CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

A Post Processing Framework for Analyzing Test Data from Vehicles

Master's thesis in Computer Systems and Networks

ALFRED AGRELL

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2019

Master's thesis 2019

A Post Processing Framework for Analyzing Test Data from Vehicles

ALFRED AGRELL

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2019

A Post Processing Framework for Analyzing Test Data from Vehicles

Supervisor: Wolfgang Ahrendt, Birgit Grohe, Computer Science & Engineering
Advisor: Michael Eidland, Volvo Cars
Examiner: Andrei Sabelfeld

Master's Thesis 2019
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in Latex
Gothenburg, Sweden 2019

A Post Processing Framework for Analyzing Test Data from Vehicles
ALFRED AGRELL
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Today at Volvo Cars, thousands of logs are taken from the test cars containing massive amounts of data, in the Measurement Data Format (MDF). This data should be processed and analyzed. This thesis seeks to design and implement a system that can automatically analyze the vehicles' logs. There is a lot of prior work in related fields, but we weren't able to find anything as expressive as this system. Our work focuses on analyzing signal values over time, and correlating multiple simultaneous signals, using a flexible rule-based system based on a custom domain-specific language. This language is designed and developed for this project, and represents most of our contribution. An analysis consists of the user writing a number of rules in this language, representing (for example) "signal X may not remain zero during a consecutive 15-minute period" or "signal X may not be 4 or higher unless signal Y is 6 or higher", and telling the system to run said rules on an MDF file. The language interpreter is written in Python, using a small piece of C++ to interface with an MDF parser. We also implemented a manager process that coordinates the analysis processes, informs the user of the progress, and queues up work if the system has insufficient capacity to run more analysis processes. The system has enjoyed a positive reception at Volvo Cars, exceeding planned problem sizes already in the testing phase; the system could handle the increased load with only small changes, and still offers plenty of potential to increase performance and scalability even further.

Keywords: Automotive industry, post-processing, domain-specific language, rule-based language.

Acknowledgements

Thanks to my supervisors at Volvo Car Corporation, Anders Gustafsson and Michael Eidland, for providing this opportunity and assisting with the creation of both project and report; my supervisors at Chalmers, Birgit Grohe and Wolfgang Ahrendt, for moral support and reviewing the report; and the number 8.

Alfred Agrell, Gothenburg, February 2019

Contents

1
Introduction

When developing new cars, a part of the process is to perform a large number of tests, and collecting data from these tests. Efficiently analyzing the data can reduce the number of tests and the time for development, and thus cost.
Today at Volvo Cars, thousands of logs are taken from the test cars containing massive amounts of data. Unfortunately, this data is not processed and analyzed as much as it could be. One of the reasons for this is that much of the analysis work is done manually and that grasping the vast amount of data is very difficult for a human. Volvo Cars would benefit from a system to facilitate for the testers and an (at least partially) automatic system that indicates if something goes wrong.
Such a system will play an essential part in the work towards automation and Continuous integration (CI) at Volvo Cars.
It is also a step towards reducing warranty claims, improving product quality and reducing expenses. It may also yield environmental benefits due to, for example, needing fewer spare parts.

1.1 Problem Description and Aim

The task has been to design a system that can be used by testers to analyze data from the log files and take appropriate actions.
The log files contain all data transmitted across the vehicle's CAN and FlexRay networks [14], as well as data from auxiliary devices attached during testing (for example ampere meters). While the data amount is often reduced by disconnecting devices not currently being tested, a near-complete vehicle is also often tested.
Since many signals are measured hundreds of times per second each, and there are many components transmitting data, a full recording of one test rig is one gigabyte of compressed data per three hours, and there are many test rigs running in parallel, often overnight.
There are specifications for which values the signals may have for the component to be considered correct, called 'rules'. The rules may involve one or more channels. Sometimes the rules are exact, but they usually contain allowed tolerance, like 'if signal X on channel Y is between 5 and 6, signal Z on channel W must be between 4.9 and 6.1 at least once within ±100ms'.

The goal is to process this enormous amount of data automatically or semi-automatically, so the system must be easy for the testers to use. It is also important that the system developers should understand how it works, so it can be extended in the

future. If the users and developers understand how the system works, confidence in its correctness increases, and thereby the willingness to use it.

The data to be processed is stored in MDF-files. The abbreviation MDF stands for Measurement Data Format [1]. A typical file could be half a gigabyte, and there could be thousands of rules to be evaluated. Parsing those files fast enough is a challenge.
There is already a library with C++ functions for extracting certain information from MDF-files: MDF4-lib [1]. A part of the work was to write a Python plug-in to be able to access the MDF4-lib functions from Python.
There are certain rules that must be fulfilled to check if the car behaves correctly. The rules are about signals to different channels and that those signals have to be consistent. If they are not, then there is something wrong and the goal is to detect this.

1.2 Related Work

There is little research on checking rules in log-files, or ensuring correlation. The most recent publications about finding patterns in log-files usually use various machine learning and data mining techniques such as neural networks, Multilabel Naive Bayes Classification. To further speed up the analysis, efforts to use parallelism and specialized programming languages have been made, e.g. Breier and Branišová [8]. In our case, the use of machine learning methods seems suboptimal. The data in the log-files is well structured, and a rule-based method to extract information seems faster and more suitable.
Earlier research on log-file analysis uses less advanced methods. Fageeri and Ahmad present a binary-based approach for frequency mining in database log-files [7]. There are various approaches to fault-detection and finding anomalies or fault-detection log-files. A master's thesis from 2008 written by Memon [9] uses text processing based methods for both.
In the area of Runtime Verification [12], the focus is to determine if a system satisfies or violates certain properties, at runtime. Formalisms to express runtime verification specifications may use various methods, e.g. temporal logic (TL) [10], regular expressions, state machines, rule systems. A recent book about runtime verification [3] also lists a number of extensions, some of these taking into account a time aspect, e.g. interval TL and signal TL, as well as combinations of methods, e.g. TL and regular expressions.
Unfortunately, TL by itself is not applicable. TL can demand particular events to arrive in a particular order, but it cannot apply timing constraints, like 'if signal X is 5 at time t, signal Y must be 7 at $t + 4$ or earlier'; TL offers only 'statement P is always true' or 'signal X has the value F as soon as signal Y does', no specific timing constraints.
There are variants of TL, like interval TL and timed TL, which may seem more interesting. However, interval TL just considers pairs of points in time and whether they are ordered, and is fundamentally not more powerful than basic TL; timed TL seems to satisfy timing constraints, but (TODO: figure out if it can express 'signal X

is less than signal Y plus 7 as of 200ms ago'; figure out if it's implemented anywhere or if this fundamentally is a TTL implementation).
Signal TL sounded promising, since our tests are about signals, but it turns out that its focus lies in several signals at a certain point in time, whereas our task is about different signals at different points in time, and sometimes for a time interval.
There is an article from Eustace and Mukhopadhyay from 1982 that uses Finite automata to check rules for hardware design [6], but to our knowledge no similar work on timed version has been investigated.
Regular expressions (regex) are suitable for many purposes in text processing, but they cannot handle non-textual data, like our signals (long sequences of arbitrary numbers). While the signals could be translated to text, it is hard or impossible to express 'signal X is less than 5.4321' in regex, and we need considerably more complex operations than that (like 'signal X is less than {signal Y plus 3.1415}'). Even if we could solve that, the performance would most likely be unacceptable.
State machines are, like regex, good at processing one-dimensional sequences drawn from a finite alphabet. But like regex, they are less good at infinite alphabets (like floating-point values), and given our need to correlate multiple signals over time, it is unclear which states exist, or even what would constitute a state transition.
The LARVA tool [11] appears to be an extended version of state machines, enriched with variables. However, it is deeply tied to Java and real-time monitoring, neither of is interesting to us, and does not appear to provide anything useful outside those fields, and fundamentally doesn't solve the issue of defining what a state is.
The rule system LogFire [13] looks like a generalization of our work. Like our work, LogFire has a Domain-Specific Language (DSL); unlike our work, LogFire is Scala-based and can represent considerably more complex conditions. However, LogFire is designed for real-time analysis and looks at all signals simultaneously, yielding unnecessarily high memory use; said generalizations increase complexity and reduce performance; and it is written in the Java-based Scala, which we are not using since it does not fit easily in the ecosystem of the company.

1.3 Challenges

An essential element in the problem is the vast amount of data, billions of samples continuously collected by the testing system. The data has to be processed fast thus the methods must be relatively simple.
The data and the set of specification rules has to be described and modelled in an appropriate way. We aim for a rule-based method which allows us to take time into account. To find an expressive enough model that is uncomplicated enough will be the main challenge in this project.

1.4 Limitations

There are a few operations that cannot be expressed in the current language, and would be hard to add. One of the main two are 'assert that Sig_X on Chan_Y

becomes one of [1,2,3] if ...', which would require extending the already-messy assert_one further.
The other one is an assumption that during a range, every signal is sampled at least once; violating this assumption currently yields various unexpected and undesirable results (for example assert_all passing, but assert_one failing). This started as a performance optimization, but is heavily ingrained in the system (the timestamp/range distinction is caused by this assumption) and would be an enormous task to simplify.
The company's testers need a graphical interface to the tool. This was created by other Volvo Cars employees, and is not considered part of this work.

2
Approach

In this chapter, we describe the main ideas on how to solve the problem described in section 1.1.

The system should be a bridge between the tested components' data, and the personnel who needs to know whether the components work as intended or not. It is important that the system can efficiently extract the necessary information.

It is also important that the system is flexible regarding new or changed specifications. This is done in two ways: First, the rules are structured in a flexible way, with any number of filter commands (see below), such that most new specifications can be satisfied by writing more complex rules. Second, the system is designed to make it easy for the system's maintainer to implement new commands of all three kinds described below, if the existing commands are insufficient to implement the new specifications with satisfactory performance.

An important part of the problem is the time aspect. Many of the rules involve relations between different signals at different times. Modelling those relations with existing methods (see section 1.2) appears to be impossible or inappropriate. Most or all existing methods are designed with no or an oversimplified model of time, for finite alphabets, and/or for only one channel; adding these aspects would require considerable efforts and/or sharply reduce efficiency, to the point of not offering a usable result.
When designing our system, we needed to create a more powerful approach; one that can analyze signals over time, and correlate multiple different signals, to express rules like "signal X may not remain zero during a consecutive 15-minute period" or "signal X may not be 4 or higher unless signal Y is 6 or higher". To our knowledge, no other system offers both of these; we got better results by inventing our own thing.

We haven't found any existing model that satisfies our demands, let alone a satisfactory implementation. Therefore, we made our own; this allows us to implement all features Volvo Cars wants, while excluding the ones they don't need, as simpler systems tend to yield better performance and maintainability. It also allows us to choose the implementation language that best fits into Volvo Cars' existing ecosystem, namely Python.

The system is built on a rule-based approach. This was the most natural choice, as the signal specifications are given as rules.

The rules can be written in multiple ways. One is called the fluent syntax, see chapter 3.2.1. The fluent syntax looks like English and is easy for users to read and write; it's then translated to another format, the arrow language, which has a more mechanical syntax and is more suitable for internal processing. Alternatively, the user can use the arrow language directly.
We created a domain-specific language, which implements a number of commands. They can be combined into a program which implements a rule. We named it the *arrow language*, due to the >> symbols used as separators between the commands. At a more abstract level, a program looks like this:

```
source command
filter commands (optional)
assert command
```

A program consists thus of three parts: a source command, zero or more filter commands, and finally an assertion command. The structure is similar to implications. In the arrow language, each command passes a set of points in time, or (potentially overlapping) time intervals, to the next command (except the sources and assertions, which take no input, and emit errors rather than timestamps, respectively).
For example:

```
source channel_Q
>> ifinstant channel_Q signal_R 3 4
>> widen 0ms +100ms
>> assert_one channel_Y signal_X 5 6
```

This checks the value of signal R on channel Q. If, at any point, this is between 3 and 4, the value of signal X on channel Y should be between 5 and 6 at least once between 0 and 100 milliseconds from signal R reaching the desired value. If signal X never reaches the expected value, the assert_one emits an error message.
More technically, the source command emits a list of every point in time where channel_Q is sampled. ifinstant removes those where signal_R is less than 3 or greater than 4. widen turns each timestamp T into the range [T, T+100ms], and assert_one checks that signal_X is between 5 and 6 at least once during each of its input intervals; if not, it emits an error.
All source commands start with 'source', and the assertions start with 'assert'; between them, any number of other commands may exist. For more details, see chapter 3; a complete list of allowed rules can be found in section 3.2.2.

3

Arrow and Fluent

In this chapter, we describe the various parts of the system, including our domain-specific language. Since the log files are huge, parsing is a separate issue, see section 3.1. The rules need to be understood and modelled, see section 3.2. Section 3.3 presents a schematic overview of the system, and a description of the modules. We have also described the chapter contents.

3.1 Environment

The choice of programming language is important when speed is relevant. The current parsing is done in Python. It could be made faster by translating parts of it to C++, but the current Python version is fast enough for Volvo Cars.

The files are too large to be processed in one go; one of the files contains 6880 signals across 346 channels. Some channels only have a few different signals, e.g. a time-signal, while others can have over one hundred signals.

Therefore, it will be necessary to split up parsing/processing. The files are sorted by date/time. Furthermore, there are indications that signals from the same channel are close to each other in the files. A possible approach could be to look at a few channels at a time. Another possibility would be to extract information for one or a few signals at a time. We chose the latter approach.

3.2 Rules

3.2.1 Fluent Syntax - A user friendly rule specification language

The user has the option of writing rules in something we call the 'fluent syntax' or 'fluent language'.

The fluent syntax is a more human-like language where the users can write down what they want without having to worry about the syntax. All rules in the fluent syntax start with 'assert that'.

Sample rules:

```
assert that EngineStatus on CANCore1 is 6 if
LongitudeDelta on FlexRay3 is greater than 1.5

assert that EngineStatus on CANEngn3 is 6 after 20 ms if
EngineStatus on FlexRay3 becomes 6
```

The first one is a simple if-then rule. The first rule says that if the signal LongitudeDelta on the channel FlexRay3 is greater than 1.5, then the signal EngineStatus must have the value 6 (which means "active driving") on channel CANCore1; if the former is true, but the latter is not, the rule fails and the user is alerted.

One could imagine using state machines to model the rules.

However, there are also other rules that involve a time perspective, e.g. the second example in the box above:

If the signal EngineStatus on channel FlexRay3 becomes 6, then the signal EngineStatus on channel CANEngn must also be 6 at the point in time that is 20 milliseconds later than the EngineStatus signal on channel FlexRay3.

For the second kind of rules, state machines and finite automata [4] are not enough. Possibly, a timed automaton [5] can be used, or a method listed in [3]; however, it's unclear what performance characteristics that would provide.

3.2.2 The Arrow Language - A domain specific language for analyzing log data

The user's other option on how to write rules is the arrow language. It is the core of the system; the fluent syntax is translated into the arrow language prior to execution. The arrow language was created earlier than the fluent syntax.

The first step of designing the arrow language was determining which features it must be able to represent. To do this, we asked the Volvo Cars personnel to describe the operations they require in any human-readable, unambiguous language (which we never intended to implement an interpreter for), and they provided the following:

```
for each instant of signal_R::channel_Q:
    if value == 3:
      within -100ms .. 100ms:
        assert_one signal_X::channel_Y == 5
        assert_all signal_X::channel_Y in [ 4 . . . 6 ]
        assert avg in [4 . . . 5]
```

```
every time ActiveUB is 1 and Active is 3,
    PropulsionUB should be 1 and Propulsion should be 5 at least
        once within ±100ms
```

```
every time signal R is 3, signal X should be 5 at least once within (-100ms . . . 100ms)
```

```
assert avg(signal_X::channel_Y) in [5 ... 6]
```

We analyzed the rules (and asked some additional questions - some of them are ambiguous) to determine the minimum required complexity of a domain-specific language that can represent all operations. We produced a language that has several 'commands' (for example ifinstant), where each one takes a set of timestamps (each of which is either a single point in time, or a time interval) and applies some transformations (for example discarding timestamps where a specific signal is outside some specified range) to them, before passing it to the next command. For example:

```
for each instant of signal_R::channel_Q:
    if value == 3:
      within -100ms .. 100ms:
        assert_one signal_X::channel_Y == 5
        assert_all signal_X::channel_Y in [ 4 ... 6 ]
        assert avg in [4 ... 5]

translates to

source channel_Q                                    // all points where channel_Q is defined
>> ifinstant channel_Q signal_R 3 3 // the points where signal_R is >= 3 and <= 3
>> widen -100ms +100ms                              // 200ms ranges, possibly overlapping,
                                                // each centered on a point where signal_R is 3
>> assert_one channel_Y signal_X 5 5                    // in each such range,
                                    // there must be a point where signal_X on channel_Y
                                        // is simultaneously >= 5 and <= 5 (i.e. equals 5)
source channel_Q
>> ifinstant channel_Q signal_R 3 3
>> widen -100ms +100ms
>> assert_all channel_Y signal_X 4 6
            // in each range, signal_X on channel_Y must always be >= 4 and <= 6
source channel_Q
>> ifinstant channel_Q signal_R 3 3
>> widen -100ms +100ms
>> assert_avg channel_Y signal_X 4 5
                        // the signal's average value in the range must be >= 4 and <= 5
```

To simplify the syntax definition, there is no way to say 'send the output of this command to two different places'; however, the interpreter automatically detects shared prefixes and doesn't do duplicate computations. For example, given the above three subprograms, only one each of the source/ifinstant/widen commands are executed.

The following commands/filters exist:

- source channel_X

- sourcerange channel_X
- >> assert_all channel_X signal_Y 1 2
- >> assert_avg channel_X signal_Y 1 2
- >> assert_one channel_X signal_Y 1 2
- >> assert_one_not channel_X signal_Y 1 2
- >> groupeq channel_X signal_Y
- >> ifavgrange channel_X signal_Y 1 2
- >> ifinstant channel_X signal_Y 1 2
- >> ifinstantany channel_X signal_Y 1 2 3 4 5 6
- >> iflen 1000ms inf
- >> left
- >> nextin channel_X
- >> skipfirst
- >> widen -20ms 20ms

The programs always start with 'source' or 'sourcerange' and end with one of the asserts, with zero or more other commands in between. Since most filters start with >>, we've named it the arrow language. The commands can be grouped in different categories:

Start commands

Each program starts with one of the two start commands, source or sourcerange. The first one transmits every point where the signal is defined, while the latter transmits a single range covering the entire signal.

source channel_X
One of two possible first commands, passes each sample in that channel to the following program.

sourcerange channel_X
The other possible first command, passes the signal as a range.

Assert commands

The last command is an assertion. They take intervals from the previous command, and some channel arguments, and check if the relevant signal is within the desired interval. If an assert is reached and its condition is false, an error is printed.

There are many different assertions: assert_all, assert_one, assert_avg (average), assert_one_not, see the box below:

>> assert_all channel_X signal_Y 1 2
The asserts are the only allowed last commands. They all take ranges.

_all ensures that every value in the given range on that signal is within the given interval (inclusive; the two ends may be equal).
Every interval can have one or both ends specified as '-inf' or 'inf', respectively, with the obvious meaning; 'less than or equal to 100' is the interval '-inf 100' (strictly less than is '-inf 99.999').

>> assert_all channel_X signal_Y 1 2 || channel_X signal_Z 1 2
Ensures that all values in the range matches at least one given clause; it's allowed for different ranges to match different clauses.
Channel must be same between them, signal can vary.

>> assert_avg channel_X signal_Y 1 2
Calculates the average of each range, then ensures the average is within the given interval.

>> assert_one channel_X signal_Y 1 2
Ensures that at least one value in the range is within the given interval.

>> assert_one channel_X signal_Y 1 2 && channel_X signal_Z 3 4
Ensures that at least one value in the range matches all given rules.
Like assert_all, channel must be same between them; signal can vary.

>> assert_one ch_X signal_Y 1 2 || (ch_X signal_Y 2 3 && ch_X signal_Z 3 4)
Ensures that at least one value in the range matches the given boolean expression.

>> assert_one channel_Q signal_R << atstart channel_X signal_Y -0.1 0.1
At one point in the given range, signal R must equal what signal Y is at the start of that range.
The last two numbers define the tolerance: signal R must be equal to signal Y, plus the tolerance range (-0.1 .. 0.1).
Tolerance is optional; if omitted, the signals must match exactly.

>> assert_one_not channel_X signal_Y 1 2
Takes the same arguments as assert_one.
Ensures that at least one value in the range fails the condition.

Filter commands

The language contains many filter commands. Those filter commands take timestamps or ranges, and discard the ones not matching a condition. There are several variants, for example ifavgrange, ifinstant, ifinstantany, iflen, see the list below.

>> ifavgrange channel_X signal_Y 1 2
Takes ranges as input.
If the average of the signal value is within that interval,
the command passes that range to the rest of the program;
otherwise, discards the range, ignoring all asserts further down.

>> ifinstant channel_X signal_Y 1 2
Like ifavgrange, but it takes instants instead.

>> ifinstantany channel_X signal_Y 1 2 3 4 5 6
Like ifinstant, except it passes on the instant if it's in any of the ranges [1..2], [3..4] or [5..6].

>> iflen 1000ms inf
Takes ranges.
If the range's length is within the given interval, returns it; otherwise returns nothing.

Transformation commands

The transformation commands always emit one timestamp or range for each input, for example left (takes ranges, emits the left endpoint of said range), nextin (takes timestamps, emits the next point where a given channel is sampled) and widen (takes a timestamp or range, embiggens it by the given amount, and emits it).

>> left
Takes a range, returns the earlier bound of that range as an instant.
Most commands taking instants expect the channel to be sampled at exactly the point the instants refer to; this command often emits samples not aligned with any channel, which most subsequent commands reject. The exceptions are 'widen' and 'nextin', which don't need their instants aligned anywhere; and 'groupeq', which emits ranges aligned to a channel, causing left's output to be aligned as well.

>> nextin channel_X
Takes an instant and returns the next defined instant in the given channel, which may be same as an input.

>> nextin channel_X 20ms
like normal nextin, but adds 20ms before finding the next defined instant

>> widen -20ms 20ms
Takes instants or ranges, returns ranges.
The returned range is same as input, but the two timestamps are added to the start/end of the range. They can be positive, negative, or zero; this allows extending, moving, and even shrinking the interval (though the latter is not recommended).
If the input is instants, it's treated as size-0 ranges (prior to extending, obviously); if the resulting range is outside the recording, it's clamped. Returned ranges can overlap.

Correlation commands

Some commands' output depends on not just each individual input element, but also on prior inputs. They are skipfirst (emits all inputs, except the first one) and groupeq (emits intervals in which the signal doesn't change).

>> groupeq channel_X signal_Y
Takes a sequence of instants, and stores them for later.
Once the recording ends, the instants are grouped up into the minimum possible number of ranges, where the signal value is constant within each range, which are then passed along to the rest of the program.

>> skipfirst
Takes instants or ranges, and returns the same type.
If this is the first instant or range this command has seen, it's discarded; otherwise, it's passed on.

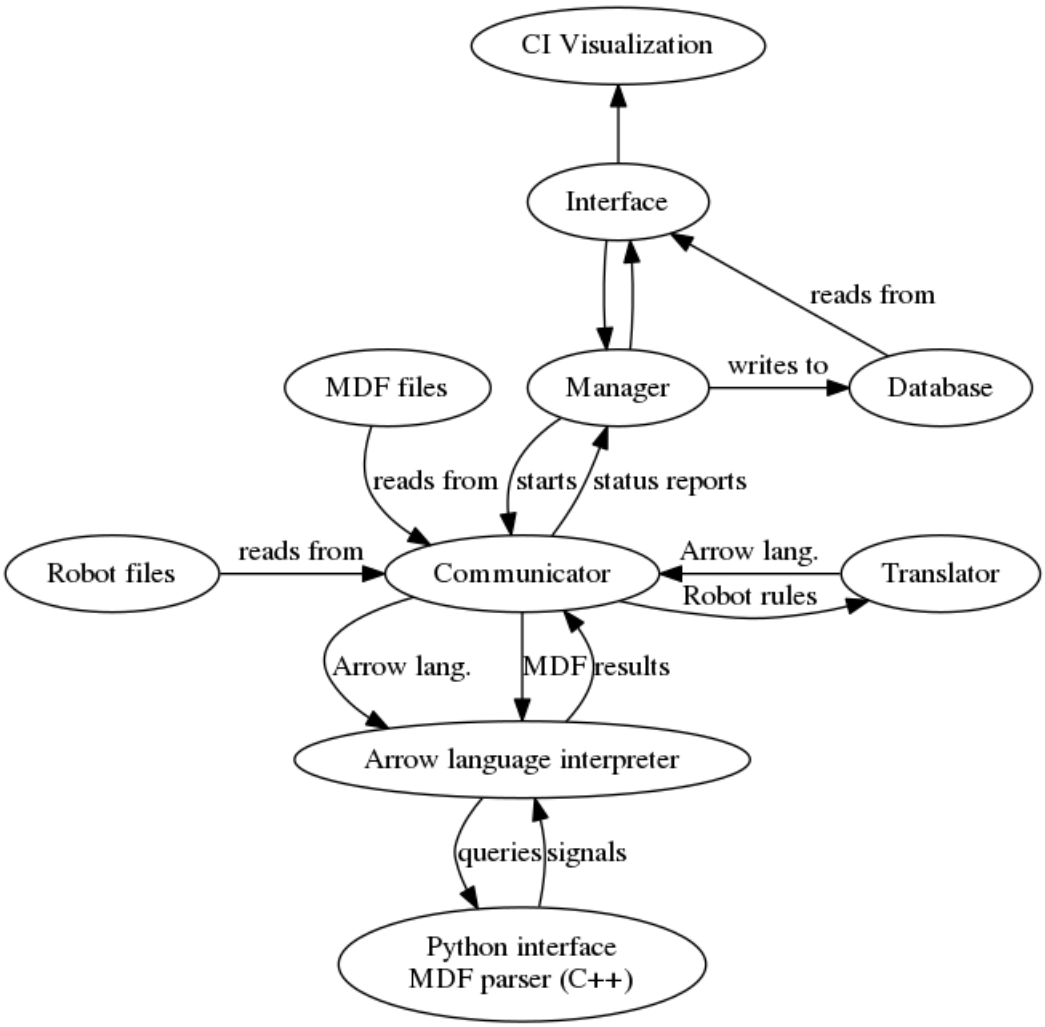

Figure 3.1: Overview over the system

3.3 System Overview

The system used for this project consists of a number of modules, see Figure 3.1. The modules are described below.

Interface

The interface presents test results to the user, see Figure 3.2. One can see 3 ellipses; one green, telling how many tests in the file passed; one red, telling how many failed; and one yellow, telling that the relevant event never happened. If a rule says "if signal X is 4, signal Y should be 5", but signal X never is 4, it becomes yellow.

A robot file contains a set of programs in the arrow and/or fluent languages. A more detailed view is available for each robot file (see figure 3.2), telling which rules failed and when they did. (Only the first failure is recorded per rule.)

All data comes from the manager process.

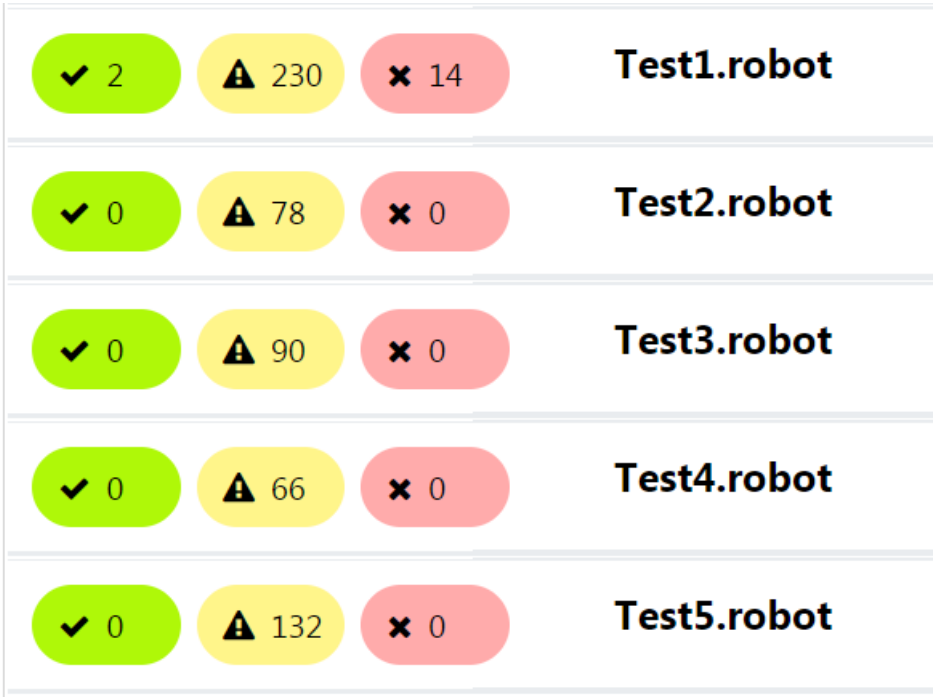

Figure 3.2: Screenshot of the interface

Manager

The manager process is responsible for starting and coordinating multiple simultaneous analysis processes, queueing up work if the system is overloaded, recording results to the database, and informing the user of progress and results.

Database

The database contains information about finished runs; ongoing ones are in the manager.

Robot files

The robot files contain rules in the fluent language, see section 3.2.1.

Communicator

The communicator is responsible for coordinating a single analysis. It reads rules from the robot files and sends them to the translator, then sends the translated rules (along with the MDF files) to the arrow language interpreter. It also sends results back to the manager.

Translator

The translator converts the robot files from the fluent syntax to the arrow language. The arrow language interpreter takes the outputs from the translator and the MDF parser and checks if the MDF values match the output rules. It takes a while to

execute, so it sends progress reports to the communicator, which sends them to the manager, from which the interface can view them. The communicator is also responsible for downloading the robot and MDF files and sending them to their respective modules.

Arrow language interpreter

This module does the actual analysis. It accepts an MDF file and some arrow language rule, and checks if the rules are fulfilled. It utilizes the Python interface to the MDF parser to read signal values from the MDF.

Python interface and MDF parser

The MDF parser[1] is written in C++ and reads data from the MDF files. We did not write this component.
The Python interface is also written in C++. It takes the lists of numbers that the MDF parser outputs, and converts them to a form usable from Python.

CI (Continuous Integration) Visualization

This module was written by Volvo Cars employees, in parallel with this work. It is part of Volvo Cars' system for monitoring and processing automotive information.

4

Results and Evaluation

Prior to this work, testers at Volvo Cars did not have the option of executing automated tests to verify signal and system performance on long recordings, while retaining accuracy. A few such tests were run, by visualizing the signals in a graphing tool (figure 4.1), but this took the tester an entire workday even for smaller MDF files. This means very little testing was done, and lots of data remained unanalyzed. Our system can run such tests in just a few seconds, making them feasible and worthwhile, and opening the possibility of more complex tests (figure 4.2).
Volvo Cars' MDF files are generally recorded from so-called boxcars, a few car parts scattered on a table and given synthetic input. They were, and continue to be, used with other tools at Volvo Cars, and their relevance to real vehicles is sufficient for those tools; as such, we assume they're also sufficiently relevant for this tool.

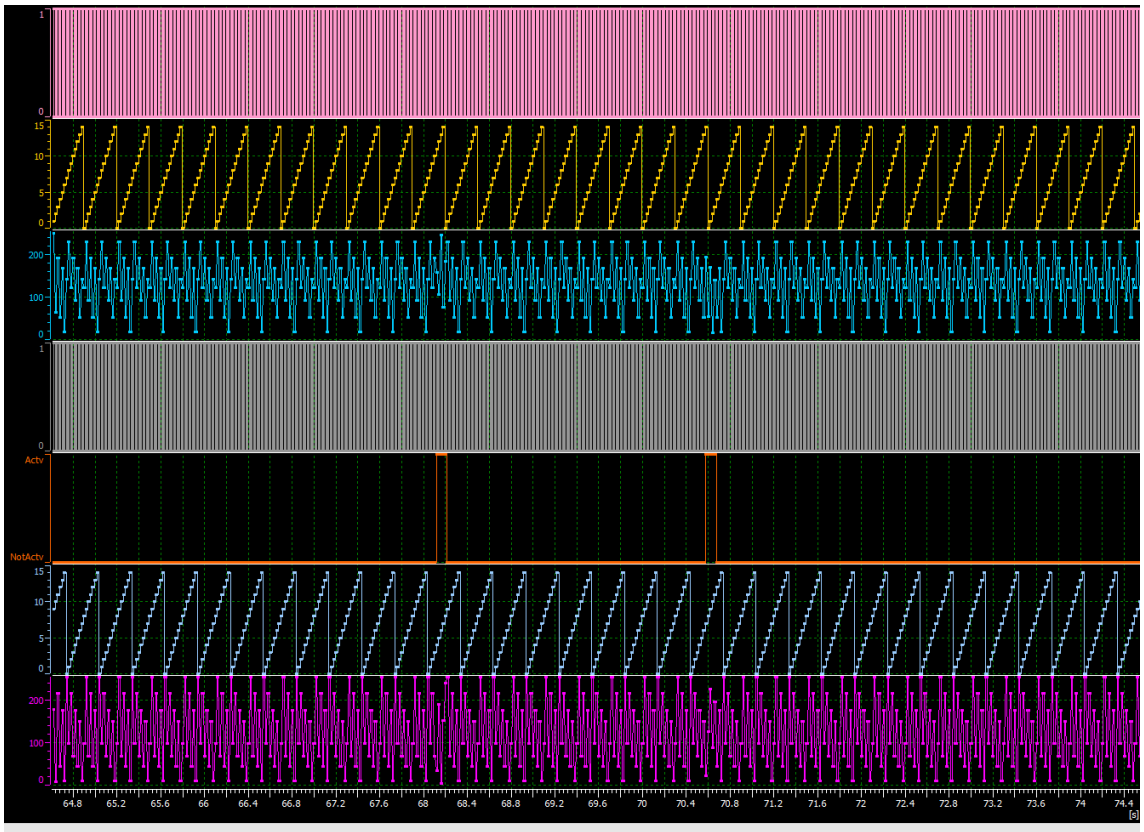

Figure 4.1: CANalyzer, the graphing tool used at Volvo Cars. Due to confidentiality requirements, various names are redacted.

Additionally, a few MDF files are recorded from real vehicles on the road.
The MDF files typically contain the speed of each wheel, whether the engine is on, the oil level, the blinkers' status, and various other vehicular parameters; all of them are measured over time, in regular intervals.
The system includes a few unit tests, which test all parts of the arrow language; this has caught several bugs, which were then quickly fixed. However, there are currently no tests for Fluent; this is a potential improvement.
The system (the arrow language, the fluent syntax, and the GUI) has been tested and used by the supervisor at Volvo Cars, a few of his colleagues, and some people from other departments. The users generally prefer to use the fluent syntax, rather than the arrow language directly; they have also found a few system descriptions in other formats, and made scripts that convert that into the fluent syntax, often hundreds or thousands of rules. While we had expected the system to be used with one or two hundred rules, the system only needed minor changes to accept the increased problem size. Should the need arise, the system still offers plenty of potential for improving performance and scalability.
The users are satisfied with the speed; there have been speed improvements, but none requested by the users. The users have, however, requested new functionality. For example, 'assert that signal X mirrors on channel A and channel B within 50 ms

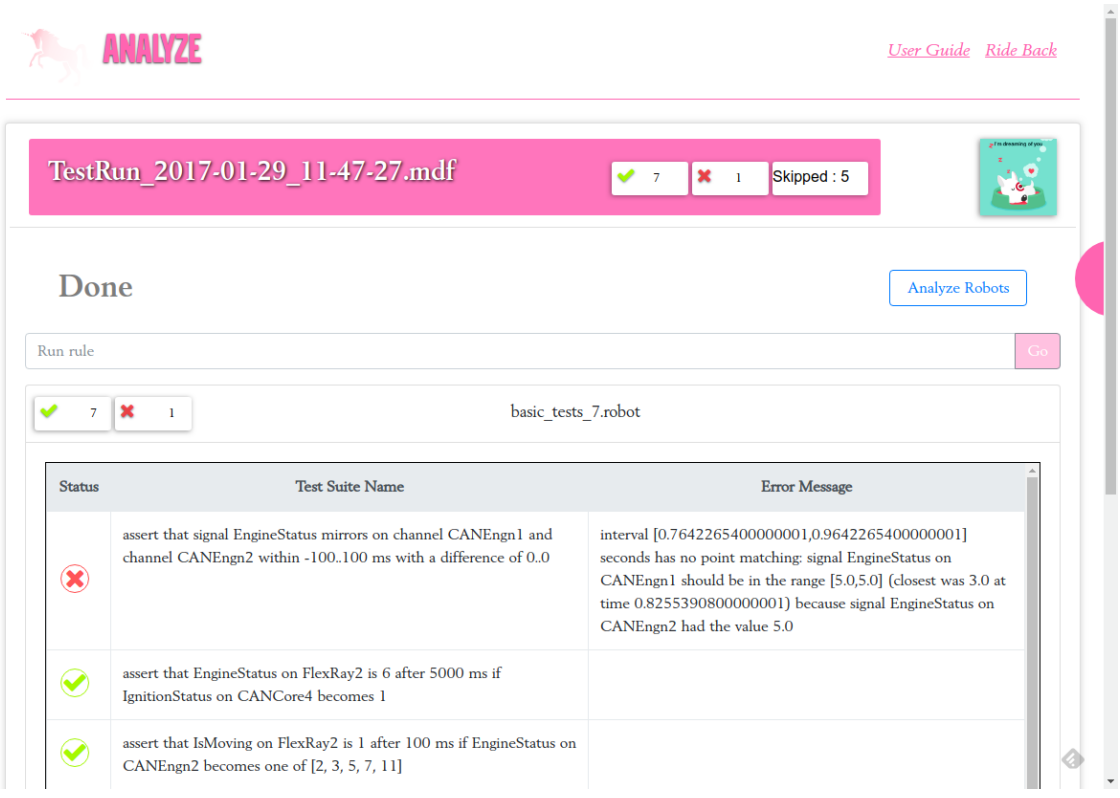

Figure 4.2: Our new system, showing a few rules in Fluent analyzing four channels (CANEngn1, CANEngn2, CANCore4 and FlexRay2) and three signals (IgnitionStatus, IsMoving and EngineStatus, the latter being present on three different channels), and whether the rules are satisfied. Like the above, numbers and names are fake.

with a difference of -0.1..0.1', which ensures that the signal exists and has the same values (modulo rounding errors and signal propagation delays) on both channels; this was implemented by creating the advanced assert_one forms, and teaching the fluent translator to process that.
Another user request is "assert that sig_1 on chan_1 is one of [0, 2, 5, 7] within 0..100 ms if sig_2 on chan_2 becomes one of [0, 3, 4]", which caused the creation of the » ifinstantany arrow language command. A third user request is 'becomes' clauses, such as "assert that sig_1 on chan_1 becomes less than 2 within 0..5000 ms if sig_2 on chan_2 becomes less than 4"; to implement this, the fluent language parser was taught to emit multiple arrow language programs, but no new arrow language commands were necessary.
The users found a few bugs during testing; some were fixed, some were user errors. At least 30 MDF files have been tested by us, plus an unknown number by other users. Smaller files and rulesets finish in a few seconds; larger take up to an hour (our largest MDF is three gigabytes, but we believe even larger exist and would work). The users are satisfied with the result; other programmers at Volvo Cars are satisfied with the internal architecture, and believe it has great potential.
The runtime is 10-15 minutes for most inputs. This could be improved by translating parts of the system to C++, but Volvo Cars is satisfied with its current performance, so we didn't do this. The languages C++ and Python were used to fit into Volvo Cars' ecosystem, which was a requirement.
The efficiency is also affected by the choice of data structures and algorithms. Luckily, the common ones (array, binary search) were sufficient for this task.

5
Discussion and Conclusion

5.1 Discussion

The arrow language was designed with several opposing constraints in mind: It must be able to represent as many operations as possible, but it must also be sufficiently constrained that rules can be executed efficiently, it must be simple enough that we can implement it in the time allocated for a thesis, and it must be simple enough that Volvo Cars can use it.
It would theoretically be possible to use the fluent syntax directly, without the arrow language, but this would've made it harder to figure out how the program should be structured and increased the development time.
There are, however, drawbacks to including the arrow language; error handling is somewhat lacking. A common bug in the translator is that it sometimes emits invalid arrow language constructs, causing absurd error messages when the arrow language interpreter tries to parse them; the goal is, of course, to fix as many bugs as possible before users start to use it, but fixing them all is hardly realistic.

5.2 Future Work

One improvement would be lifting the known limitations (chapter 1.4). None of them are fundamental to the task of validating Volvo Cars' log data, they're simply mistakes in the design process.
The system is fast enough to satisfy Volvo Cars' current requirements, but increased data amounts may require changes. The performance could be improved by translating some of the Python code to C++. Another way to speed it up would be parallelism; the current system can split work across multiple processes, but they must currently be on the same machine. Lifting that restriction would be welcome.

5.3 Conclusion

The result of this thesis is a system to assist the user in analyzing log files from test runs of cars or car components. This analysis has, to this point, only been done by hand. The system allows the user to provide rules that should be satisfied, and then provide a log file, where the system automatically tests if the rules are upheld.
The system consists of several modules (see chapter 3) where one module is the most interesting, namely the arrow language.

The arrow language is a domain-specific language created specifically to describe the relevant analysis rules. The goal was to keep it simple while being able to implement all desired rules. Initially, we checked if anything could be built on top of existing methods such as finite automata and temporal logic, but it turned out that the interval-based requirements make that infeasible within the scope of a thesis.
Therefore, we chose to implement a rule-oriented domain-specific language where time intervals are a core feature, and everything else is kept as simple as possible.

Bibliography

[1] Bührer, Michael and Sparrer, Bernd (2017). MDF4-Lib. http://www.turbolab.de/mdf_libf.htm

[2] ASAM - Association for Standardization of Automation and Measurement Systems (2014). MDF - Measurement Data Format. https://www.asam.net/standards/detail/mdf/

[3] Bartocci, Ezio; Falcone, Yliés (2018). Lectures on Runtime Verification. Springer. ISBN 978-3-319-75632-5.

[4] Hopcroft, John E.; Motwani, Rajeev; Ullman, Jeffrey D. (2007). Introduction to Automata Theory, Languages, and Computation (3 ed.). Addison Wesley.

[5] Rajeev Alur, David L. Dill. 1994. A Theory of Timed Automata. In Theoretical Computer Science, vol. 126, 183-235

[6] R. Alan Eustace, Amar Mukhopadhyay. 1982. Deterministic Finite Automaton Approach to Design Rule Checking for VLSI. 19th Design Automation Conference. p.712-717.

[7] Sallam Osman Fageeri, Rohiza Ahmad. 2014. An Efficient Log File Analysis Algorithm Using Binary-based Data Structure. In 2nd International Conference on Innovation, Management and Technology Research. Vol 129, p 518-526.

[8] Jakub Breier, Jana Branišová. 2015. Anomaly Detection from Log Files Using Data Mining Techniques. In Lecture Notes in Electrical Engineering 339, Information Science and Applications, Springer-Verlag Berlin Heidelberg, p 449-457.

[9] Ahmed Umar Memon. 2008. Log File Categorization and Anomaly Analysis Using Grammer Inference. Master's thesis Queen's University, Ontario, Canada.

[10] Peter Øhrstrøm; Per F. V. Hasle (1995). Temporal logic: from ancient ideas to artificial intelligence. Springer. ISBN 978-0-7923-3586-3.

[11] Colombo, C., Pace, G.J, Schneider, G: LARVA - safer monitoring of real-time java programs (tool paper). In: Proceedings of SEFM 2009: The Seventh IEEE International Conference on Software Engineering and Formal Methods, pp. 33-37. IEEE Computer Society (2009)

[12] http://runtime-verification.org/

[13] rulebased runtime verification revisited

[14] https://automotive.softing.com/en/standards/bus-systems.html