brutal_kill,
29              type => worker,
30              modules => [dqsd_otel_tcp_client]}
31        ],
32        {ok, {{one_for_one, 5, 10}, ChildSpecs}}.
```

F.1.4 dqsd_otel_tcp_client.erl

The TCP client, it can send outcome instances to the oscilloscope.

```
1
2      -module(dqsd_otel_tcp_client).
3      -behaviour(gen_server).
4
5      -export([start_link/0, send_span/1]).
6      -export([init/1, handle_call/3, handle_cast/2, handle_info/2,
    terminate/2]).
7      -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
58        IPStr = case IP of
59            Bin when is_binary(Bin) -> binary_to_list(Bin);
60            Str when is_list(Str)   -> Str
61        end,
62        case gen_tcp:connect(IPStr, Port, [binary, {active, false}])
   of
63            {ok, Socket} ->
64                io:format("dqsd_otel: Wrapper connected to ~s:~p~n", [
   IPStr, Port]),
65
66                case State#state.socket of
67                    undefined -> ok;
68                    OldSocket -> catch gen_tcp:close(OldSocket)
69                end,
70                {noreply, State#state{socket = Socket,
   logged_disconnected = false}};
71            {error, Reason} ->
72                io:format("dqsd_otel: Connection to ~s:~p failed: ~p~n
   ", [IPStr, Port, Reason]),
73                {noreply, State}
74        end;
75
76  handle_cast(disconnect, State = #state{socket = undefined}) ->
77        io:format("dqsd_otel: Socket already disconnected.~n"),
78        {noreply, State};
79
80  handle_cast(disconnect, State = #state{socket = Socket}) ->
81        io:format("dqsd_otel: Disconnecting TCP socket.~n"),
82        gen_tcp:close(Socket),
83        {noreply, State#state{socket = undefined, logged_disconnected
   = false}}.
84
85  handle_info(_, State) ->
86        {noreply, State}.
87
88  handle_call(_, _From, State) ->
89        {reply, ok, State}.
90
91  terminate(_Reason, State) when is_record(State, state) ->
92        case State#state.socket of
93            undefined -> ok;
94            Socket -> gen_tcp:close(Socket)
95        end;
96  terminate(_Reason, _Other) ->
97        ok.
```

F.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
1  %%%-------------------------------------------------------------
2  %%% @doc
3  %%% 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{}}.
```

```erlang
58  %%------------------------------------------------------------------
59  %% @private
60  %% Handles the start request. Binds to the given IP and port.
61  %% On success, starts the accept loop.
62  %%------------------------------------------------------------------
63  handle_call({start, IP, Port}, _From, State) ->
64      Options = [binary, {packet, line}, {active, false}, {reuseaddr,
        true}, {ip, parse_ip(IP)}],
65      case gen_tcp:listen(Port, Options) of
66          {ok, Socket} ->
67              Acceptor = spawn(fun() -> accept_loop(Socket) end),
68              io:format("dqsd_otel: Listening socket started on ~p:~p~n"
        , [IP, Port]),
69              {reply, ok, State#state{socket = Socket, acceptor =
        Acceptor}};
70          {error, Reason} ->
71              io:format("dqsd_otel: Could not start listening socket on
        ~p:~p~n", [IP, Port]),
72              {reply, {error, Reason}, State}
73      end;
74
75  %%------------------------------------------------------------------
76  %% @private
77  %% Handles stop request. Stops acceptor and closes socket.
78  %% Returns error if server is not running.
79  %%------------------------------------------------------------------
80  handle_call(stop, _From, #state{socket = undefined, acceptor =
        undefined} = State) ->
81      io:format("dqsd_otel: Server not running.~n"),
82      {reply, {error, not_running}, State};
83
84
85  handle_call(stop, _From, #state{socket = Socket, acceptor = Acceptor})
        ->
86      catch exit(Acceptor, shutdown),
87      gen_tcp:close(Socket),
88      io:format("dqsd_otel: Stopped listening from oscilloscope~n"),
89      {reply, ok, #state{socket = undefined, acceptor = undefined}}.
90
91  handle_cast(_, State) -> {noreply, State}.
92  handle_info(_, State) -> {noreply, State}.
93
94  %%------------------------------------------------------------------
95  %% @private
96  %% Accept loop that runs in a separate process.
97  %% Accepts TCP connections and spawns a handler for each one.
98  %%------------------------------------------------------------------
99  accept_loop(ListenSocket) ->
100     {ok, Socket} = gen_tcp:accept(ListenSocket),
101     spawn(fun() -> handle_client(Socket) end),
102     accept_loop(ListenSocket).
103
104 %%------------------------------------------------------------------
105 %% @private
106 %% Handles a client socket connection. Reads line-by-line.
107 %% Forwards trimmed binary lines to `dqsd_otel:handle_c_message/1`.
```

```
108 %%----------------------------------------------------------------------
109 handle_client(Socket) ->
110     case gen_tcp:recv(Socket, 0) of
111         {ok, Line} ->
112             Trimmed = binary:replace(Line, <<"\n">>, <<>>, [global]),
113             dqsd_otel:handle_c_message(Trimmed),
114             handle_client(Socket);
115         {error, closed} ->
116             gen_tcp:close(Socket)
117     end.
118
119 %%----------------------------------------------------------------------
120 %% @private
121 %% Parses an IP address from string, binary, or already-parsed tuple.
122 %%----------------------------------------------------------------------
123 parse_ip("127.0.0.1") -> {127,0,0,1};
124 parse_ip("0.0.0.0") -> {0,0,0,0};
125 parse_ip(IP) when is_list(IP) ->
126     {ok, Addr} = inet:parse_address(IP), Addr;
127 parse_ip(IP) -> IP.
```

Appendix G

Erlang Application Files

G.1   Root

G.1.1   dqsd_otel.app.src

The app.src file of the adapter.

```erlang
{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"}]}]}.
```