OULUN AMMATTIKORKEAKOULU

Kalle Lassila

Enhanced Modem SW trace visualization

Enhanced Modem SW trace visualization

Kalle Lassila
Final projects
Spring 2023
Bachelor of Engineering, Information Technology
Oulu University of Applied Sciences

ABSTRACT

Oulu University of Applied Sciences
Bachelor of Engineering, Information Technology

Author: Kalle Lassila
Title of thesis: Enhanced Modem SW trace visualization
Supervisors: Olli Himanka, Asko Ruotsalainen
Term and year when the thesis was submitted: Spring 2023
Number of pages: 33

The goal of this work was to ease embedded modem software debugging and error analysis by producing a visualization software which is able to show modem events and commands in air interface time or any other needed time base. This study examines how information from different sources can be visualized in one place and why those visualizations are important. Difficulties in displaying visualizations using different time axis is also discussed and some problems caused by varying time formats.

Evaluation of different plotting and visualization software from full software solutions to open-source libraries was necessary to be able to do this. In the end bokeh python library was selected for the job. For this reason, Python was selected as the used programming language as it is the language bokeh is written on and for.

The goal of easing modem software debugging sets many requirements for this software: the input data may be scattered or sometimes some of it may be missing entirely. Types of modem software problems being debugged change, which present a challenge: a need to be as generic as possible to be able to add different types of visualizations quickly. Also, when the type of the problem changes so does the contents of the input data files.

Trace log, tracing, data visualization, modem, logging

CONTENTS

ABREVIATIONS

AIR IF	Air interface
AIR IF time	A time at which an action takes place in air interface.
API	Application Programming Interface
CSV	Comma Separated Values: A text file format.
Event	Some action taken by the modem.
GUI	Graphical User Interface: Specifically, a graphically implemented UI.
HTML	Hypertext Markup Language: Used to render web pages.
JavaScript	A programming language for the web.
MTK	MediaTek
Python	An interpreted programming language.
somename.py	A python source code file: A file containing Python code.
Trace	A piece of logged data from a device.
UI	User Interface: Some interface facing the user.

1 INTRODUCTION

When developing software on a computer, you have the advantage of the development system and the target system being the same so it is possible to add breakpoints in the code to interrupt execution and read things from memory to see what is happening while debugging.

This advantage does not exist when working with embedded modem software. Typically modem SW development is done using a lot of SW/HW based tracing and these are processed offline for further analysis. A trace is basically a log entry written to a file to be analysed later. It has a timestamp and it contains some data of an event that happened. SW developer can typically quite freely define the trace content, thus in embedded devices the tracing bandwidth might be limiting the amount of data and speed of how much you can trace out. Instead of tracing everything that happens in a device as they happen, there are compressed instructions which are traced out. Quite often this leads to the trace being written out and the event happening sometime in the future. This is the challenge which this thesis work will try to tackle.

The tool we are developing is intended to help with the following problems:

- Speed up log analysis process with visualization
- Combine Traces from multiple sources
- Time alignment between multiple source traces
- Use real Air Interface time for Plotting

This thesis work will try to explain those challenges in more detail in following chapters. Also, there will be a clarification why those challenges make SW and system debugging difficult. One could think that having some data and a timestamp for said data would be good enough for plotting; find the place on time axis which corresponds to the time in the timestamp and show the data there. This is how the graphical plotting traditionally works in embedded systems. The tracing and debugging embedded systems require the following information to be available: Data when trace came out, Data which is “decoded/encoded” and time aligned into AIR IF time.

The timestamp from when the trace was written, and the event triggered by the instruction traced happening in air interface do not match. This means that the generated graphs do not accurately represent reality, timing-wise. Figure 1 illustrates this well; the two graphs are of the same operations, but different timing information is used to plot these. The upper graph is air interface time and the one below is based on the time when SW trace was written out. From Figure 1 can see that the two graphs do not match, so if an engineer was to use the later graph for debugging it would not give an accurate result of what is actually happening. For example, some changes in state seem to be simultaneous while they are not. In essence those two lines are not identical in shape which will lead into incorrect analysis from system behaviour.

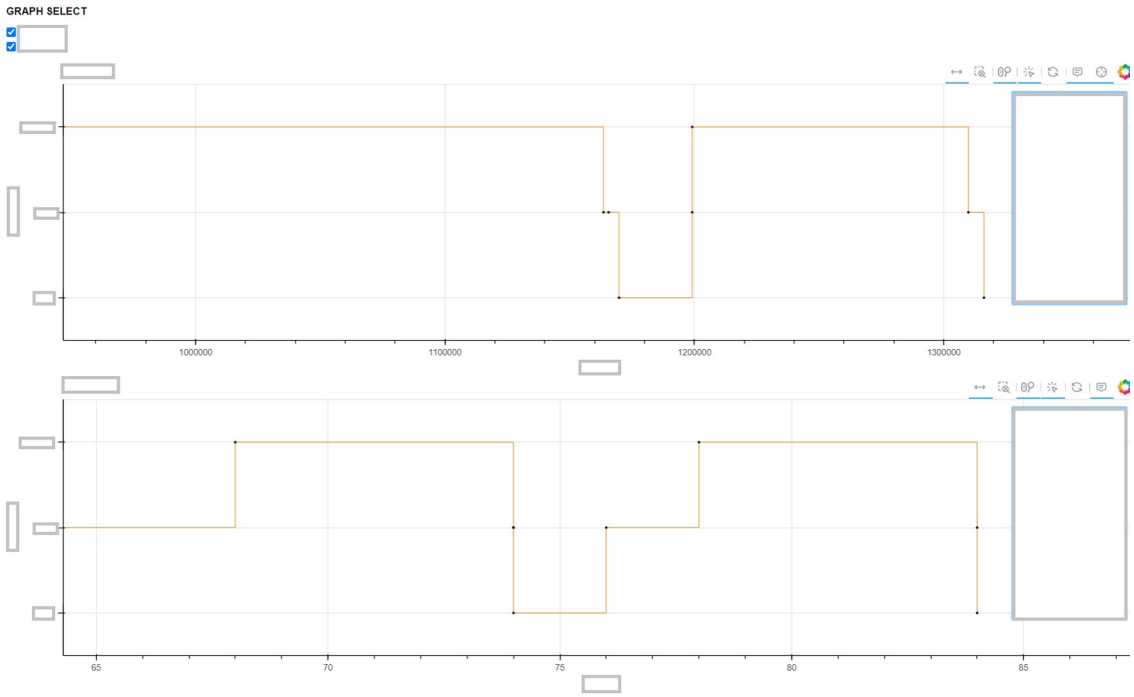

Figure 1: Same events plotted using differing timestamps.

2 PROBLEM STATEMENT OPENING

The main problem in modem debugging is that it takes time. First time is spent looking for all the required files and then more time is needed to scroll through those files to find out what went wrong. The following chapters explain the main problems in detail.

2.1 Text log vs. visualization

Log analysis is always a time-consuming exercise and for quick customer response we need to be able to do it as fast as possible. One log file could contain hundreds of thousands of lines of traces and just text format analysis is very time consuming. Graphical plotting is almost a necessity in modem software debugging as there is so much happening at once that reading text logs requires much effort and consumes time. The log entries may or may not be in the same file or readable from the same interface so switching between different UIs is necessary. Our intention is to simplify this process by the means of employing graphical interpretations of the log-files to make it easier to go through a lot of data.

For all graphical plotting systems, it is important to have possibility to scale and filter data efficiently. Also, overall speed of the tool is important. With graphs you need to quickly get an overview of what the system scenario was when the problem arose and then you can go into detail view to check what is wrong.

2.2 Scattered data

All information necessary for error analysis is not found in one place or a single log file. There could be multiple log files from different CPU systems or HW. This will bring additional challenge for log analysis. To combine all the traces efficiently and in “readable” format is one of the key things for all SW engineers when they start to debug issues. Also, all the tracing is not enabled all the time and different traces use different buffers causing the final log-file not be ordered nicely. This leads to some problems explained in more detail in the next chapter. Also, as a bonus the format of a specific trace may change over time.

2.3 Timing differences

Like in most even slightly complex digital electronics, modems also have a multitude of clocks and counters for various purposes. In this work we need to consider two of them; both of which happen to be free running counters. One clock is a slower one which for all intents and purposes can be thought of counting up for all of eternity, at least from this work's point of view even if the clock technically does wrap around at some point. The other, a faster counter has the possibility of wrapping in the time ranges we are working with, which will pose an issue for our work.

Clock overflowing and starting again from 0 is called wrapping. A single wrap is the interval between the counter starting from a value and coming back to that same value. We will come back to this wrapping and the problem it causes and how that is solved in more detail in chapter 6.3.2 which discusses clock wraps.

In short: When the timestamps start again from zero, it is no longer clear if the time belongs to a wrap or the one before or the one after (3) or it may be from 10 wraps ago or a day ago. The timestamp from the faster clock alone does not contain enough information.

The current way of plotting is the following: The already existing plotter encounters a trace name which is selected for drawing, it reads the timestamp from the slower clock and other necessary info and draws a marker to a graph at that timestamp. This slower clock is not accurate enough for our use.

We now need to plot events by their air interface time, which is slightly more complicated in terms of figuring out the proper time to draw the markers in. Considering the traces we are interested in, in majority of the cases the timing information is included somewhere in the trace; it may be offset by some amount, or missing entirely, which poses a challenge.

The log not being in entirely chronological order as told previously also contributes to this wrap problem, as if all the traces were in the order they happened, we would not have a problem. The order in the log would tell us which wrap the trace belongs to, but as said the log entries may be spread around and mixed up.

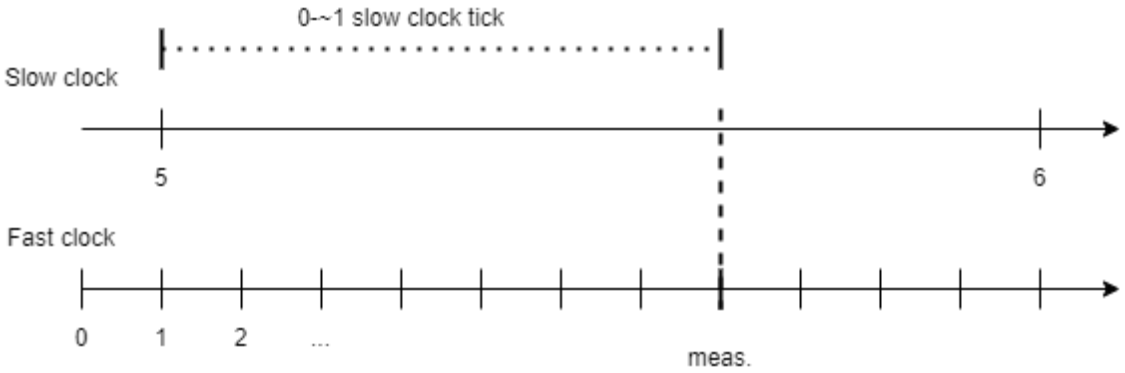

Figure 2: Time sync inaccuracy.

There is a possibility to sync these two timers by a trace which contains the value form both counters at a particular time. The problem we run into doing this is that we do not know how long ago the slower counter incremented, this means the sync is only accurate within one slower clock tick (Figure 2), which is not accurate enough for our use. This happens because two independent counters do not increment at the same moment and there is no way to know how long ago a counter increased.

3 COMMERICAL APPROACHES

A lot of commercial log-file visualization software exist. (10) Many of these already existing, basically turn-key solutions are mostly meant for computer network infrastructure analysis, performance measurement and threat detection or require directing logs to some system as a service solution to be processed and thus not suited for our use case. Some solutions like Dynatrace or new relic could be used, but the idea of such service is that it already supports the data you have and can immediately plot it, which would not be the case in our situation as the trace data format is our internal. These mentioned solutions also have a focus on data collection, which we do not need to do here. The need to reveal internal information to a third party to be able to use a turn-key solution rules all these solutions out. As turn-key solutions are now out of question we turn to bare plotting software like matplotlib, gnuplot and bokeh.

3.1 Matplotlib and Gnuplot

In terms of useability matplotlib, gnuplot and bokeh all are similar in all most common use cases. They do not need much preparation, just import the library, use some method to create a graph and another method to plot data on it.

Gnuplot being geared towards on-the-spot visualization and scripting with for example in MATLAB or Octave, it is not the best for our use case, even though it could be used to generate PNG images and these could be shown in some user interface. This would make for a very light weight plotter in terms of processing power because you just basically render PNG images on screen. But those images lack some interactive functionality which we need or its more difficult to implement because gnuplot is based around interactive command line interface.

Matplotlib on the other hand provides all necessary functions for this tool we make and could be used. Matplotlib examples and feature highlights (11) shows some of the interactive graph tools and features, which are similar to the ones found in bokeh, even though the default set of tools is not so extensive.

4 SELECTED OPTION: BOKEH

There are several factors impacting our approach selection. One of key reasons is that we have several internal requirements and with commercial approaches it is not that easy to fulfill those without revealing internal design and company proprietary information to third party. These requirements are introduced in more detail in chapter 5.1.1.

Also, when we develop the tool by ourselves, we have the freedom to customize it however we may need in the future. The great advantage is that we still use general framework, bokeh, which will allow us to benefit from other developers who maintain Bokeh. This reduces our workload by not creating yet another plotting framework.

4.1 This is Bokeh

Bokeh is an open-source visualization library for Python. Its core idea is to be simple yet powerful plotting framework which provides consistent, high-performance output (20). Bokeh is built around modern web browsers with the goal of having high level of interactivity and performance to accommodate that interactivity.

4.2 Why we selected Bokeh

From the options available, bokeh seemed to have most of the needed features:

- **Flexibility:** Offered by letting the programmer modify basically every object attribute to change plotting outcome.
- **Interactivity:** Although many plotting software can do this, in bokeh it is particularly easy to do as many interactive tools are enabled by default and even more of the included tools can be enabled without much hassle(12).
- **Customization:** Bokeh also provides a very powerful base for custom plotting via a custom JavaScript functions, thus providing the ability to inject and run developer or even user provided JavaScript in arbitrary places (13).

- **Open source:** Bokeh is mature open-source project and there is no associated cost to using it and the developer community is of great help when issues using the framework arise. (19)
- **Re-use:** Maybe the biggest reason to use bokeh was that there already was existing use of it internally so the knowledge for how to use it and how it functions was already present.

Flexibility and customization in bokeh are implemented as follows: Bokeh has two or three main means for making and modifying visualizations depending how one counts. A simple way is by using more high-level methods (15). This allows for very quick visualization trials to try out different plotting styles and for our use it allows us to quickly build plots. High level plotting methods are very generic and do not contain any special features. This is where the other ways of plotting comes into play. It consists of lower-level methods in bokeh.models interface and in bokeh.plotting interface. Also modification of the graph attributes directly is possible (16). A very useful thing about these two ways of making plots is that the lower level is just a continuation of the high-level usage. A high-level method generates a graph and after that graph object exists a developer can use those low-level methods or modify the graph attributes directly, for example by injecting JavaScript in some parts of it. (17)

Bokeh is built to be used interactively with a web browser so it has a great deal of interactive tools at its disposal. For example, the toolbar in every generated figure by default includes most common tools used while looking at visualizations. Even when the html files are pre generated and no python code is running, bokeh leverages JavaScript in the html page to present interactive toolset for the user. (18)

5 REQUIREMENTS

This chapter will introduce overall requirements for this tool and the features. With more details about what those features are and how they are made. The most important functionalities will be clarified in more detail.

5.1 GENERAL REQUIREMENTS

Internal tool requirements can be divided into two categories:

- General system and tool requirements
- Plot specific requirements

General requirements are those that are tool wide affecting every plot and the overall usage and development of the tool. Plot specific requirements are features from a specific plot. These specific features are general in a way that the tool must be able to handle these but may or may not be used only in one specific place.

5.1.1 Graph design requirements

One of the fundamental requirements is to improve readability. The graphs need to be information dense because there is a lot of information, but not too dense as to keep it readable. A few important aspects for the graph design are:

- Intuitive zoom functionality
- Pan feature
- Color coding and usage
- Whitespace use
- Line width
- Line and point shape
- Scaling into screen

Graphical UI and visualization design needs careful consideration and close co-work with end users so that without additional documentation you can easily understand the meaning of each plot.

5.1.2 API/Input file

This visualization tool does not work independently, and it creates strong requirement for API and input file formatting. All the parameters need to be clearly specified. In the API we need to define what “each column” means from input file. This is an extremely important requirement as we want to keep this tool as general as possible so that we could extend and use it for multiple use cases with the same framework. The goal was to end up with as generic as possible piece of software so that future additions and possible, still unknown, future use-cases would be easy to implement.

5.1.3 Standalone and integrated usage

Plotter needs to support standalone and integrated use. We might have use cases where we just want to use this plotting with “one-off” data, and we also have cases where the plotting is integrated as a part of the other log analysis tool set and Plot´s generated automatically.

Standalone use comes up with some rare or new issues or new use-cases where the plotting capability has not been created. This way one can generate and parse data manually from various sources and feed it to this plotter with proper formatting mentioned in chapter API/Input file.

Integrated usage is the most common use-case as this tool mainly comes packaged with another tool which uses this integration to visualize traces.

5.1.4 Transparency from the user’s point of view in integrated usage

One requirement was to be absolutely transparent from the user’s point of view. With transparency we mean that the user should not need to know how and what is done to get these

visualizations. Instead, all should just appear automatically to the end user without extra tool version installations. Python is not the best solution to solve this requirement as it requires an interpreter to be present to interpret the program. The solution for this problem is to use a library, in this case cx_freeze, which is able to compile the program along with all of its dependencies into an executable and in this case also a directory containing the needed dependencies. (4)

This application needed to be self-contained as multiple layers of dependencies are cumbersome for the user to install and prone to errors such as: installing the wrong version of something or conflicting with another software. By having everything packed up means we can be sure that everyone always uses the same versions of the dependencies. The need to be self-contained created some problems like increased startup overhead and file size. Some of this overhead can be mitigated by using dependencies which we know absolutely will be present on the system.

5.1.5 Plotter resolution

As already explained in introduction chapters, there are multiple timers which are used to trace out SW or HW operations. One of the most important requirements for this tool is to make sure that it can handle different time resolutions and synchronization between those. Additionally, there are also some operations happening more rarely, and we need to make selections which traces are visualized from the whole log and which ones are visualized only from selected time range.

As a solution for this requirement, we defined two levels of resolution for plotting:

- High level plotting view
- Detail level plotting view

High level plotting view includes two different time formats: a slow ticking time stamp and an even slower ticking timestamp mapped to each other. This allows the visualization of more traces as the more accurate timing information is not always available. The second timing resolution for this tool is the very fast clock time stamps. Generally, it is always used when it is available in the trace or can be made available somehow.

There is one simple rule which is used to select which traces comes into high level or low-level plotting. Very low occurrence traces can be plotted from the timespan of the entire log file creating a nice overview of the general actions taking place in the file. This allows the user to have some context from a wide time range. Very high occurrence traces on the other hand are only parsed and displayed from a very limited time span. It would be too time consuming to process all frequently happening traces out of the log file and trying to render all of it would consume a lot of system resources.

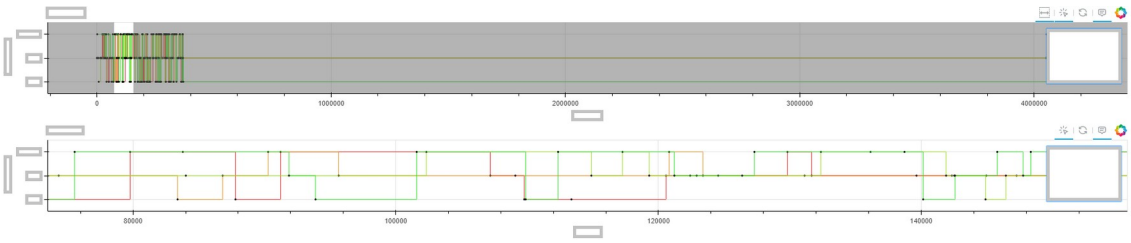

Figure 3: Changing viewing resolution.

Figure 3 shows how the viewing resolution can be altered on per trace or per plot basis. Only one plot was used in the figure to demonstrate. Essentially some rarely occurring event is plotted on the upper graph which displays the timespan of the entire log. The highlighted area is the time range of the lower graph. The highlight, operating sort of like a magnifying glass, can be moved around with interactive tools. Once the moving is done the “zoomed in” graphs also update to reflect the new time range. The highlighted area also moves to the correct place if the time range of the lower graph is altered, or if the range of any other graph which shares the time axis is changed by moving around.

5.1.6 Overflow handling

Interpolation capability is needed for traces which produce too many items to display either performance or visibility wise at once. It is possible that a single plot contains thousands or tens of thousands of datapoints, which all are individual items with their own tooltips and associated popup text data. The browser renderer will suffer from immense performance issues if everything is rendered at once and everything does not fit in the screen anyways. This performance issue is specific to our use-case and bokeh is capable of plotting millions of datapoints. (1) But as we need some extra data/functionality with each specific datapoint we can not use Python image

library or datashaders (13) to down sample our plots. This issue is solved by using the bokeh-server. It is the only reason currently why the tornado-based server is in use.

There is a callback implemented into the existing and custom tools like drag, pan, zoom, etc. All of them can change the view. The callback is part of the graph builder module and it was designed in a way to allow access to all plot data from all the graphs to be able to modify them. Bokeh plots use something called *column data source* to vectorize glyph and plot data. This CDS can be modified via a certain update() method while the plots are being displayed to change them. This modification is done in the callback method to remove all items which are out of bounds of the graph and from overlapping. Of items which are drawn too close together only one is shown. This hiding of too close together items is done by level of detail event which is a bokeh feature. All of this is done in the server side except for level of detail event so the browser never needs to handle too much data. This solution leaves at worst some hundreds of glyphs per plot to render, improving the page responsiveness.

6 ARCHITECTURE AND IMPLEMENTATION

Another internal tool is used to get input for this visualization. Below is an image (Figure 4) showing a very high-level processing flow of this project. The key steps in this overall tool flow are:

1. **Raw log parsing w**ith MTK internal log parser. This phase will create one or multiple csv files for visualization process. The formatting generally follows a certain template but is subject to change as new use-cases appear. This possible variation in the number of files and their content poses some complications in the csv parser design.
2. **Log parsing** process will read CSV files and organize data into desired format for actual visualization. This phase will also write the Plotting data into disk or local web server for visualization.
3. **Visualization in HTML** is the final state. This is just showing the pre-processed data in a web UI.

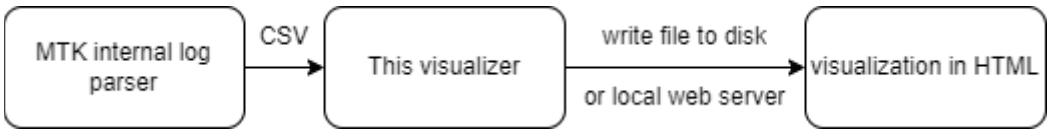

Figure 4: Information flow.

6.1 Architecture overview

Next the visualization tool architecture and SW modules involved in this visualization process are discussed more in detail.

Figure 5 illustrates the high-level architecture of the whole system. First from our point of view unprocessed log files are parsed and filtered by the tool this tool is part of. It produces a CSV file(s) and calls this tool with command line arguments the CSV file is then read by the parser module and the parsed data is fed to “graph_builder” module which generates the bokeh related document and displays it in the browser.

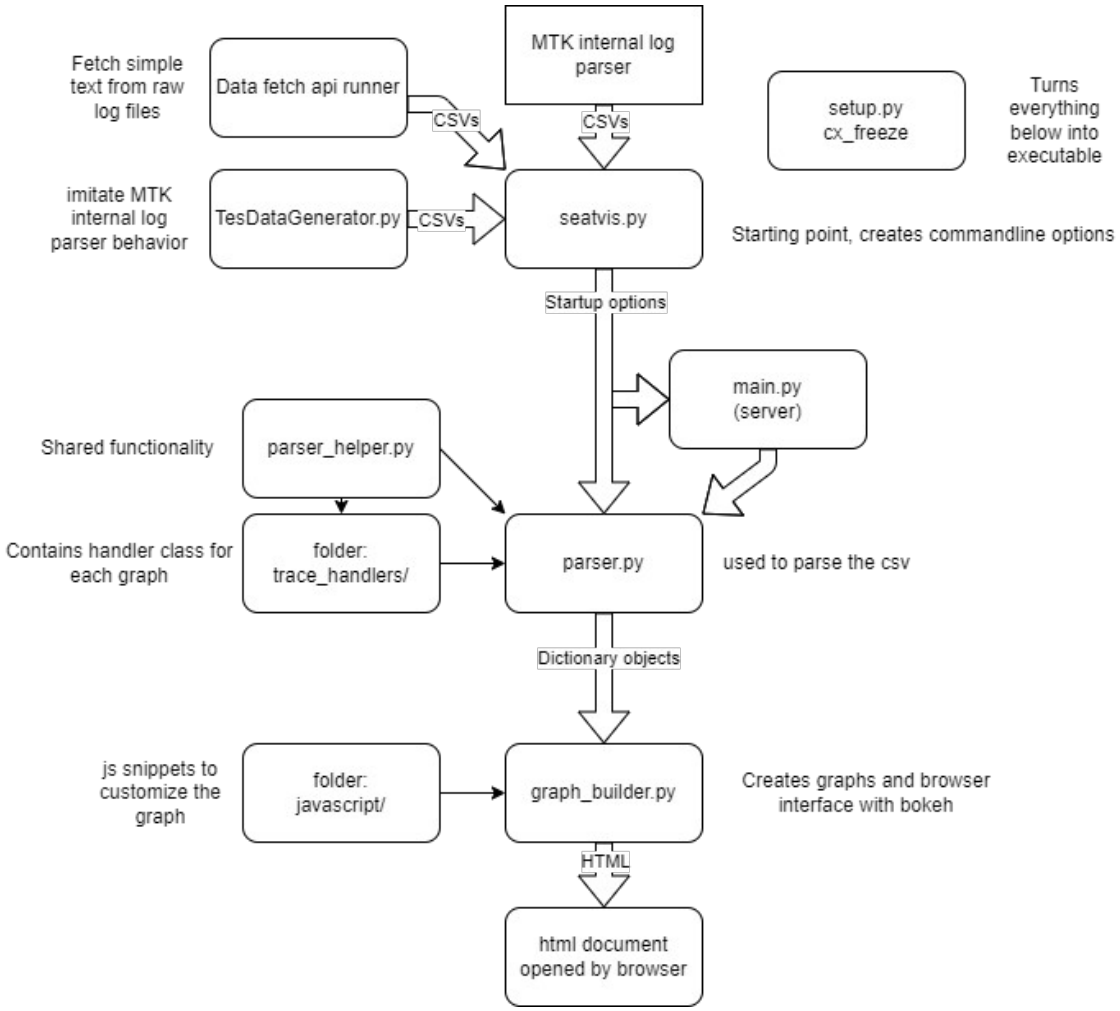

Figure 5: A high level flowchart.

6.2 Modules

This visualization software (Figure 5) consists of 4 main modules and a number of trace specific handler modules (Figure 6) in the trace handlers folder. The 4 main modules and some notable features are introduced below.

- Seatvis: seatvis in seatvis.py file, handles everything related to startup.
- Parser: seatvis_parser in parser.py, handles file parsing and trace options.
- Graph_builder: graph_builder in graph_builder.py, handles everything bokeh related.
- Data fetch api runner: And interface using internal log fetch API.

6.2.1 Seatvis

This module functions as an interface between the parser module and user. This user can be either a person using this from a command line or the MTK internal parser tool. The seatvis module is sort of start-up script; it takes in user input and sets all the settings the other modules need and finally starts the parser module once all other necessary modules are finished. Those other necessary modules include for example fetching some extra data or starting the web server.

6.2.2 Parser

This module is used to handle reading csv or any other files containing data to visualize. The files are read and passed through different parser sub-modules (Figure 6) depending on which parts of the data is wished to be visualized at that time. The sub-modules handle trace specific formatting and turn the data into python dictionary containing all necessary information for plotting.

In the parser design the code re-use and strong input format specification are essential requirements. These allow us to maintain this tool efficiently and make the long-term use and development possible.

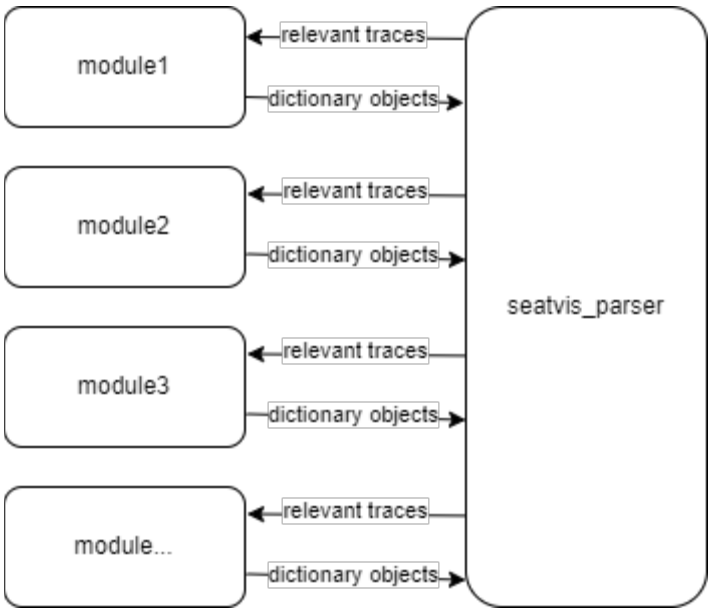

Figure 6: Modularity.

6.2.3 Graph_builder

This module takes dictionary objects generated by the parser module as its primary input. Runtime options and other settings are also relayed by the parser module, which constructs and calls objects from this module. This module does not make very many decisions about how the input data is processed, all of that is already done in the parser module. The point of this module is to turn the already processed data in the dictionary objects into html file for visualization. This mostly consists of creating graph and plot objects with bokeh framework and creating links to values and behavior between those objects. For example, a graph is an object and the time axis is another object inside it. Linking those time axes from different graphs is an example of what graph builder does.

6.2.4 Data fetch api runner

Data fetch api runner uses one MTK internal log parsing API to get text entries from an unprocessed log file. Unprocessed means the log file is still in MTK internal format which can not be read very easily. This API runner is used to parse the log file in cases where the log file is from some uncommon HW platform or incomplete and thus can not be parsed using the MTK internal log parser which is otherwise always used. This API runner does not have any features which are not essential, it simply just fetches trace entries and writes those to a CSV file for the parser to handle. It is used as a backup in non-automated use-cases.

6.3 Notable features

6.3.1 Bokeh server

Bokeh has a feature where a webserver can be used to transmit data between the created page and the program written in Python. This allows for all kinds of live data streaming and such

interesting use cases. We do not need this since we use the tornado webserver in a way which allows us to stream only the data visible in the viewport to the page to be handled by JavaScript running in the browser. This “single local user” is not necessarily the intended use-case for the tornado server. Because the bokeh webserver provides a lot of callbacks and app-hooks it can be made to work properly and invisibly to the user (9). Basically, when this software is started with the webserver enabled, all the software components are run by the webserver main program and some internal flags are set so that server only features can be enabled and the right methods for graph generation and displaying is used, as some parts work differently depending if a standalone html file is created or if interactive file is created.

There is an effort not to use this locally hosted web interface as starting the webserver takes time thus adding more seconds to the start-up overhead, which is bad because it causes more wait time for the user.

6.3.2 Handling clock wraps

The free running counter wrapping, commonly known as overflowing, and the problem it causes was introduced in the chapter 2.3. The problem occurs in the following way: The counter always counts up, so the number on the clock keeps rising and rising. But the clock does not have infinite memory to store the ever-increasing count so at some point it will run out of bits and flip back to 0(2). Figure 7 illustrates this visually. On the x axis going to the left is time going forward, and the numbers represent the count the counter is storing at any given time.

From the image one can see that a single count for example 1, can represent multiple moments in time. 1 will repeat again and again, separated by the time the counter takes to wrap.

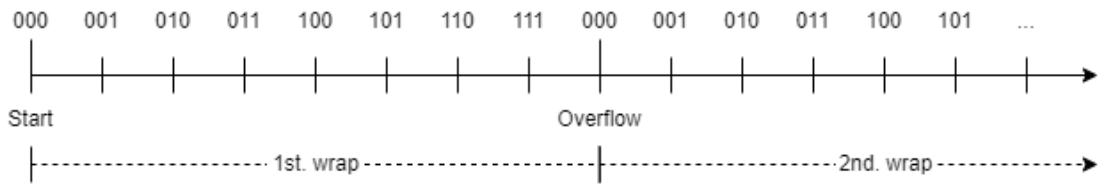

Figure 7: Counter wrapping.

Displaying the wraps is done by placing a red ray at wrap (Figure 8) interval to the graph and for the axis numbers there is a tick formatter, a bokeh feature, which modifies the axis using JavaScript. This tick formatter just takes modulo of the data value using the counter max value as divider and displays the result as axis value.

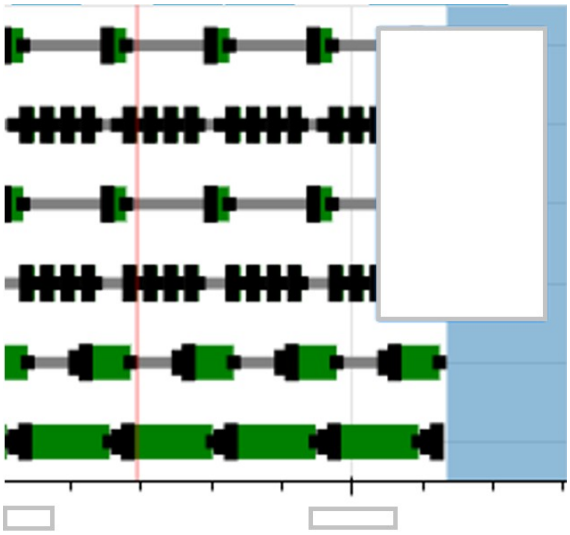

Figure 8: Visualizing counter wraps.

6.4 Graph design

Graph design is an important part of any visualization software which deals with plots. Next it will be discussed, how the visualizations are made and what different graphs look like, for example, what color choices were made and why certain colors were picked over others.

6.4.1 Colors

As typical shades of green are used to convey that something is on, powered up or used. Yellow is used for states of inactivity or powering up. Red is used to indicate errors or other items that are of importance. Black and gray means inactivity or power down.

Qualitative color sets are used to draw items for the most part, as most of the data to plot is categorical in nature meaning two things in one graph do not directly relate to each other.

Categorical colors are just colors that have the same saturation but different hue; this is so that one item does not look more important than any other (5). An example of this kind of colors in a plot can be seen in Figure 9.

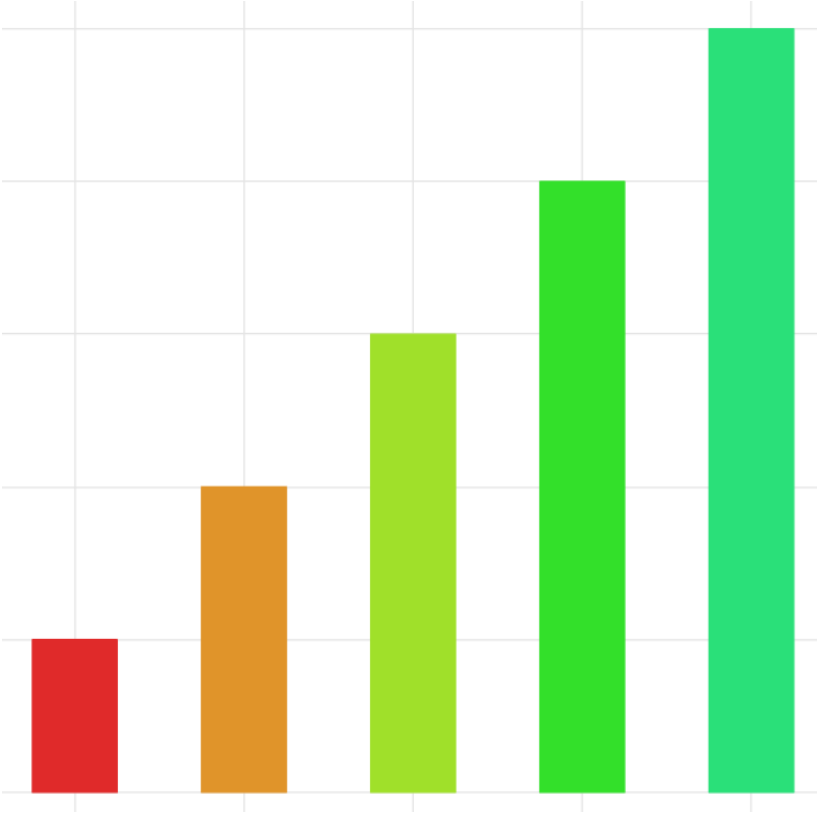

Figure 9: Example of a qualitative color set.

6.4.2 Step line plot

The point of step line plot is to clearly visualize state changes of a single state machine. One y axis level corresponds to a single state. These states are usually all known beforehand and thus predefined in the graph settings when programming some graph, but predefining possible states is not a requirement. In case the states are not defined the plotter will look through the data and add all states that are in the data and alphabetically orders them. If one wants order other than alphabetical, then it is necessary to define the states manually, as then the order can be manually defined. Step line plot visualizes state changes more clearly than a bar plot which basically just uses color to show what the state is at a given time. With differing colors and clear naming, it is

possible to have multiple state machines displayed in a single graph, but the readability quickly degrades as the number of lines increase. Figure 11 illustrates this loss of readability.

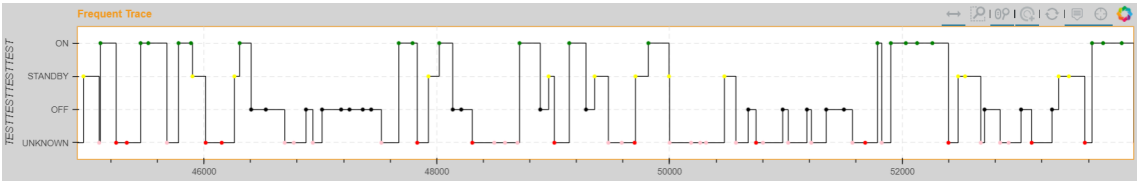

Figure 10: Readable line plot.

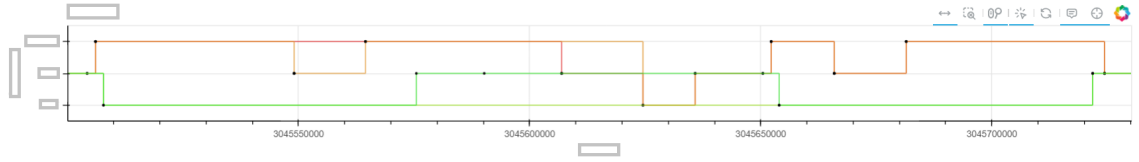

Figure 11: A bit less readable line plot.

6.4.3 Bar plot

Bar plot shown in Figure 12 can be used to visualize multiple state machines in a single graph in a very compact space, one y-axis level for one state machine and different states are then displayed using different colors and line widths. There are a few inconveniences here. First, it is not nearly as easily read as the step line plot is. The second problem is that this causes some performance issues. The way bokeh handles multiline plots, the glyph used to make bar plots is so that a new renderer is made for each line segment. This can cause some stuttering while dragging or panning the graph with some of the bigger datasets we need to plot.

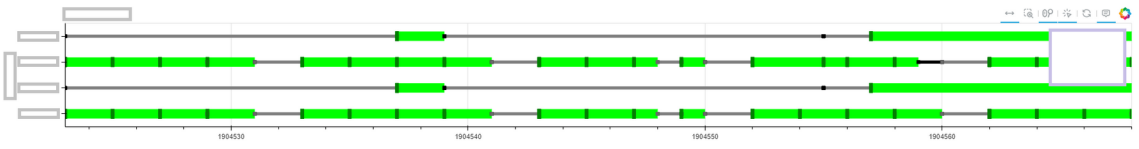

Figure 12: Example bar plot.

6.4.4 Glyph(marker)plot

Glyph plot is very simple xy-plotter which plots coloured markers in given co-ordinates. It can be used for any 2-dimensional data, additional information can be encoded in colour and shape. For example, in Figure 13 the data has three attributes: time, type and subtype. The type can only be

a whole number and the subtype is binary, so both type and subtype can be plotted on one axis. Marker just above one is type 1, subtype 1 and a marker just under one is type 1, subtype 0. The colour is just used to highlight the difference.

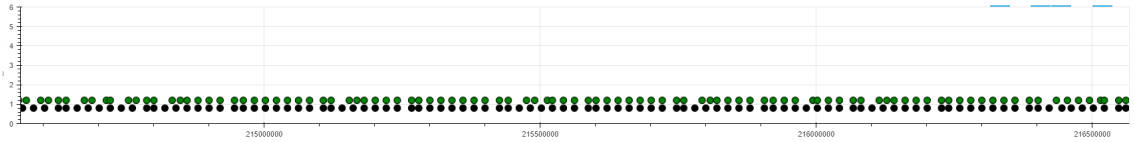

Figure 13: Example glyph plot.

6.4.5 Semaphore line plot

Semaphore line plot is used to visualize semaphore states. For example, it can show how many of certain commands are being processed currently. The semaphore is increasing in value as more commands come in and respectively decreasing in value as processing finishes. It is useful for visualizing the number of pending operations, memory consumption or queue lengths.

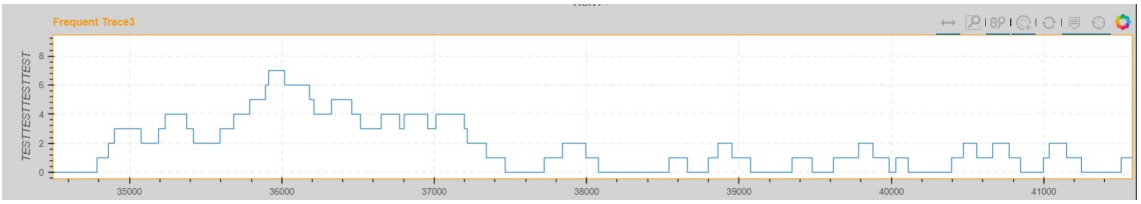

Figure 14: Example semaphore line plot.

6.5 Result

A few different use-cases for this visualizer are shown in Figure 15 and Figure 16. In Figure 15 bar graphs are used to display different state machines and the states at any given time. The bottom 3 graphs are linked via time axis as those share the same timestamps. All interactive actions are shared between graphs which share an axis, this is so that when the user is looking at an event in one graph, all the other graphs are moved to same position making switching

between graphs easier. All these use-cases basically are a collection of calls to trace handler modules (Figure 6). Which modules and in which order just changes between use-cases.

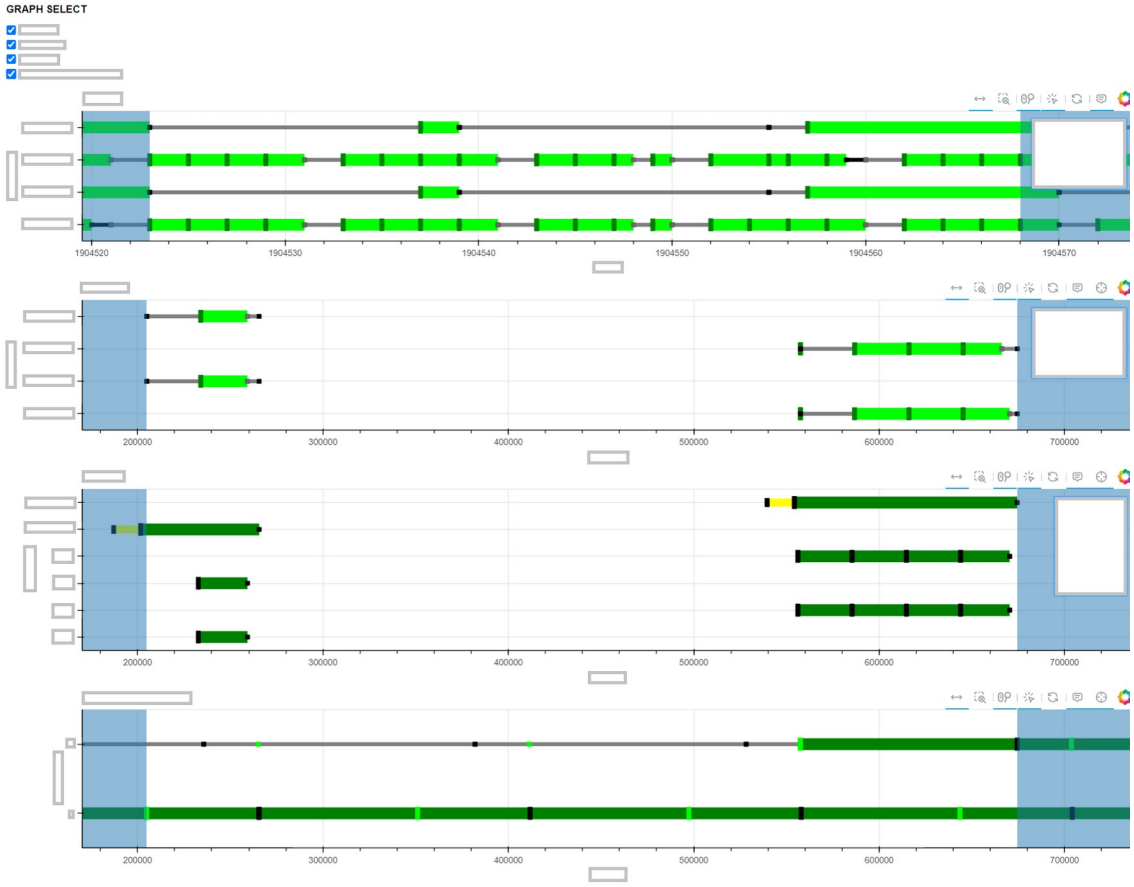

Figure 15: Example use-case 1

The graphs in Figure 16 are line and marker graphs. Here in the last graph a step line plot is used to represent states of a state-machine. Multiple state-machines are shown in this case. Marker graphs here are used to represent specific SW writes to a certain place and the graph with the green and black markers is the one shown in chapter 6.4.4. None of these images show a graphical user interface where one could input data. This is because there is none, MTK internal log parsing tool handles all of that and just provides command line arguments for this tool. The command line interface is the main interface used. It can be used by the MTK internal log parsing tool or the user in some specific use-cases.

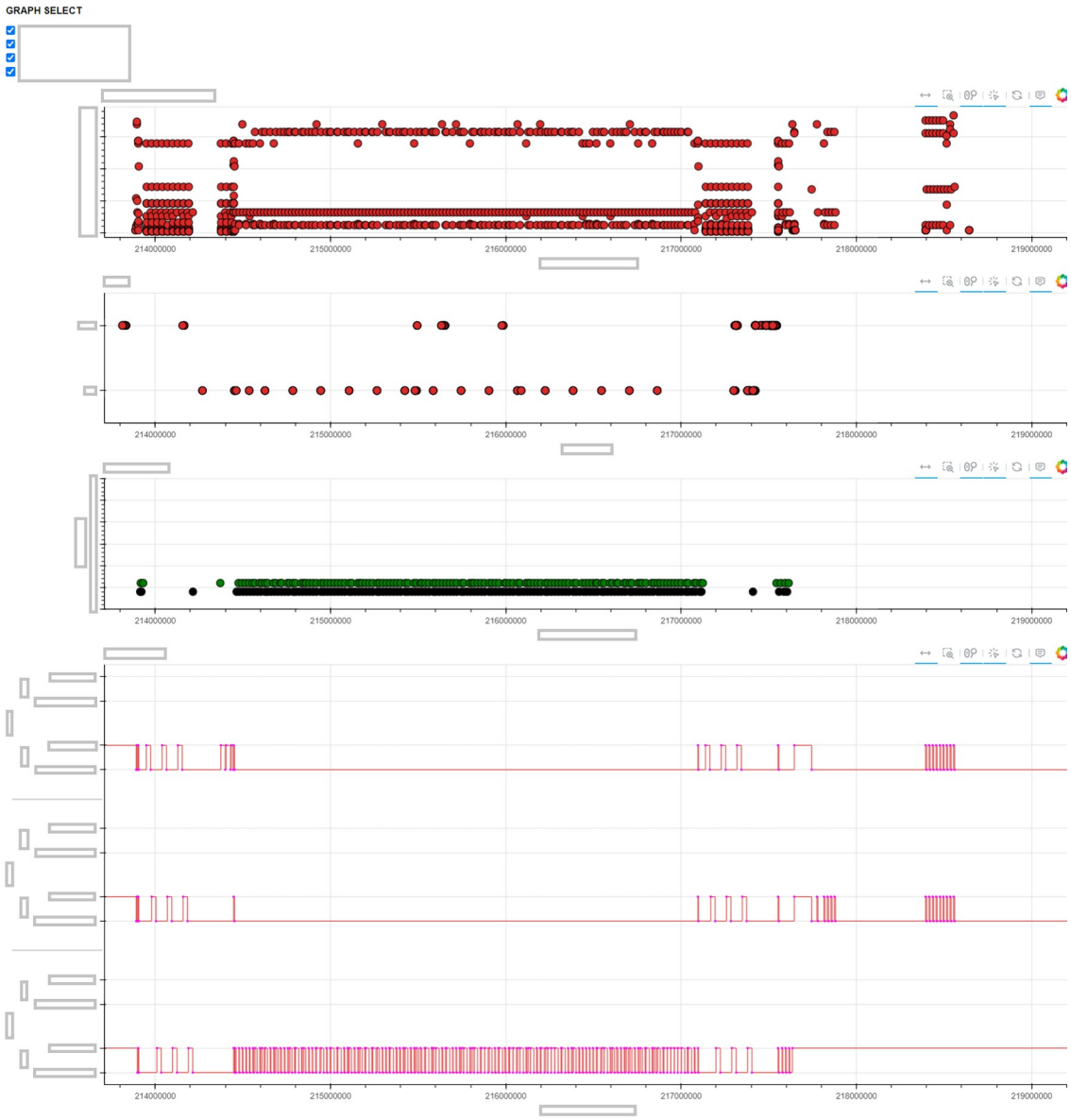

Figure 16: Example use-case 2

7 CONCLUSION

Generic visualization software was successfully implemented. It now works with the expected trace log files and creates graphs from those either automatically as a part of the tool which makes those files, or via a command line interface which is used as a backup should any new use-case arise. The user experience is what it was supposed to be; visualizations open up automatically after the plotter has finished running requiring no actions from user.

The software is maybe not as easily generic as we would originally have liked it to be. The addition of new graphs requires some extra work brought up by scattering of the code base. Consequently, it seems that the codebase itself is in need for refactoring. This is because many originally unique features ended up being more like universal features for many graphs and were thus moved around and merged to other originally unrelated code.

The greatest challenge during this work was to keep the plotter as generic as possible, not to lock down any features and not to rely too greatly on any assumptions about the input data. These challenges were encountered many times during development. New features of the input data had to be handled in a way that assumes the feature may or may not be available in the future. This brought some limitations in terms of processing power, as all things must be checked and then double checked, leaving this visualization software to be slower than it maybe could be. Most of these performance related issues can be solved just with better optimized code.

Work continues in terms of fixing any bugs which arise by adding new graphs and possibly new styles of visualizations. Additionally, new ways to optimize are being searched, for example exploration of solutions to avoid the use of the cx_freeze library. This could be done by depending on Python interpreter being available as a part of another software which this tool already depends upon.

REFERENCES

1. Santilli, Ciro 2019. A survey of open source interactive plotting software with a 10 million point scatter plot benchmark on Ubuntu. Cited: 20.11.2022 https://stackoverflow.com/questions/5854515/interactive-large-plot-with-20-million-sample-points-and-gigabytes-of-data
2. Luckyresistor 2019. Real Time Counter and Integer Overflow. Cited 20.11.2022. https://luckyresistor.me/2019/07/10/real-time-counter-and-integer-overflow/
3. Coates, Eric 2020. Digital Counters. Cited 20.11.2022. https://www.learnabout-electronics.org/Digital/dig56.php
4. Duarte, Marcelo 2022. Cx_freeze. Cited 20.11.2022. https://cx-freeze.readthedocs.io/en/latest/setup_script.html#setup-script
5. Yi, Michael 2019. How to Choose Colors for Your Data Visualizations. Cited: 20.11.2022 https://chartio.com/learn/charts/how-to-choose-colors-data-visualization/
6. Ku´c, Rafal 2022. Semantext. Cited 20.11.2022 https://sematext.com/blog/log-analysis-tools/
7. Dynatrace 2022. Cited 20.11.2022. https://www.dynatrace.com/monitoring/platform/log-management-analytics/?utm_source=google&utm_medium=cpc&utm_term=log%20analysis%20tools&utm_campaign=dach-im-log-management&utm_content=none&gclid=Cj0KCQjw166aBhDEARIsAMEyZh6dw1EU0xa7jlFrCWBKIp0RdWhBqAqikqqh3jueG6UOpdxGXPqDeusaAjYPEALw_wcB&gclsrc=aw.ds
8. Bokeh 2022. Lifecycle hooks. Cited:20.11.2022 https://docs.bokeh.org/en/latest/docs/user_guide/server.html#lifecycle-hooks
9. Bokeh 2022. callbacks Cited:20.11.2022 https://docs.bokeh.org/en/latest/docs/reference/models/callbacks.html?highlight=callback#module-bokeh.models.callbacks
10. Bokeh 2022. Call back user guide Cited:20.11.2022 https://docs.bokeh.org/en/latest/docs/user_guide/interaction/widgets.html
11. Matplotlib 2022. cited:15.12.2022 https://matplotlib.org/stable/gallery/lines_bars_and_markers/index.html
12. Bokeh 2022. cited:17.12.2022 https://docs.bokeh.org/en/latest/docs/reference/models/tools.html#actiontool
13. Bokeh 2022. cited: 15.12.2022 https://bokeh.org/

14. Bokeh 2022. cited 18.12.2022 https://docs.bokeh.org/en/latest/docs/user_guide/compat.html
15. Educba 2022. What is python bokeh? Cited 16.01.2023 https://www.educba.com/python-bokeh/
16. SHUBHAMSINGH10 2020. GeegsforGeegs. Introduction to Bokeh in Python. Cited 16.01.2023 https://www.geeksforgeeks.org/introduction-to-bokeh-in-python/
17. GeegsfoGeegs 2022. Python Bokeh tutorial – Interactive Data Visualization with Bokeh. Cited: 16.01.2023 https://www.geeksforgeeks.org/python-bokeh-tutorial-interactive-data-visualization-with-bokeh/
18. D'Angio Leon 2022. Real Python. Interactive Data Visualization in Python With Bokeh. Cited: 16.01.2023 https://realpython.com/python-data-visualization-bokeh/
19. Bokeh 2023. Bokeh. Cited:18.01.2023 https://github.com/bokeh/bokeh
20. Bokeh 2023. Readme.md. Cited:18.01.2023 https://github.com/bokeh/bokeh#readme