RUNTIME VERIFICATION OF A PLC SYSTEM: AN INDUSTRIAL CASE STUDY

Mohammad Heidari
mhi15001@student.mdh.se
saidhdr@gmail.com

Examiner: Daniel Sundmark
Mälardalen University, Västerås, Sweden

Supervisor: Adnan Causevic
Mälardalen University, Västerås, Sweden

May 17, 2017
Abstract

The term verification can mean different things to different people based on their background and the context of research. Comparably the term runtime might bring about similar disarray with lesser confusion. Whilst a software application executes on a system, there are libraries and codes that facilitate the execution of that binary, these processes and libraries make up the runtime environment, and verification in this context refers to the concrete techniques that verify a run and checks if it satisfies or defies the specified requirements. This study is specifically concerned with runtime verification systems, and how to make use of them in a limited-resource environment such as embedded systems or Programmable Logic Controller (PLC). It is of utmost importance to validate the conformance of a system, specially those in automotive and transportation industry. Vehicles such as trains are responsible to transport millions of people everyday which makes safety and reliability of the system the most prime component of bunch. Many manufacturers employ state of the art technologies and processes to endure the safety of the final product. Since most of the parts in vehicles (including trains) are computer controlled, the system (OS, application, control system, etc.) should undergo proper and comprehensive tests to single out even the smallest anomalies and misbehaviour. While traditional software testing can detect most of the anomalies, runtime verification can be used as a complementary method to passively monitor the running system and point out those behaviours and malfunctioning that are otherwise close to impossible to be caught. This thesis investigates runtime verification in train manufacturing, and propose a structured, contemporary process and system to monitor the train control system for conformance. Furthermore several methods of formal specification to be used for declaration of specification were explored and a suitable choice for this case study was suggested, in addition a proof of concept tool to demonstrate the entire process is developed as well.

Keywords — software testing; runtime; runtime verification; experiment; software specification; formal specification; PLC
Acknowledgments

First and foremost I would like to thank my thesis advisor Adnan Causevic (IDT, Mälardalen University) without whom I could not complete this task. He was kind enough to let me use his resources (office/laptop) to do my thesis. During this five months of collaboration with him, not only he helped me with technical issues and difficulties but also his warm and friendly attitude gave me extra inspiration for doing my best. I would also like to thank Ola Sellin (Bombardier Transportation) for providing the necessary resources and support to conduct my research and development at Bombardier site and special salute to Joakim Svartengren and Gustav Dafgård (Bombardier Transportation) for their patience and nice explanation of Bombardier tooling. In like manner I would like to thank Prof. Daniel Sundmark (IDT, Mälardalen University) for his insights during this thesis and other experts and individuals who were involved during this project and helped me in any possible way they could. Last but not least thanks to my family for all their moral support.

"I would spread the cloths under your feet:
But I, being poor, have only my dreams;
I have spread my dreams under your feet;
Tread softly because you tread on my dreams."

W. B. Yeats
# Table of Contents

Abstract I

Acknowledgments II

List of Figures V

List of Tables VI

List of Code Snippets VII

Abbreviations IX

1 Introduction 1
  1.1 Problem formulation ........................................ 1
  1.2 Context ...................................................... 2
  1.3 Expected outcome .......................................... 2
  1.4 Preliminaries ............................................... 2
  1.5 Report outline ............................................. 2

2 Background 3
  2.1 PLC and its software layout .............................. 3
    2.1.1 PLC programming languages .......................... 3
  2.2 Unifying terms for Runtime and RV ........................ 4
  2.3 Verification and its phases ................................ 4
    2.3.1 Instrumentation ....................................... 5
    2.3.2 Monitor and monitor creation .......................... 7
  2.4 Usability of a runtime verification systems ................ 8
  2.5 Formal specification ...................................... 8
    2.5.1 Gherking ................................................. 9
    2.5.2 Temporal logic & PLCspecif .............................. 9
  2.6 Data exchange protocol & COM ................................ 11

3 Related Work 12
  3.1 Runtime verification ....................................... 12
  3.2 Formal specification ...................................... 14

4 Method 16
  4.1 Problem solving process ................................... 16
  4.2 Motivation domain requirement ............................ 16
  4.3 Design ...................................................... 16
  4.4 Targeted System for monitoring .......................... 17
    4.4.1 C30 Train ............................................. 17
    4.4.2 TCMS .................................................. 17
    4.4.3 SoftTCMS ............................................. 18
    4.4.4 DCUTerm .............................................. 18
    4.4.5 Current process ...................................... 18
    4.4.6 New process .......................................... 18
  4.5 Interface between monitoring and targeted system .......... 19
  4.6 Specification of targeted system .......................... 20
  4.7 MVTerm ..................................................... 20
    4.7.1 Hardware for monitoring system ...................... 20
    4.7.2 Software for monitoring system ........................ 20
  4.8 Effects of RV in PLC development ........................ 21
5 Technical Approach

5.1 Converting requirements to formal specification ........................................... 23
5.2 Converting formal specification to database .................................................... 25
5.3 MVTerm implementation .................................................................................... 26
  5.3.1 Architecture .................................................................................................. 27
  5.3.2 DCUTerm script and setup ............................................................................. 27
  5.3.3 Communication between DCUTerm and MVTerm ........................................ 28
  5.3.4 Stimulating the target system (simulated CCU) ............................................... 29
  5.3.5 Producing verdict and visual indication ......................................................... 32
  5.3.6 Final tool and demo ...................................................................................... 33
  5.3.7 Tracking time ............................................................................................... 33

6 Result .................................................................................................................... 34

6.1 Results of applying stimuli .................................................................................. 34
6.2 Analysis of results ............................................................................................... 35

7 Delimitation .......................................................................................................... 39

7.1 Discussion ............................................................................................................ 39
7.2 Limitation ............................................................................................................ 39
  7.2.1 Construct validity .......................................................................................... 39
  7.2.2 Internal validity ............................................................................................ 40
  7.2.3 External validity ........................................................................................... 40
  7.2.4 Reliability ..................................................................................................... 40

8 Conclusion ............................................................................................................. 41

8.1 Summary ............................................................................................................. 41
8.2 Future work ......................................................................................................... 41

References .............................................................................................................. 42

Appendix .................................................................................................................. 1

A MVTerm Interaction ................................................................................................ 1
  A UI: MVTerm ......................................................................................................... 1
  B UI: Setting ........................................................................................................... 2
  C UI: Plotting and warning ..................................................................................... 3

B DCUTerm Interaction ............................................................................................. 4
  A Signal analysis ..................................................................................................... 4
  B Axle manipulation script ..................................................................................... 5
  C IoTool automation script ..................................................................................... 6
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>PLC in action</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>RV Process</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>Instrumentation</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>Possible verdict domains</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>PLCSpecif's module and its compositions</td>
<td>10</td>
</tr>
<tr>
<td>6</td>
<td>PLC's shared memory and verification library</td>
<td>13</td>
</tr>
<tr>
<td>7</td>
<td>Problem solving's process</td>
<td>16</td>
</tr>
<tr>
<td>8</td>
<td>Overall design of solution's process</td>
<td>17</td>
</tr>
<tr>
<td>9</td>
<td>TCMS Box (ridiculously simplified!)</td>
<td>18</td>
</tr>
<tr>
<td>10</td>
<td>Bigger picture of overall current process</td>
<td>19</td>
</tr>
<tr>
<td>11</td>
<td>Bigger picture of new process</td>
<td>19</td>
</tr>
<tr>
<td>12</td>
<td>Task A: 10 executions histogram (Normal)</td>
<td>21</td>
</tr>
<tr>
<td>13</td>
<td>Task A: 10 executions histogram (Suspicious)</td>
<td>22</td>
</tr>
<tr>
<td>14</td>
<td>PLCSpecif's input-output-core-logic connection</td>
<td>24</td>
</tr>
<tr>
<td>15</td>
<td>PLCSpecif's module specification of train speed</td>
<td>25</td>
</tr>
<tr>
<td>16</td>
<td>MVTerm's containers</td>
<td>27</td>
</tr>
<tr>
<td>17</td>
<td>DCUTerm GUI</td>
<td>28</td>
</tr>
<tr>
<td>18</td>
<td>MDCT signal manipulation</td>
<td>31</td>
</tr>
<tr>
<td>19</td>
<td>Stimulation process</td>
<td>32</td>
</tr>
<tr>
<td>20</td>
<td>DCUTerm axles' signal values</td>
<td>34</td>
</tr>
<tr>
<td>21</td>
<td>DCUTerm axles' signal zones</td>
<td>35</td>
</tr>
<tr>
<td>22</td>
<td>Zone 1 signals zoomed</td>
<td>36</td>
</tr>
<tr>
<td>23</td>
<td>Zone 1 signals split-ted and zoomed</td>
<td>36</td>
</tr>
<tr>
<td>24</td>
<td>Zone 2 signals zoomed</td>
<td>37</td>
</tr>
<tr>
<td>25</td>
<td>Zone 2 signals split-ted and zoomed</td>
<td>37</td>
</tr>
<tr>
<td>26</td>
<td>Zone 3 signals zoomed</td>
<td>38</td>
</tr>
<tr>
<td>27</td>
<td>Zone 3 signals split-ted and zoomed</td>
<td>38</td>
</tr>
<tr>
<td>28</td>
<td>MVTerm main UI</td>
<td>1</td>
</tr>
<tr>
<td>29</td>
<td>MVTerm main UI</td>
<td>2</td>
</tr>
<tr>
<td>30</td>
<td>MVTerm Plotting and evaluation area</td>
<td>3</td>
</tr>
<tr>
<td>31</td>
<td>DCUTerm signals' plot for axles</td>
<td>4</td>
</tr>
</tbody>
</table>
List of Tables

1  AOP’s concepts ......................................................... 6
2  Software requirement 1 to be monitored. .......................... 23
3  Software requirement 2 to be monitored. .......................... 23
4  DCUTerm most used APIs list ....................................... 29
5  Signals to manipulate .................................................. 30
List of Code Snippets

<table>
<thead>
<tr>
<th></th>
<th>Code Snippet</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Unsequenced side effects [1]</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>Optimization side effects [1]</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>An aspect’s declaration [2]</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>XML declaration of specification</td>
<td>26</td>
</tr>
<tr>
<td>5</td>
<td>Connecting DCUTerm and MVTerm</td>
<td>29</td>
</tr>
<tr>
<td>6</td>
<td>IoTool script (XML declaration)</td>
<td>32</td>
</tr>
</tbody>
</table>
### Abbreviations

**AOP** Aspect Oriented Programming  
**BIOS** Basic Input/Output System  
**CERN** European Organization for Nuclear Research  
**CLR** Common Language Runtime  
**COM** Component Object Model  
**CPU** Central Processing Unit  
**CTL** Computational Tree Logic  
**DDE** Dynamic Data Exchange  
**DLL** Dynamic Link Library  
**DSL** Domain Specific Language  
**DSP** Digital Signal Processor  
**EtherCAT** Ethernet for Control Automation Technology  
**FBD** Functional Block Diagram  
**GUI** Graphical User Interface  
**HIL** Hardware-in-the-Loop  
**IDE** Integrated Development Environment  
**IEC** International Electrotechnical Commission  
**IL** Instruction List  
**IP** Internet Protocol  
**ITL** Interval Temporal Logic  
**LTL** Linear Temporal Logic  
**MDCT** Mitrac Desktop Component Tester  
**MDE** Model-driven Engineering  
**MPI** Message Passing Interface  
**ODK** Open Development Kit  
**OS** Operating System  
**PC** Personal Computer  
**PLC** Programmable logic controllers  
**PoC** Proof of Concept  
**PSL** Property Specification Language  
**RAM** Random Access Memory  
**RDBMS** Relational Database Management System  
**RTOS** Real Time Operating System
RV  Runtime Verification
SFC  Sequential Function Chart
SIL  Software-in-the-Loop
SL  Stockholms Lokaltrafik
SRS  Software Requirements Specification
ST  Structured Text
STML  Simulation Trace Mapping Language
SVA  SystemVerilog Assertions
TCMS  Trian Control and Management System
TCP  Transmission Control Protocol
TPSL  Trace Property Specification Language
USB  Universal Serial Bus
WPF  Windows Presentation Foundation
XML  Extensible Markup Language
1 Introduction

Automation process is one of the most important aspects of modern industrial era and Programmable Logic Controllers (PLC) are the building blocks whereupon industrial automation was molded. Furthermore PLC is the prime component in automation and is used heavily in sensitive industries such as power plants, water distribution systems, car manufacturing and even defense systems which yields unparalleled attention to its reliability and integrity. PLC and its execution environment are designed to be simple computing machines whose programs can be easily verified and validated by the hardware they are controlling [3]. Although this simplicity can speed up development process, it hinders real-time/dynamic verification of developed code due to strict timing and limited resources of PLC systems. Verification techniques can be categorized into static and dynamic analysis. Usually, static verification is common to be utilized for scrutinizing such systems with limited resources, however, these techniques suffer from shortcomings. For instance, model checking can suffer from state-space-explosion (as the size of the system increases, the computational power needed to verify the system increases) or theorem proving requires great amount of manual labour to carry out the proof. Additionally static analysis verifies the system behaviour prior to its execution and does not provide any information on runtime behaviour [2]. On the other hand, dynamic verification provide the opportunity to investigate and verify if a run of the system under scrutiny satisfies a predefined correctness property [2]. It can be a complementary method to the traditional static verification technique. An integral part of dynamic verification is runtime monitoring which is the key factor to collate defects and deviation in the executing system.

At present there are few approaches for PLC runtime verification, for instance simulation-based verification which promises a cost effective solution. S. Makris et al. state that Hardware-in-the-Loop (HIL) and Software-in-the-Loop (SIL) are the main approaches for doing a simulation-based verification [4]; SIL being a static method simulates both the PLC program and the plant. On the contrary HIL or (soft-commissioning) is a real-time simulation where a real PLC program communicates with a simulated plant. The latter approach produces more realistic results and can verify more complex control scenarios. Fast and reliable communication between hardware and verification system plays an integral part of monitoring system. Currently Ethernet for Control Automation Technology (EtherCAT) which is a real-time industrial Ethernet, is popular in the development of monitor systems in industrial automation, as well as static/offline modeling, verification by code instrumentation and etc., all of which have their own pros and cons. Runtime verification can be used for many purposes such as safety policy checking, debugging, testing, security, behaviour modification, fault protection, profiling and etc. The main goal of this thesis is to provide a thorough research on a dynamic runtime verification system which focuses on testing and validating the PLCs code for its semantic and functional correctness. A Runtime Verification (RV) system accepts two inputs: (1) A system to be checked, and (2) a set of predefined properties of the system to be checked during its execution time.

1.1 Problem formulation

Currently, runtime verification is a state-of-the-art technique which is usually used in large-scale systems equipped with powerful processing units which has elongated and costly implementation. On the contrary, most small and medium sized companies and manufacturers are using cost effective PLCs with limited processing power which restrict them to use traditional static verification, and causes the adaptation of runtime verification to be very challenging. The difficulties of bringing model checking techniques to automation industry are twofold. First, both a formal model of the system to be verified and a formal specification of verification criteria need to be created. They are usually the result of a cooperation between a control engineer and a formal method experts, with a potential of misunderstanding. A second concern is the amount of necessary computation power resources [5]. Furthermore limited resources of PLC environment and time restrictions calls for an accompanying hypervisory verification tool. The aim of this thesis is to propose a suitable static verification method that can be easily converted to dynamic method for checking the functional correctness, semantic and timely execution of PLC codes/cycles, and address the following questions:

- How to convert PLC’s properties into formal specification for RV systems?
How to instrument the PLC’s environment to generate relevant events for monitoring and verification?

What are the effects of introducing RV in PLC software development process?

1.2 Context

The main idea of the thesis surfaced while communicating with a company named Bombardier Transportation. Bombardier is an aerospace and transportation company with a site in Västerås city. Bombardier uses PLC devices heavily in their trains. One of the main concerns is to verify the PLC’s performance to minimize all the deviation and unforeseen issues while the system is running. Thereupon dynamic verification of PLC is introduced to tackle the issue. Bombardier Transportation were kind enough to allow data collection, testing and development take place at their site which resulted in developing/proposing a solution which is customized for Bombardier development environment, although the main concept can be extended to any development system.

1.3 Expected outcome

This thesis will contribute to the research area of static/dynamic verification of PLC environment which leads to a verification method and mechanism that can validate functional correctness, semantic and timely execution of PLC codes/cycles. A tangle outcome consists of:

- State of the art research of online runtime verification approaches in automation industry (PLC-enabled) and an analysis of different approaches.
- Analysis and description of proposed method to be used in the verification environment.
- A proof of concept tool/system for runtime verification of PLC systems.

1.4 Preliminaries

It is worth noting that, these words are used interchangeably throughout this report: (Dynamic/Online), (Static/Offline), (PLC Code/Cycles), (Bombardier Transportation/BT), (Microcontroller unit/MCU). For the sake of not repeating the word ”proposed solution” throughout the thesis the name: MVTerm is chosen and will be used instead or interchangeably.

1.5 Report outline

This section outlines the structure of the thesis, and provides a brief description of what each chapter includes.

Chapter 2 — Present background, and preliminaries on runtime verification and establishes the necessary foundation for the thesis.

Chapter 3 — It explores state of the art related to the thesis topic and provides a wide to narrow approach for introducing and comparing of related papers.

Chapter 4 — This chapter introduces the idea (method) and explains it scientifically. The formal method of finding solution and the process of achieving it, is described thoroughly.

Chapter 5 — It focuses on the technical approach and the steps taken for implementation.

Chapter 6 — Outcome and final results of the thesis will be consolidated in this chapter. An analysis of the results will be provided as well.

Chapter 7 — A discussion about the overall process of the thesis, from problem formulation to the findings and all the challenges involved in between.

Chapter 8 — This is the final summary which points out the research question and highlights the breakthrough of the research, furthermore some suggestions as future work.

2 Background

This chapter spreads out the foundation and basic terminologies in the RV context, and renders an illumination on better understanding the thesis. First and foremost a general description of PLC, runtime and RV will be made current so there will be a unified agreement throughout the report when talking about runtime or RV. It is then followed by an explanation of verification phases used in common practises. Furthermore, different classifications of RV will be introduced as well as a discussion on the usability of RV systems.

2.1 PLC and its software layout

Traditional control panels consist of components such as hardwired relays and timer logic which made them extremely effortful to implement and control. PLC was introduced to overcome these issues and provide an easier and more flexible control using programming and logic instructions. A PLC consists of four main parts; (1.) input/output units, (2.) Central Processing Unit (CPU), (3.) memory and (4.) programming terminal [6]. Instructions will be saved into PLC’s memory via programming terminal, CPU will continuously run and scan input signals and executes the instruction logic on them which produces output signals to be fed to the connected machines. PLC devices are usually equipped with standard interfaces so they can communicate with input/output units (sensors, actuators, etc) without the need for intermediate circuitry.

PLC’s software layout depends on the manufacture but it usually has three layers; (1.) Control program (Instruction logic), (2.) firmware and in some cases (3.) Basic Input/Output System (BIOS). Control program is the instruction logic (ladder logic, structural text, etc) that has been uploaded to the memory. Firmware is the manufacturer code that executes when PLC turns on and is responsible for loading other parts such as BIOS and operating system (if any). BIOS does the preliminary diagnosis of input and output units attached to the system [7]. Figure 1 depicts a PLC’s software layout and its connections, the code insides the PLC will be running in a loop and checks the input (sensor in this case), upon any changes in the sensor and depending on the control logic in the PLC memory, control program will be executed and actuation takes place (opening valve in this case).

2.1.1 PLC programming languages

Like any other platform that provides an array of different tools and programming languages to be utilized for development, PLC offers the same flexibility. International Electrotechnical Commission (IEC) has identified the five emerging programming languages used in process and discrete programmable controllers and introduced IEC 61131 ² standard (IEC 61131-3 specific to programming languages) to include and specify these languages: Ladder Diagram (LD), Instruction List (IL), Function Block Diagram (FBD), Structured Text (ST) and Sequential Function Chart (SFC) [8]. These will be the languages that are being used in this case study hence IEC 61131 standards is applied as well. Full explanation of the languages and the standards is out of scope for this thesis.

2.2 Unifying terms for Runtime and RV

In computer programming, runtime refers to a special library that is used by a compiler to implement functions built into a particular programming language. This library is designed to support the execution of that language. Often it houses the implementation of basic low-level commands and provides, type checking, debugging, code generation, garbage collection and so forth [9]. This is the runtime definition throughout the report. Most runtime systems will produce the same behaviour during execution if the code is compiled by a compiler with the same optimization and specification. Usually, there is more than one compiler for most of the popular programming languages such as C/C++, Java, C# and etc. Each of which enforces its own optimization on the code that results in a slightly different behaviour in the runtime, for instance if this code snippet 1 is compiled by clang\(^3\)

```
int main() {
    int x = 0;
    return (x = 1) + (x = 2);
}
```

Code Snippet 1: Unsequenced side effects [1]

the output will be 3 with a warning for undefined behaviour in the trace log. Interestingly if the same code 1 is compiled by gcc\(^4\) the output will be 4 and that is because of the optimization that gcc does behind the scene and changes the code to 2:

```
int x = 0;
x = 1;
x = 2;
return x + x;
```

Code Snippet 2: Optimization side effects [1]

Although these kind of undefined behaviours can be prevented by practising good programming but a runtime verification system can catch all these behaviours and produce a more deterministic result as well as providing a systematic check during execution.

Runtime verification is the discipline of Computer Science that deals with the study, development and application of those verification techniques that allow checking whether a run of a system under scrutiny satisfies or violates a given correctness property [10]. In RV, a run is typically represented by some log or trace. That trace can either represent a sequence of program states, or a series of events representing the program behavior such as I/O, system calls, etc. Figure 2 shows a general process of a verification system.

- Monitor: It is usually created from properties e.g. property A and property N.
- Instrumentation: The bridge between monitor and system to generate events.
- Response: Outcome of monitor that can be categorized as:
  - Verdict: It is a status given to each event generated by system to identify its success or failure.
  - Feedback: The feedback that monitor sends back to the system for further processing such as corrective action.

2.3 Verification and its phases

A run of a system can be translated into a finite sequence of events describing each action taken by the system or state selected by the system. In general events in verification systems consist of a name and attached data values or dataset which depend on the context of the running system. These are the definitions which are used throughout the report:

\(^{3}\)is a compiler front end for the programming languages C, C++. available at: https://clang.llvm.org

\(^{4}\)is a free compiler for GNU system (C/C++). available at: https://gcc.gnu.org
Event — is a pair of \((e, \bar{v})\) where \(e\) is the name of the event and \(\bar{v}\) is the value(s) in the event. And the short notation will be \(e(\bar{v})\).

Trace — is a finite sequence of events, an empty trace will be represented by \(\epsilon\) and \(\text{Trace}(N)\) will refer to all the traces over all sets of event \(N\).

Property — in general is a synthetic object that belongs to a particular logic or language. Due to different formalization or logic in different verification systems, each system can express properties in a different manner. For easier denotation (semantic) properties can be modeled as functions of traces to a given verdict domain. Let \(\text{Prop}(N, \mathbb{D}) = \text{Trace}(N) \rightarrow \mathbb{D}\) which refers to all the properties from traces over the event \(N\) to the verdict domain \(\mathbb{D}\).

One advantage of verifying a system during runtime is that the system can act upon and take corrective course. To achieve this, the system under scrutiny communicates to monitors via feedback and verdicts. A running system produces an infinite number of events (traces) and monitoring such system requires powerful units, in order to contain the hectic number of traces and logs, it is usually common to consider finite number of traces in a running system, and by analyzing finite sequence snapshot of traces a continuous and deterministic verification can be achieved [2].

2.3.1 Instrumentation

The process of extracting events and states from a running system and feeding them to the monitor (Instrumentation) for analysis can be of crucial importance and if done improperly can generate large overhead for the runtime. This can be overlooked if the analysis takes place on log files and traces (offline). Currently there are different techniques for instrumentation ranging from manual instrumentation, to customized libraries, to augmenting virtual machine, to Aspect Oriented Programming (AOP) and to building special hardware support. Depending on the system under scrutiny one technique or combination of few might provide a better verification platform. Building special hardware for instrumentation can provide its own specialized environment for processing hence less overhead for system’s runtime, which makes it a prime contender in the embedded runtime verification since the processing power is scarce. However, the caveat is the cost of hardware and implementation. One recent popular choice for code instrumentation is AOP that promotes modularization of crosscutting concerns. Figure 3 depicts an instrumentation system interfaced to a running system, depending on the situation the interface can be purely software (generated instrumented classes by an instrumentation engine) or a mixture of hardware and software connected to the system such as EtherCAT.

Aspect oriented programming — In software systems there are sections of code such as logging, tracing, security management, policy enforcement and etc., that need to be scattered all over the code which results in a disseminated code that does not follow modularization. This kind of code is called Cross-Cutting which raises concerns for scalability, maintainability and testability of the entire system [11]. AOP aims at solving these issues by defining a new module named Aspect that houses all the cross-cutting codes hence the AOP naming. A system which is decorated by aspects consist of both the main application that contains the basic functionalities of the system and sets
of aspects that address the cross-cutting concerns and functions as the instrumenting system. The instrumented system then starts executing and its targeted/interested events will be observed by aspects, then these observations will update the monitor state which finally can produce a verdict and if implemented sends feedback to the instrumented system. The main concepts of AOP are detailed in table 1.

<table>
<thead>
<tr>
<th>Concept</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Join points</td>
<td>It is a well-identified point during runtime/execution for instance: a method call, object creation, an access to a property or attribute of an object and so forth.</td>
</tr>
<tr>
<td>Pointcuts</td>
<td>It is like a method signature (or a single line interface) in normal programming languages, and it is used to access the context of selected join points.</td>
</tr>
<tr>
<td>Advice</td>
<td>An advice is a piece of code like a method or function that contains a pointcut and some codes (body of the method), upon matching of pointcut and join point the code will be executed.</td>
</tr>
<tr>
<td>Aspects</td>
<td>An aspect is a module that contains a collection of pointcuts, advices and class declarations</td>
</tr>
</tbody>
</table>

Table 1: AOP’s concepts

There are three kinds of advice statement which are based on the execution order of the advice’s code when a joint point that is declared in a pointcut is matched. The orders are as follows: Before — The advice will be executed before the pointcut is called. After — The advice will be executed after the pointcut is called. Around — The advice will be executed instead of the pointcut. Although there is a Proceed statement that can be used to execute pointcut even though it is in the around execution mode. Since AspectJ\(^5\) is the most widely known AOP, the following example 3 [2] will be in AspectJ which explains the concepts introduced in table 1.

```java
public aspect IncreasingNumbers {
    int current = 0;

    pointcut com(String name, int number, Task task):
        call(void Task.sendCommand(String, int))
    && args(name, number) && target(task);

    before(String name, int number, Task task) : com(name, number, task) {
        assert(number == current + 1):
        "wrong job number for " + name + " (" + task.id + ")";
    }
}
```

Code Snippet 3: An aspect’s declaration [2]

This simple aspect checks if the command job has increased by one. Line 1 is the aspect declaration, line 2 introduces an integer variable \( \text{current} = 0 \). Line 4 declares a pointcut named \( \text{com} \) with three arguments \((\text{String name}, \text{int number}, \text{Task task})\) that get activated when a call to a join point (line 5) with similar signature of \( \text{void Task.sendCommand(String, int)} \) is matched. Line 6 is a directive \( \&\& \text{args(name, number)} \) that bounds the parameters’ values to the \((\text{name, number})\) and makes them accessible during runtime, \( \&\& \text{target(task)} \) binds the object on which the selected methods are called to \text{task}. Line 7 is the advice body with three arguments \( \text{before(String name, int number, Task task)} \) and \text{before} keywords indicate it will be executed before any join point. Pointcut parameters are extracted by this statement \( : \text{com(name, number, task)} \) and are passed to the advice’s body. Line 8 will assert using a simple comparison and will prompt the message constructed in line 9.

Interfacing — is another integral part of instrumentation, this is the section that connects the instrumentation engine to the executing system. Depending on the technology used this can vary from purely software based to a combination of hardware and software. Software based instrumentation provides an engine (compiler-like-system) that wraps around the targeted system (like AOP) and captures all the interactions between system calls and passes them through to the monitor system. Other scenarios that special hardware is required, targeted system will be tweaked to communicate with instrumentation engine via technologies such as EtherCAT and instrumentation will happened alongside the running system.

2.3.2 Monitor and monitor creation

Designing a monitor for a system depends on these key factors; where it (monitor) executes, for instance on external hardware or on the system; what it monitors, for example instructions’ correctness, timeliness, memory values, etc.; and how inputs are passed to the monitor, via instrumentation, direct sensors, etc. Due to time restriction and limited resources in realtime systems such as PLC, monitors are usually running on another device which is connected via some interface to the instrumented target system [12]. The main ingredient of a monitor are the specifications. Specifications are the extracted and formalized atomic properties from the targeted system (e.g. capacity limit of a tank in an automated plant).

Formal monitoring — Atomic propositions which represent system’s properties after observation, are the base of specification logic in monitors. Let \( AP \) be a set of atomic proposition, in order to map it to a set of truth values for specification this notation can be used: \( AP \rightarrow \{ \top, \bot \} \) which means \( AP \) can have either \( \top \) or \( \bot \) as its state [12]. A good specification should be minimal, adequate, unambiguous, complete, consistent, and satisfied. It should also considers Constructability, Usability, Communicability, and manageability [13].

Properties to formal specification — Implementation of formal specification is based on different factors such as what type of system is going to be modeled by it, at what point of software development cycle they are being introduced, and how they are going to be implemented and applied [14]. During formal verification of system, properties to be checked are often described in: temporal logics, such as Linear Temporal Logic (LTL), Property Specification Language (PSL), SystemVerilog Assertions (SVA), or Computational Tree Logic (CTL) [15]. Then these properties will be verified against mathematical objects such as: finite state machines, vector addition systems, timed automata, hybrid automata, labelled transition systems, process algebra, and Petri nets [16]. The explanation for formal verification concepts are out of the scope of this thesis and are only mentioned to provide clarification for further topics. However, a comparison of two formal specification languages (temporal logic & PLCSpecif) will take place in the Formal specification (section 2.5.2) to explain which one suits more for this particular work. This thesis will focus on a dynamic verification system that directly monitors targeted system so it does not require a mathematical object of the system properties to be modeled.
Monitor implementation — Depending on technology used different paths can be taken to create monitor for a system. Although using an AOP for instrumentation and monitoring can assist developers for creating a monitor using provided tools, it is still the developer’s duty to choose the best approach. For instance, according to [2] using AspectJ there are three possible ways to create a model: (1.) defining each model as a self-contained aspect that contains pointcut definitions, (2.) Separate aspects and properties by letting each property to have different method to be invoked, and then be caught by instrumentation which executes the aspect, (3.) Define an abstract aspect (like a parent class) that declares only the base properties of the system then do extension on the class like inheritance in traditional programming.

Responses — Monitors respond to events generated by targeted system via verdicts and feedback (if necessary). Once monitor receives an event it will execute an analysis and produce a verdict and feedback. The common judgment for verdict domain is only true or false value whereas many runtime verification systems use more that these binary values to provide a better understanding of the state of the process. For instance an event/state can be going towards success or failure but its current status received by monitor does not convey it totally by true or false, so it can be still true or still false and the final result might change [17][2]. Figure 4 shows different states of an event that show it can be true at the end even though it started as a false state.

![Figure 4: Possible verdict domains](image)

2.4 Usability of a runtime verification systems

According to [2] there are three main characteristics that each runtime verification system should follow in order to make them usable in every day development life-cycle:

Efficiency — Since RV systems can be used in any scale, it should be able to function with different trace sizes from small to large and adopt accordingly with minimum impact and overhead on the target system.

Expressiveness — Most specifications provided with different languages are not very straightforward, so a good tool should be expressive of what it models or analyzes to be comfortable to worked with.

Elegance — Writing specification is a cumbersome task, a system that provides ease whilst writing specification and is concise has elegance.

2.5 Formal specification

Mathematically based techniques that have formal vocabulary, syntax and formally defined semantics that can be used to infer useful information from a system/software are called formal specification. In Computer Science, formal specification is to help with the implementation of a system by describing the system and analyzing its behaviour. Furthermore it highlights the key properties of interested domain via reasoning [18]. Formal specification outlines "what" a system should do rather than "how" it should do it, and it is not an implementation by any means [19]. It can be used to demonstrate if a system’s design is correct with respect to its specification or not, and can detect incorrect designs in early stage. Even though formal methods are not very popular in software development due to not being very cost effective and sometimes go against dynamic and agile nature of software development [20], their role cannot be undermined. After all

---

8
a design or an implementation can only be correct with respect to a specification and not on its own. According to [13] these specification paradigms can be used for modeling properties based on what kind of system they are trying to model or at what stage it is introduced:

- **History-based specification**
  - behavior based on system histories
  - assertions are interpreted over time

- **State-based Specification**
  - behavior based on system states
  - series of sequential steps, (e.g. a financial transaction)

- **Transition-based specification**
  - behavior based on transitions from state-to-state of the system
  - best used with a reactive system

- **Functional specification**
  - specify a system as a structure of mathematical functions

- **Operational Specification**

2.5.1 Gherking

Gherking is a language parser that is used in the heart of Cucumber\(^6\) (framework for software testing) and describes software behaviours in a logical language understandable for none-technical users. It belongs to a wider group of Domain Specific Languages (DSL), which are capable of describing software specification in an unambiguous way, and many companies use DSL to conduct day to day testing [21]. There are Gherkin-type syntax already used in industry to specify software specifications and behaviour, although it is not a formal specification language, but there is a possibility of forking a customized version as a method of formal specification which has a general purpose application. However it is dropped in favour of more suitable method (PLCspecif) which is specifically target PLC development.

2.5.2 Temporal logic & PLCspecif

Temporal logic (or tense-logic) refers to any system that consists of rules and symbolism for describing and reasoning about temporal information or any case qualified in terms of time [22]. An abstract way of thinking about time is to consider a line which is the simplest way of representing it. Then there are time points with some relation properties and measurement between them. This flow of time and description of what is happening is the core concept of temporal logic. Based on topology, place and spatial position there are varieties of temporal logics such as **Linear Temporal Logic (LTL)**, **Interval Temporal Logic (ITL)**, **Computational Tree Logic (CTL)**, **Property Specification Language (PSL)** and so forth. Temporal logic and state machine automata (another formal specification method) are popular specification languages to be used in ”mainstream IT”, whereas in PLC domain lack of software knowledge deprives developers from utilizing these tools and generally Model-driven Engineering (MDE) [23]. Dániel Darvas et al. introduce a formal specification method (PLCspecif) for PLC-Based application in [24] that is used in European Organization for Nuclear Research (CERN). Their method is based on four main principles: (1.) provide formalisms that are on par with need and knowledge of PLC community, (2.) decouple input-output handling hence cleaner core logic, (3.) proper support of events, and (4.) be more specific and less expressive which leads to less possible errors.

PLCspecif’s main building blocks are modules. A module can be either (a.) composite : contains sub modules and its behaviour described by them, or (b.) leaf module: that describes its

---

\(^6\)https://cucumber.io/docs
own behaviour directly. Then regardless of module type it will be further decomposed into three main parts based on their functionality: (1.) input definitions, (2.) output definitions and (3.) core logic. Like PLC’s cyclic execution, these parts are executed similarly. Figure 5 illustrates a breakdown on the module and how they are decomposed into smaller parts.

Figure 5: PLCSpecif’s module and its compositions

The leaf module descriptions of specification can be done with familiar concepts of state-machine, data-flow diagram and, PLC timers for PLC developers. Input definitions can consolidate functionalists such as Button1 pressed, Button2 pressed, ButtonX pressed to Button Pressed expression which can be used in different parts in a modular way and simplify the core logic. Event input, AND/OR tables and switch-case tables are also available in input/output expressions. Since modules can be either state-based, data-flow oriented or time dependent, core logic can increasingly become more complex. To address this issue, three different types of description can be used in the core logic:

State-machine — This module can be composed of hierarchical states or transitions. Furthermore a state can be either basic or composite (containing basics states at once). A transition can transform from any state to a basic state whereas the opposite is not possible to prevent unnecessary complexity. In order to control the flow of transitions, they are categorized into two type: (1.) event-triggered and (2.) none-event-triggered. These transitions can be enabled if the source state is enabled or in the case of event-triggered the connected event fires.

Input-output Connection Module — This type of module describes those part of logic that can be easily represented by integers and numbers (input/output values) and is inspired by Functional Block Diagram (FBD). It can graphically define how current input-output values should be connected based on the calculation of the previous cycles. The description contains pins for input/output representation and edges for connection between them.

PLC Timers — Although clock variables can be used in state machine to describe time-related behaviour but it is an error-prone method. To compensate for this and also utilizing a familiar concept for PLC developers, PLC timers can be used in this section like conventional timers in PLC (TP, TON, TOFF) and so forth.

The formal specification method proposed for this thesis is based on PLCSpecif and will be utilized in the Method section and will be briefly explored in the next chapter, however a full explanation of this method is out of scope for this thesis, and further information/instructions can be found in [24][23][25].
2.6 Data exchange protocol & COM

Many companies have developed in-house software to communicate with their specific hardware, for scalability reasons developed software follow some standards to enable data exchange between different software without re-implementing new hardware specific libraries. Dynamic Data Exchange (DDE) is a communication protocol to share information between applications in Microsoft Windows® platform. It is a set of messages and some guidelines that enable memory to be shared between applications and send information from one application to another upon incoming data. Another protocol for data exchange is ActiveX Automation Server. This method is based on Component Object Model (COM) which provides a framework for integrating reusable binary software components into an application. And furthermore it allows one application to be controlled by other Windows application. Since Windows® has a message-based architecture DDE and COM are both excellent protocols for sharing information between applications. And it is already used in realtime industry for such purposes.

---

3 Related Work

This chapter is divided into two sub parts and consists of the investigation of different works which are related to the topic of this thesis. Firstly the concept of RV in general computing will undergo an analysis, it will be narrowed down to the utilization of RV in embedded and realtime systems. In the second part few related works on formal specification method will be discussed and compared to the solution provided in this thesis.

3.1 Runtime verification

Nutt [26] mentioned in a tutorial on computer monitor system that the central questions to be answered whilst monitoring a system are "what to measure" and "why the measurement should be taken", which in turns output answers such as testing, debugging, verification and so forth. To further explores this area, different methodologies and techniques were used based on the target environment (from general purpose computers to embedded). The use of complementary hardware to run the monitoring system is not a new idea, and it is a practical alternative in the embedded systems scenarios due to all the shortcomings and limitations of targeted systems. Furthermore due to the fact that realtime systems are getting more complex and open which results in a more expensive development and verification process [27], RV seems to be able to provide a glance of correctness of the system without developing tools and testing methods to cover the entire system. This section will introduce related works that are using similar approaches such as separate execution environment or coupled environment (hypervisor) with PLC and online monitoring systems.

Model based verification and virtual prototyping is promoted by Daian Yue et al in [28]. The researchers have introduced a robust architecture for verification of systems using modeling of the system under scrutiny, then gathering traces of the execution and running it through the monitor system. The proposed architecture follows four steps; firstly, generating traces’ meta model to reflect the models. This step is the output of some constraint on the simulator that enables the system to generate arbitrary binary data and tagged them with meta data for further explanation and manipulation. Secondly transforming the model using Simulation Trace Mapping Language (STML) which is a domain specific language (DSL), this process maps the trace data to an abstract logical clock. The logical clock will map every ticks to the trace data to keep track of changes in the system, for instance using this clock it is possible to highlight the change of a sensor at a particular tick and provide a conventional timeline, Eclipse Modeling Framework were used to implement this part. Thirdly is the process of defining properties using an English-like DSL named Trace Property Specification Language (TPSL). Last but not least is to verify the defined properties against the mapped traces by TPSL's compiler. Each module will be compiled into a set of parallel automata which accords with properties and will be verified. The method proposed by Daian Yue et al unified a lot of good practise in RV, however it requires great amount of effort to be implemented and utilized in a system. The authors claim flexibility in traces generation and so forth, although the tooling and modeling method crave for a very steep learning curve and can hinder development of the system in early stages. The proposed solution in this thesis however is able to generate pure executable code and the specification language used is very close to the one which is used during development, this will shorten the development time and provide advantage for debugging, implementing and extending. When it comes to embedded world processing power and resources become prime concerns and most proposed solutions are trying not to violate the internal embedded policies of resource distribution and management. C. Watterson and D. Heffernan [29] have investigated few possible and suitable solutions for RV system in embedded world. It is highlighted by authors that running the monitor on a separate environment rather than target system is more preferable and reduces the overhead of monitor code execution on target system. Four main approaches; hardware, software, hybrid and on-chip monitors were discussed in [29]. All of each have its own pros and cons. In an attempt of a hybrid monitor Luis Garcia et al [3] investigate security and code integrity in PLC systems by introducing a more powerful embedded system to be coupled with PLC as a hypervisor environment to do the resource intensive calculation and analysis. The hypervisor carries a much more advanced
operating system that is capable of complex operation and analysis. In this case the hypervisor
is running on Windows embedded8. The solution proposed by Luis Garcia et al. is to designate
different memory locations in the PLC named temporary buffer and destination buffer. Writing to
the designated temporary buffer will be restricted and when PLC instructions are loaded into the
temporary buffer a verification library will be invoked to check if the value of temporary buffer is in
accordance with defined specification only then it will be passed to destination buffer for execution.
Researches have developed their solution (a verification library in a Dynamic Link Library (DLL))
by using SIMATIC WinAC Open Development Kit (ODK). All the checking and verification will
happen in the DLL. SIMATIC is then generate necessary programming blocks to be uploaded to
the PLC and can directly communicate with DLL via PLC’s shared memory. Figure 6 shows the
process of proposed verification by Luis Garcia et al. For instance if a new instruction changed
a value in the temporary buffer (turning on a valve etc), it invokes the associated WinAC ODK
function in the library. The function will then analyze the value and if passed the specification
safety will redirect it to destination buffer otherwise a warning will be sent out to the monitor.
Even though this is a promising approach towards RV in PLC systems, it can take longer time for
integration in bigger systems. The solution proposed in this thesis will run on Windows enabled
PCs and does scale up easily in enterprise applications. Furthermore this solution does not need
to meddle with the PLC code or shared memory, an approach which is frowned upon by bigger
companies that have more restrict rules and regulations.

[Image of a flowchart showing the process of proposed verification by Luis Garcia et al.]

In [30] Abbas Jafari et al. propose a solution on monitoring PLC enabled Glazier9 using
external hardware such Personal Computer (PC). Researchers suggest a pre-existing technology
for interfacing the PLC to the PC such as EtherCAT, USB, RS-23210 or RS-48511. Since each
memory location in PLC has an unique address and can be accessed easily the proposed solution
by Abbas Jafari et al. take advantage of this fact. The monitoring program uses high level
languages such as C#12 or MATLAB13 that is running on a PC, there is a companion component
(service-like) application that is running in the background and listens to the connected port.
This piece of component can be written in DOS for faster response time. The communication
between monitoring application and PLC is done by shared files. When PLC is running it will
update the memory addresses and the interfacing part of the code will read those memory changes
and update the files. Simultaneously monitoring application will read these files for changes and
executing analysis, upon decision making it can send new instructions to the same PLC memory.
This solution can provide a very fast accessing time and the only restriction is PLC’s CPU clock
speed. Abbas Jafari et al solution based on the fact that it can manipulate PLC addresses directly
for command instruction, in small applications this is a feasible solution. On the other hand in
bigger scenarios when dozens of PLCs are communicating and working together and there are
lots of different sub components that need manipulation, Abbas’s solution need to be aware of
all this subsystems and manipulations which defeats the purpose of scalability and modularity.

---

9Glazier: is an automatic cutting-machine that cuts crystal and glass in pre-defined size to be used in constriction, building and so on
10RS-232 is a standard for transmitting data in a serial manner between two or more devices
11RS-485 is a supper-set of RS-422 standard which enables serial transmission with more extended scenarios
12C# is a multi-paradigm programming language
13MATLAB is a multi-paradigm numerical computing environment that is capable of running other codes such as C/C++
Nevertheless solution proposed in this thesis is a sub-system in a bigger system and does not need to implement all the interfaces of other subsystems. The communication is handled by other parts and this part will make use of current components that makes scaling up and integration less dependent and more modular.

Yu Wei et al. present an online state monitor system for a 6M25 Compressor in [31]. The compressor is functioning in a poor-working-condition production plant and due to the harsh environment all sorts of mechanical, thermal and performance faults were excessively happening. A two-stage monitoring system were proposed. A master node which was an industrial computer and a slave node (PLC) were introduced. The PLC was connected to all the sensors and transducers. The monitoring program was running on the industrial computer and connecting to the PLC via Message Passing Interface (MPI) for .Net. All the working conditions and properties of malfunctioning compressor were extracted after observation and documented as the specification for monitor system. All the sensors providing information from compressor’s temperature, vibration, exhaust gas temperature were sending information to the PLC which in return were captured by monitor tool. Authors concluded that combining 6M25 Compressor technical flow and monitoring system results in a more reliable operation of the production plant. It is a novel solution with a specific target. It is specially tailored for the 6M25 Compressor and act like a black box and makes any attempts for extension a cumbersome one. Although the proposed solution in this thesis is also targeting towards specific hardware for Bombardier tooling, it is designed in such a way that can be easily tweaked and extended by adding more sub components and introducing more specifications and requirement using a formal specification method.

In [32] Ognjen Bjelica et al. introduce a cost effective method for verification of PLC codes via simulation of PLC environment. Firstly a modular circuitry is designed with input/output to act as targeted system (PLC controlled system). Then the monitoring application is running on windows which is developed with Microsoft Visual Basic.Net. The communication between PC and circuitry is managed by serial protocol, according to [32] because of re-programmability of the microcontroller on the circuit different programs can be uploaded to the system so makes it possible to be scalable for different scenarios. Although a shortcoming of this approach is the latency and slow speed of serial communication between PC and circuit. Ognjen Bjelica et al’s solution is simulating a small array of functionalists in a PLC system whereas a bigger system can have much more functionalists to be simulated or monitored. Another shortcoming is the fact that it does not compare the runtime execution traces against pre-defined specifications. The solution introduced in this thesis will monitor the signals and runtime of a system and can dynamically or statically call a comparison against a specification in the database. All the specifications can be converted using a formal language and many different scenarios can be monitored.

### 3.2 Formal specification

A key factor of reducing confusion and misunderstanding between developers, functional analyst and testers in a development environment is to declare specification and functionalists in a language that is both concrete and easy to understand or in other word is formal with proper syntax and convention. Depending on the target system and development environment different formal languages are proposed, and for this particular thesis an array of formal specification methods and languages were explored. A first attempt was made using Gherkin, As Abigail et al. are mentioning that Gherkin has been used in some scenarios as a communication tool between developers and clients, so the features and specifications can be drawn by clients to speed up and effectively communicate [33]. The authors introduced a method to extract test and monitors for medical devices using Gherkin. Developers and designers use this language to declare the specifications and later the scrip will be compiled by Cucumber (a Gherkin script compiler) for validation. The syntax used by Gherkin and its integration with some open source Integrated Development Environment (IDE) such as Eclipse makes it a tempting choice for formal specification. However it is not very familiar in the PLC world, most PLC developers are used to techniques such as ladder logic, block diagram and so forth and Gherkin is much more similar to natural language. Another aspect is the notion of time and state in the realtime system as well as automatic code generation

---

14 6M25 compressor is a power equipment that is used heavily in ammonia production

15 It is a programming framework from Microsoft®
from specification, all of which pointed to using a more compatible language (PLC-like) for this thesis. One of the most recent attempts to specify PLC components formally is PLCSpecif [34] which is similar to PLC development as well.

Darvas et al. use this method (PLCSpecif) in [34] to check the conformance of PLC program and specification in CERN. In the very beginning authors defined the relation between formal definition and conformance. In order to simplify timing, the length of each PLC cycle was flagged as non-deterministic with an accuracy of one milliseconds, furthermore user defined interrupts were ignored and only communication or OS related interrupts were served during execution. Strict equivalence relation, permissive conformance relations with fixed and variable delay were studied. For the first scenario same output sequence needs to be achieved for the same input sequence under similar conditions and timing. For permissive conformance relations with fixed delay, each delay could be few PLC cycle apart for the output with same input and timing; this would results in an identical shape for output signal. The last conformance was permissive conformance relations with variable delay that allowed different amount of delays for between output sequences as well as execution time. In the later stage PLCVerif (a tool that provide verification of PLC code using model checking techniques) were used to evaluate and verify the specification which were produced by PLCSpecif. Authors provides methods for checking the validity of the approach for scalability and permissive relations. The methods introduced by Darvas et al. are specifically designed and tested on the equipment in the CERN facilities and authors claimed that due to the generic nature of PLCVerif, PLCSpecif, temporal logic and few more tooling used in their work, it can be easily scaled up and be extended. However the method proposed in this thesis makes use of PLCSpecif for only specification declaration, and decreases dependency on proprietary technologies and uses open standards such as XML for storing and extracting information, hence facilitate the chances of extensibility and usability in different environments.
4 Method

This section presents a viable solution to satisfy those issues and questions that were raised in the problem formulation section in the context of this case study being carried out in Bombardier Transportation AB. Firstly the process in which the method arised from will be explained, then it will be followed by a more detailed and in depth explanation of the proposed solution and methods for approaching it.

4.1 Problem solving process

Figure 7 depicts the path for research and solution proposal for this thesis. Repeating some of the texts or topics that were introduced during background of the thesis is inevitable, and prevents jumping back and forth between different areas of the report. After studying the literature that are relevant to the thesis’s topic a preliminary solution or hypothesis will shape, then the design process will implement and apply the solution after which, it will be tested and optimized. The results of the tests will be analyzed and and if necessary the proposed solution or hypothesis will be refined and the process will run once again for better results. Upon satisfactory testing and results analysis, a conclusion of the findings and final results will be discussed.

4.2 Motivation domain requirement

Specification implementation — A good and clear specification can always lead to a more robust product, finding a common ground and common way of declaring a specification which corresponds to the implementation can significantly improve the development, test and deployment time, this is one of the key motivation of the case study.

Conformance verification — Once a system is running, how much it deviates/corresponds from/to the expected result can be of utmost importance, specially in the case of realtime system which a failure can be a catastrophic event. Such events can be detected much earlier with the help of RV in as a complementary method to the common testing in software development.

4.3 Design

The monitoring process will be implemented on an external hardware such as a PC or an industrial computer. Figure 8 shows the overall process of the design. PLC will be connected to the monitoring system, monitoring system consist of different components, a windows app will be constantly monitoring the connected port (via in-house software) and log all the incoming information to a shared file. Then the monitoring engine (Windows application) access the file for analysis, it will compare the results against a specification database for any timing violation, functional incorrectness and so forth. All these finding will be logged and a verdict will be displaced to the operator for further action. There is a possibility of sending a feedback to the PLC as a fault detection warning or recovery code.
Limitation — There are few limitations involving this thesis, first and foremost is the PLC equipment and its working system. Due to complexity and tight testing schedule in Bombardier there is limited number of time to work with real hardware. Since most development environment for PLC and verification systems are commercial and highly costly it is wise to account for unwillingness of companies for sharing their home built technologies such as specification extraction and feeding them to a system. The proposed method for formal specification is based on PLCspecif which is very new and is specifically developed for CERN makes it even more difficult to access documentations and tutorials. Although authors of the methods are being contacted for more instructions. Last but not least, is the time limitation, which shifts the focus on only monitoring and logging rather than feedback and recovery measures.

4.4 Targeted System for monitoring

Instrumented system will be a PLC enabled automation system (C30-Train) in Bombardier\textsuperscript{16}. This is a modern train that is controlled by Bombardier own control system named Train Control and Management System(TCMS) which will be explained in later sections. The proposed system uses DDE protocol to communicate with the entire control system via another tool called DCUTerm, short for: (Data Communication User Terminal). DCUTerm is able to communicate with both real hardware or simulated hardware which in turn makes it possible for the proposed solution to be able to communicate with real or simulated hardware.

4.4.1 C30 Train

C30 is one of the many train models that Bombardier manufactures. It is made for Stockholms Lokaltrafik\textsuperscript{17} (SL) and can go as fast as 90 km/h. The control system and management for this train is based on Bombardier’s TCMS.

4.4.2 TCMS

TCMS is an array of hardware and software which provides control system, communication, infotainment and all the control elements of a train. It is consist of many sub systems (e.g brake system, door and propulsion system), all of which are communicating and working together to provide a seamless control over the train. TCMS is based on MITRAC® CC. MITRAC® CC is a computer system consisting of hardware, OS, development-diagnostic-and-maintenance tools developed specifically for applications in rail vehicles and manages all the monitoring and controlling

\textsuperscript{16}http://www.bombardier.com/en/home.html

\textsuperscript{17}http://www.bombardier.com/en/transportation/projects/project.MOVIA-C30-Stockholm-Sweden.html?f-region=all&f-country=fr&f-segment=all&f-name=all
of the vehicle. Figure 9 depicts a super simplified imitation of a TCMS box. All the components (hardware/software) of MITRAC® CC are made by Bombardier which makes it rely less on subcontractors and decreases response time for any bug fixing or issue tracking.

![TCMS Box](image)

Figure 9: TCMS Box (ridiculously simplified!)

### 4.4.3 SoftTCMS

SoftTCMS is a simulated environment (framework) for TCMS systems. All the subsystems such as communication (busses and gateways), control unit (SoftCCUs) and etc are loaded in a separate application. All these subsystems are glued together to simulate a working TCMS system. Developers can use SoftTCMS to simulate how the code will be running on the hardware. Furthermore it can be used for monitoring, debugging and signal processing.

### 4.4.4 DCUTerm

DCUTerm is a terminal program, running on a PC, used to communicate with computer units with processor capability within the MITRAC® CC family. It can also be used as part of SoftTCMS to handle the communication between simulated environment if the simulated system contains a monitor with a TCP/IP connection. Proposed solution for this thesis will heavily make use of DCUTerm to catch signals coming from the system and also for sending commands to simulated/real environment. The communication between DCUTerm and proposed solution will be over DDE protocol.

### 4.4.5 Current process

Figure 10 shows how the current process and cycle works. The output of the MITRAC IDE\(^\text{18}\) will be a hardware specific (dependant) "c" code that will be executing on MITRAC® CC system. In order to test and debug developed code during development, a simulated environment (SoftTCMS) will be used. Firstly the generated "c" code will be converted to a standard ANSI library (without hardware dependant libraries). Then this code will be compiled into a Windows binary and other subsystems such as communication protocols, control unit, network components and so forth will be added to the simulation scenario. Once all the subsystems are ready, a TCMS system's procedure can be executed in the simulated environment and be monitored by DCUTerm.

### 4.4.6 New process

In the new process the proposed solution (MVTerm) will be communicating with DCUTerm. It will execute it and listen to the communication. MVTerm in turn cross check the signals value with the specifications that are stored in a database. This specification are generated form the software requirements for C30 train (requirements are in natural language). The requirements are converted to formal specification using PLCSpecif method. Figure 11 shows a rough sketch of new process.

---

\(^{18}\) An IDE developed and customized by Bombadier for in-house development of PLC and control systems
4.5 Interface between monitoring and targeted system

Most PLCs are equipped with standard interfaces, such as Ethernet, Serial port, EtherCAT etc. Apart from these standards conventional TCP/IP is in use in Bombardier as well. Since the solution is suggested for a realtime system (PLC) selection of a fast and reliable communication channel is of prime importance. Serial communications are limited by speed and suffers from latency based on the standard used. The current popular choice for realtime monitoring communication is EtherCAT, however TCP/IP will be used since it is already supported by all the tooling in Bombardier and requires minimum effort to be utilized.
4.6 Specification of targeted system

Specifications of the system can be extracted by observing the system and combining it with formal specification methods. These specifications are the main ingredients in the decision making process of producing a verdict. So proper formalizing and fast access to the specifications is crucial to the final result. To realize this goal an Relational Database Management System (RDBMS) is used. And the converted specification will be stored there.

4.7 MVTerm

This is the name of the tool for the proposed solution and refers to the solution as a whole which includes a Windows application, database (xml based) of specification and a formal method of specification.

4.7.1 Hardware for monitoring system

Monitoring system has to run on a powerful enough machine to perform all the analysis and monitoring in a fraction of a second, and come up with a verdict to be logged and reported to operator for necessary actions. So the hardware selected should be capable of running Windows platform with necessary input/outputs. Most modern PCs with a high end CPU (e.g 2.0 GHz) and RAM (8.0 GB) can run the monitoring system. Other criteria such as input ports are to be decided.

4.7.2 Software for monitoring system

The proposed monitoring system is based on Windows platform, even though Windows is not a realtime operating system (OS) it can deliver monitoring results with low latency if properly integrate with current tooling in Bombardier. The main application will be written with Microsoft C# and the graphical user interface (GUI) will be generated using Windows presentation foundation (WPF). WPF is used due to its high efficiency in Windows platform, there is a need to draw signals and graphs on the screen and WPF can facilitate this task efficiently.

Visual Studio 2017 Community — Visual Studio provides great support and knowledge base for programming environment for Windows hence most of the monitoring system will be developed by this tool. It natively supports communication with ports such as serial, Ethernet and so forth. The communication with RDBMS can be done with minimum effort as well as built-in file access system.

Microsoft SQL Server Community — All the specification and possibly logging can be stored in SQL files, using a RDBMS makes it possible to do further analysis on the gathered information not to mentions logging and specifications can grow rapidly and so accessing time can be a challenge, by employing a database system not only the access time will be improved but also scaling up the application can be easily done.

Extensible Markup Language (XML) — This is an alternative to the RDBMS and can be used for fast prototyping, Microsoft .Net supports it natively and provides fast accessing and manipulating for XML documents via specially designed XML providers.

MATLAB 2016 — MATLAB can provide a great toolbox for analysis of information, MATLAB will be used as a tool for statistic purposes and evaluation of gathered information.

DCUTerm v4.1 (Case study dependent) — DCUTerm (Bombardier internal tool) will be used to communicate with other subsystem in the SoftTCMS or TCMS enviroment. The latest version while writting this report is 4.1 and that is the one used for this thesis.

MDCT v2.6.3.0 (Case study dependent) — Mitract desktop component tester is an internal tool by Bombardier that starts the train software in a simulated environment (SoftCCU) on a regu-
lar PC and testers and developers can see the code execution visually and invoke any subsystem on demand. This tool will be used to identify which signals or income/outcome values are interesting to be monitored by DCUTerm. It can support single or multiple SoftCCUs for master-local logic testing.

**SoftCCU (Case study dependent)** — It is a console application that will be invoked directly by tools such as MDCT and simulates the logic of a CCU in a PC environment.

**IoTool (Case study dependent)** — It is a Window application that is capable of manipulating the simulated memory in the SoftTCMS environment. All the PLC memory locations in the SoftTCMS are available to this application and they can be overridden for a specific amount of time with a specific value.

Other tools will be investigated but not mentioned since they are too specific for this case study.

### 4.8 Effects of RV in PLC development

Most of the fallout against RV system can be around process and project management point of view such as; cost, learning curve, time to develop and integration. Whereas, upon proper integration the overall outcome will make a more reliable and high quality output. Conventional method of quality assurance for software development to assess security and reliability were based on some static analysis and tools that are based on some known rules and warn the user if it faces any violation [35]. One hypothetical example that can highlight the positive effects of introducing RV in the PLC development life cycles is; testing if a task with high priority and specific deadline behaves accordingly in a run. For instance Task A with a period of two milliseconds supposed to finish at certain deadline, using conventional testing method, a developer or a tester could make a unit test or test script that run the task and see if it meets its deadline. The problem with this method of testing is that, the test is designed (unintentionally by developer or tester) in such a way that is intended to pass and with no means is a representative of real time situation with hundreds of tasks running. Now assume there is a RV system in place which is passively monitoring that specific task (Task A) while the entire system is running, the monitor can log and see if the task is near missing deadline, missing deadline or passing. Another information that is extractable by this method is that anomalies which are usually undetectable via conventional methods can be seen and analyzed.

![Figure 12: Task A: 10 executions histogram (Normal)](image)

Task A execution timeline is shown in figure 12, this is a histogram of ten (each black dot is a run) executions of the task and shows that almost all the runs were within the specified deadline of two milliseconds, on the contrary figure 13 illustrates another ten executions and it shows that the task is awfully close to miss deadline (even though still within rage) and can be flagged for further investigation by developers to see what is the root cause of near missing deadline scenarios. Or another scenario can be if the task is tested in the simulated environment it seems to have a high response time whereas under certain scenarios it throttle and almost misses the deadline. All these sort of information can be implicitly extracted from an RV system. The caveat is, all these benefits from RV system will not reduce or replace conventional testing or safety protocols, it will
be an extra level of checking that most companies does not want to invest on. However there are few areas such as system engineering, product certification, safety assessment and code verification that formal verification which is a prerequisite for a good RV has gained foothold in the last few years [36]. Even though the term runtime verification is only over a decade old, it is becoming an active and popular research area due to its power for evaluating correctness and situations that are not simply possible by using conventional testing and model checking [37]. Practical formal methods combined with RV can provide high level of confidence in final product and productivity in long run for safety critical systems.
5 Technical Approach

This chapter is divided into two main sub sections, the first part explains and highlights the method which is used for conversion of requirement into specification using a formal language. The latter sub section will depict and illustrate the implementation of the system and dive deep into technical details of software development point of view.

5.1 Converting requirements to formal specification

Understanding and producing software requirements is an integral part of software development process, and it is usually conducted via a form of natural language and are often incomplete and ambiguous [38]. Many good practices are developed that helps to streamline the process of Software Requirements Specification (SRS) even few attempts to use natural language processing to do so such as in [38]. Most requirements in Bombardier systems are declared using natural language and does not follow any formal language. This is a conscious decision since investing in a formal specification method can be costly and time consuming in the beginning. The method (PLCSpecif) proposed in this thesis can benefit the company in long run. The formal language used by PLC-Specif is very similar to PLC logic and structure hence lower learning curve for current employees. Another benefit is that specifications declared using this method are more clear and straight to the point which can reduce misunderstanding between developers and functional analysts, furthermore authors of PLCSpecif has introduced some tooling and methods in [23] which can convert the output of PLCSpecif to Structured Text (ST) that are directly corresponding to the declared specifications which in turn reduces debugging and implementation efforts dramatically.

Since this is a proof of concept and due to lack of time/resources only two software requirements are chosen to be evaluated using PLCSpecif and then those corresponding signals will be monitored via MCUTerm and a final verdict will be shown to the user. Table 2 and 3 show the interested software requirements and their information.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRS1</td>
<td>Functional — An analog output signal (MWT_REAL) that shows train’s speed, it has dependency on DBC_SV_Xv_TrnsPd (boolean signal [MWT_BOOL]) which shows if the speed reading is valid or not.</td>
</tr>
<tr>
<td>Signal Name</td>
<td>DBC_SV_Xv_TrnsPd</td>
</tr>
<tr>
<td>Location:</td>
<td>LAB5/SpdCalc_T2</td>
</tr>
<tr>
<td>Function</td>
<td>block → DBC_SV_TrnsPdCalc</td>
</tr>
<tr>
<td>XML name</td>
<td>TSpeed</td>
</tr>
</tbody>
</table>

Table 2: Software requirement 1 to be monitored.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRS2</td>
<td>None functional — This signals shows the cpu load of the running system</td>
</tr>
<tr>
<td>Signal Name</td>
<td>DIAG DV X CpuLdAvg</td>
</tr>
<tr>
<td>Location:</td>
<td>LAB5/SystemSupervision</td>
</tr>
<tr>
<td>Function block</td>
<td>→ DIAG_DV_CpuLdAvg</td>
</tr>
<tr>
<td>XML name</td>
<td>TCPULoad</td>
</tr>
</tbody>
</table>

Table 3: Software requirement 2 to be monitored.
SRS1 — Figure 14 shows the connections between input, output and core logic connection in SRS1 in order to calculate the train speed. Inputs are coming from four different axles’ speed value which are placed in the front and center of train. The maximum value of first two (axle1 and axle2) will be compared with the maximum value of axle15 and axle16, then the highest value will be displayed as the train speed. There is another value of interest which is named (IsValid) this value determines if the calculated speed is valid or not, the specifications that sets the value (boolean) are based on different situations and criteria such as: is the brake engaged or not and so forth (it is not mentioned here for the sake of simplicity and space). Input-output connection module is used in the core logic which is distinguishable by the double arrows which connects input and output to the core logic module, this is done due to the fact that calculation of speed depends on many other incoming information and does not require complex state machine like scenarios. The PLCspecif’s noble method of separating core-logic from I/O is depicted here that simplifies understanding and execution.

Next step is to produce a (simple/leaf) module that incorporates the mentioned SRS1 in figure 15; to do so an input-output connection module (which is one of the three available module types in PLCspecif: state machine, data-flow diagram, PLC timers) is used. This specification (15) consists of six different sections:

Variable declaration — which states all the input/output variables necessary for this module including axle1 to axle16, and also Max_Sp which is a constant holding the maximum speed that this particular train (C30) can travel. Furthermore another variable named Sp_Valid which is a boolean variable and holds a value that determines is the speed coming from axles are valid so far.

Input definitions — that constrains the input variables, no constraint in this case.

Event definitions — declaration of events and their triggering scenarios, there are two events involved in this module: (1.) brakeActivated which is fired based on value coming from Read-BrakeSignal, (2.) brakeDeactivated that is set by ReadAccelerationSignal.

Core logic — Since speed calculation is an arithmetic operation and the value can be displayed using numbers Input-output connection module deceleration of logic is chose which is based on data-flow diagram hence familiar for PLC developers. Each pair of axles’ speed will be input of a max function and eventually the output will go through another max function and the final result is the train speed. Although the validity of this speed will be checked later on.

Output definitions — This section sets the output values and constrain them, for this use case the value of Sp_Valid is of utmost importance and determines the validity of train speed. This is done by utilizing an AND table which has three input arguments and each of which can hold true or false values. The inputs are: (1.) if train speed is less than maximum speed, (2.) if brake signal is off), (3.) if the previously calculated speed is valid (Sp_Valid). Since each argument can only represents two value (true/false) there will be eight possible combinations. \(2^3 = 8\)
### Invariant properties

This section can embed all the invariant properties of the modules such as checking if the manual brake should not be engaged if calculating the speed and so far, in this case, this is a hypothetical setting/properties which states that upon successful evaluation of all conditions, it assumes that train speed should be always between 0 and maximum possible speed.

**Figure 15**: PLCSpecif’s module specification of train speed

### SRS2

This is a diagnostic signal which is used to evaluate and retrieve the CPU load of the system, there is no need for PLCSpecif representation of the SRS.

### 5.2 Converting formal specification to database

The output of PLCSpecif can be directly used to generate PLC code and be used in development and SRS writing, however since the authors of the tool (PLCSpecif) have not provided it by the time this thesis has been being written, the alternative is to convert the proposed method (PLCSpecif) into a database to be fed to the monitor system. This database should be easy to create, maintain.
and extend. The original plan was to utilize a RDBMS for storing and fetching specifications, due to lack of time XML which is a document-oriented database will be applied. However using XML will provide more flexibility for fast prototyping and the data stored in it can be directly queried using XQuery and serialized for porting to other database system with minimum effort. Another reason to use XML is that due to extensibility of its elements the code using them will not break, for instance adding a new element will not affect the handling-code, although the handling-code needs to be modified to make use of the new element. But there are times that new information needed to be added to the file (database) and the manipulation code will be added later which makes this approach applicable. For consistency a code convention for designing XML file is introduced which is described in listing 4. First line specifies the XML’s scheme and encoding version, line two is the main parent node named \textit{MVTerm} after the application name. Third node \textit{MVTermSignals} contains all the signals that are being monitored, each \textit{signal} node then includes five more elements, each of which describes an attribute of the interested signal. The first child element of signal node is \textit{name} that contains the exact signal name which is needed to be sent to DCUTerm, next is \textit{period} that specifies the period of the signal, coming after is \textit{priority} element that indicates the priority of the task that signals is part of. It is then followed by \textit{checkPeriod} elements that tells whether MVTerm, should monitor the execution time of the signal/task or not (if the value cannot be extracted from DCUTerm, it will be estimated using simple calculation of invoking time and returning signal time). The last element of signal node is \textit{checkCorrectness}. Checking correctness of a signal requires to embed the correct (expected) execution into the XML file as well or retrieve it from some sort of database which is not a trivial task, for this case study the correctness will be interpreted if the task does not violates its deadline, or for a simple task the action can be translated into simple mathematical (like train’s speed) or boolean value.

Camel-casing\footnote{It is a method of phrasing compound elements so the word(s) in the middle start with a capital letter for easier reading.} should be used for children elements. If there is only one word it should be lowered case. All child elements need to be indented properly using Tab (configured to be four spaces; avoid pressing space-bar four times!). All elements have open and close tags, since most of the tags will have child elements, DONOT use short-hand closing tag.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<MVTerm>
  <MVTermSignals>
    <signal>
      <name>DBC_CT_S_EmbApl</name>
      <period>2</period>
      <priority>5</priority>
      <checkPeriod>1</checkPeriod>
      <checkCorrectness>1</checkCorrectness>
      <PLCSpecificCommands>
        <command id="01">
          <body>ALWAYS trainSpd STET argument0 ASSUMING SpdCalc IS true</body>
          <arguments>
            <argument name="trainMaxSpeed" unit="kmh" type="int">90</argument>
          </arguments>
        </command>
      </PLCSpecificCommands>
    </signal>
  </MVTermSignals>
</MVTerm>
```

Code Snippet 4: XML declaration of specification

5.3 MVTerm implementation

The main implementation of the solution will be described in the following sub sections. Firstly an overall architecture of MVTerm will be explained and will be followed by in depth details of how it communicates and interacts with other components in the system. Next the process of coming up with a verdict will be shown and finally a demo and some screen shots of the final tool and
results. Good programming and practices such as SOLID\(^{20}\) principle and MVVM\(^{21}\) design pattern are partially used with code-behind for scalability and extensibility of solutions. The application is mainly written using C# (C# 7.0 tulips, in-line functions are used) and the GUI is based on WPF.

### 5.3.1 Architecture

Figure 16 shows a high level architecture that includes all the containers of MVTerm application. Each container is responsible for similar operations within MVTerm, such approach will increase extensibility of the application for future.

![Diagram showing MVTerm’s containers](image)

**Figure 16: MVTerm’s containers**

### 5.3.2 DCUTerm script and setup

A script which initializes and setups DCUTerm for the specific purpose of this proof of concept (PoC) has been prepared. The script adds nine buttons to the DCUTerm’s main GUI. Each button is responsible for an automated task such as hooking up all the signals, manipulating train’s speed and so forth. Figure 17 shows the generated buttons at the bottom of the main windows. The automated scripts behind each button are as follows:

- **Init** — This will disconnects all the current signals, dump the current log file and hook up all the signals again.
- **Reset** — This will act in the same manner as reset except dumping the log file.
- **Speed = 0** — Set the train’s speed to 0 kmh.

\(^{20}\)It is the first five object-oriented principles by Robert C. Martin and promote designing software in such a way that is easy to maintain and extend: https://scotch.io/bar-talk/s-o-l-i-d-the-first-five-principles-of-object-oriented-design

\(^{21}\)Model-view-viewmodel (MVVM) is a software architectural pattern which is developed by Microsoft® and facilitates a separation of development of the graphical user interface.
Speed = 5 — Set the train’s speed to 5 kmh.
Speed = 20 — Set the train’s speed to 20 kmh.
Speed = 80 — Set the train’s speed to 80 kmh.
Speed = 150 — Set the train’s speed to 150 kmh.

Speed Calc = valid — Set a flag in the systems to true which indicates that the calculated speed is valid.

Speed Calc = invalid — Set a flag in the systems to false which indicates that the calculated speed is invalid.

Figure 17: DCUTerm GUI

Providing a script to manipulate the interested signals automatically without typing them in the DCUTerm’s terminal will increase testing speed, and minimize manual efforts. This script is named mvtermsetup.dts and can be extended for future use.

5.3.3 Communication between DCUTerm and MVTerm

DCUTerm has the ability to act as a COM server which is utilized in this tool. MVTerm references the DCUTerm executable and extracts its interfaces via reflection (automatically done by .Net framework). Once a reference to DCUTerm is attained it will be executed via MVTerm, whilst both tools are running; DCUTerm will act as a server and MVTerm will be the client. MVTerm’s main window is shown in appendix A. After clicking on the “Connect to DCUTerm” button, MVTerm will execute DCUTerm and a handle to the app will be held internally. This handle will be used throughout the application to communicate with DCUTerm. DCUTerm comes with a manual for its API. This manual explains which functions and services inside DCUTerm can be accessed via COM service. Table 4 lists the most used commands (API) in DCUTerm which are extensively applied to send/receive commands or monitor DCUTerm via MVTerm. It is worth mentioning that DCUTerm has a long list of API that can be utilized.

This code snippet 5 depicts the way that a handle to DCUTerm is being created inside MVTerm. It is wise to make the handle to be protected (access modifier) which makes it available throughout all the classes that inherits from the main class. All the functionalists are separated into smallest possible form and each action can be put into a function (void or non-void) and that function

\[22\] dts is proprietary extension for DCUTerm and the setup file will be saved on the Bombardier resources
<table>
<thead>
<tr>
<th>API</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConnectWith</td>
<td>This command connect to the DCUTerm app with specified arguments</td>
</tr>
<tr>
<td>GetTraceWinLastSampleSet</td>
<td>It will read the DCUTerm’s trace window continuously and read the latest</td>
</tr>
<tr>
<td></td>
<td>samples that are available in the trace window</td>
</tr>
<tr>
<td>ExecuteCommand</td>
<td>Sends a single command to DCUTerm and execute it</td>
</tr>
<tr>
<td>EmptyCommandQueue</td>
<td>This API will empty all the queued command in DCUTerm that have not executed</td>
</tr>
<tr>
<td>ExecuteCommandsFromFile</td>
<td>This API will send a text file to DCUTerm and DCUTerm will execute that file</td>
</tr>
</tbody>
</table>

Table 4: DCUTerm most used APIs list

will be called from there on. There is a simple check of a boolean flag in the beginning which prevent double execution of "connecting code". This is the only code snippet of MVTerm that is displayed (due to space being premium and also company policy), a private code repository (Git) will house all the generated codes, for further information regarding accessing documents or source code contact the author.

```java
private DCUTerm.App DCUTermApp;
private void ConnectToDCUTerm()
{
    try
    {
        if (DCUTermIsConnected) return;
        DCUTermApp = new DCUTerm.App();
        // Since DCUTerm needs some times to start, we delay current thread before connecting to it */
        System.Threading.Thread.Sleep(1000);
        DCUTermApp.ConnectWith(DCUTermConfiguration.ConnectType,
                                DCUTermConfiguration.Target, DCUTermConfiguration.Port);
        DCUTermIsConnected = true;
        stopWatch.Start();
    }
    catch (System.Exception ex)
    {
        DCUTermIsConnected = false;
        Debug.Assert(false, ex.ToString());
    }
}
```

Code Snippet 5: Connecting DCUTerm and MVTerm

5.3.4 Stimulating the target system (simulated CCU)

In order to obtain the results for the next stage, the system needs to be stimulated using different scenarios and cases. The stimuli for this thesis are designed to be effective at component level rather than system level on account of that the comprehensive system could not be executed and stimulated on test machine (the machine that this thesis was developed on, however there are tooling and systems (VCS) that can simulate the entire system but are out of scope for this thesis because of time limitation). The stimuli that will be injecting to the system will manipulate the
axles’ properties (diameter, rotational speed) which results in train’s speed changes. The injection of stimuli can be done manually or automatically. For automatic way; IoTool (described in page 21) is used to generate a script that will house all the targeted signals (table 5). Once this script is running it will override and manipulate the signals for a specific amount of time with predefined values. In addition for manual way Bombardier’s MDCT (described in page 20) can be used. There is another automatic way that can manipulate signals more efficiently using DCUTerm (described in page 20), however, at the moment of writing this thesis DCUTerm is limited to running a single instance as simulated COM service which is why the stimulation will use IoTool. A DCUTerm scrip (appendix B) and IoTool scrip (appendix C) are prepared. Table 5 shows the targeted signals and their descriptions (note: real signal names are changed to these short names due to confidentiality reasons).

<table>
<thead>
<tr>
<th>Signal Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>En1</td>
<td>Enables the sub function block for calculating speed using axles properties</td>
</tr>
<tr>
<td>En2</td>
<td>This signal enables the function block for calculating speed</td>
</tr>
<tr>
<td>En3</td>
<td>This signals enable the entire block of speed calculation</td>
</tr>
<tr>
<td>Ax11</td>
<td>Sets the rotational speed for wheel</td>
</tr>
<tr>
<td>Ax12</td>
<td>Enables axle 1 values</td>
</tr>
<tr>
<td>Ax13</td>
<td>Sets the wheel diameter for axle 1</td>
</tr>
<tr>
<td>Ax21</td>
<td>Sets the rotational speed for wheel</td>
</tr>
<tr>
<td>Ax22</td>
<td>Enables axle 2 values</td>
</tr>
<tr>
<td>Ax23</td>
<td>Sets the wheel diameter for axle 2</td>
</tr>
<tr>
<td>Ax31</td>
<td>Sets the rotational speed for wheel</td>
</tr>
<tr>
<td>Ax32</td>
<td>Enables axle 2 values</td>
</tr>
<tr>
<td>Ax33</td>
<td>Sets the wheel diameter for axle 2</td>
</tr>
<tr>
<td>Ax41</td>
<td>Sets the rotational speed for wheel</td>
</tr>
<tr>
<td>Ax42</td>
<td>Enables axle 1 values (BCU12)</td>
</tr>
<tr>
<td>Ax43</td>
<td>Sets the wheel diameter for axle 1</td>
</tr>
</tbody>
</table>

Table 5: Signals to manipulate

A train might have different number of consist, and each consist will have its local speed (this speed will be used for internal purposes). In order to calculate the speed those wheels attached to none-driven axle will be used to reduce self-posing values (sliding, traction, etc) from axles itself. There are different computation units in each consist, each of which is responsible for different tasks such as fire-alarm, entertainment, communication, brake, drive units and so forth. At any given time there are many tasks that are being computed and processed, those important tasks such speed calculation are placed inside tasks with higher priority and OS guarantees some process time for these high priority tasks. However in safety critical system one should look out for any misbehaviour and situations that a task might starve of processing time. Hence monitoring this
signal to evaluate the processing time can detect unforeseen anomalies. The implementation details such as the formula that calculates the speed is not of our interest and the time it takes is the main stake here. Figure 18 shows the function block section in MDCT (source code) which is responsible for calculating axle speed, the signal values are marked with blue color and correspond to table 5. Once everything is hooked up and the simulation is running, operator can use MDCT and open up this section, by double clicking on each signal there will be a pop up allowing user to input values manually and override the pre-filled (default) values, and this is how the stimulating works in this case study.

![Figure 18: MDCT signal manipulation](image)

The entire steps of stimulating the system is shown in figure 19. Once the system is running (simulation mode), DCUTerm will be executed and initialized with the configuration file (specially desinged for this case) which attach the signals mentioned in table 5 to fast logger (a feature of DCUTerm to log a specific signal) and start the fast logger task. Next, whilst the system is running IoTool will be manipulating the signals using the predefined script. There are three scenarios in which IoTool start manipulation: Normal CPU load, Heavy CPU load, and Mixed CPU load.

**Normal CPU load** — In this mode the CPU on which the simulation is running is under normal work stress, hence IoTool’s scrip will be running in a simulated system under normal stress.

**Heavy CPU load** — In this mode a third party tool (CPUStress\(^{23}\)) is used to mimic high CPU usage, this will utilize CPU time heavily hence IoTool’s scrip will be running in a simulated system under heavy stress.

**Mixed CPU load** — This mode will first run the scrip and then CPUStress will occupy CPU time to see how it affects the response time in the middle of scrip execution.

\(^{23}\)Available at: https://blogs.msdn.microsoft.com/vijaysk/2012/10/26/tools-to-simulate-cpu-memory-disk-load/
Stimuli process — The anatomy of IoTool script that manipulates the axles’ values is as follow: firstly, all the flags that indicate speed calculation using axles is valid, will be set to true for 30 seconds, next, the axle1’s wheel diameter and rotational speed will be set to 50cm and 5000ms respectively. Other values will change in the same manner and there is a five seconds offset between each axle’s update which allows us to visually see the difference in the signal window. A XML representation of IoTool script is shown in code snippet 6 (real signal name is changed due to confidentiality).

```xml
<xml version="1.0" standalone="yes"/>
<DocumentElement>
  <ToXml>
    <Signal_Name>Signal Name</Signal_Name>
    <Override_Value>1</Override_Value>
    <Override_Time>300</Override_Time>
    <StartTime_Offset>0</StartTime_Offset>
  </ToXml>
</DocumentElement>
```

Code Snippet 6: IoTool script (XML declaration)

5.3.5 Producing verdict and visual indication

All the incoming information from DCUTerm will be currently saved into an in-memory dictionary, and can be easily transferred into any database. Dictionaries are used to speed up the process of development, furthermore this data type (dictionary in .Net framework concept) provides fast data access with key-pair values for sorting, manipulation and retrieving any value based on the key. When DCUTerm produces any information in the trace window, this information (signals’ values) will be intercepted by MVTerm. Communication with DCUTerm for new information can take place in three different fashion:

- Pulling
- Multi-threading
- Events (not applicable for now): It requires the DCUTerm developers to extend their API and include an event to which some delegates can subscribe. This will be the most efficient way and will eliminate all the boiler-plates code and guards against mutation, concurrency and thread management.

Pulling is the most simplest way to get information from DCUTerm, however it causes an array of difficulties and restrictions. For instance constant pulling will use CPU time unwisely, the code
will be complicated and very difficult to extend, the main thread will be busy serving the main loop in the pulling scenario and so forth. That is the main reason to avoid this technique, the second best alternative is to use a multi-threaded scenario with the help of timers and dispatchers\textsuperscript{24}. Most of the heavy duty task will be processed in the main thread and after a new result is obtained it will update the UI using UI thread (dispatcher). Then a sub function which is responsible to analyze the incoming information from DCUTerm will be called in the callback function of the timer. The resolution of the timer can be adjusted to suits the needs (since Windows is not a realtime OS and also it communicates with a simulated enviroment, the time keeping system can be improved in the future). There are few different timers in .Net framework, the one which is used in this case is derived from the Threading.Timer namespace and the main reason is that, other timers are executing in a worker thread which is obtained from Common Language Runtime (CLR)’s thread pool and does not provide so much flexibility as timers in Threading.Timers namespace.

The information stored in the dictionary will be compared with the specifications stored in the XML file in a straightforward manner, it will be a one-to-one comparisons between a signal value (speed in this case) and its predefined restriction in the XML, and upon any violation it will be reported in the evaluation area. Figure 30 in appendix C displays a 100 seconds evaluation of speed with plotted information and violated ones as a visual indicator for operator.

5.3.6 Final tool and demo

A working demo of the final proof of concept tool (MVTerm) is hosted on private source control repository and can be obtained by contacting author and supervisor (this is on account of agreement with Bombardier Transport AB).

5.3.7 Tracking time

In order to keep track of time in MVTerm, once monitoring process starts (by pressing Start Monitor button) a stop watch timer will be triggered. Its resolution is set to be 1000 ms, and there will be a mutex checking for its consistency. It will be processed in the main thread as not to block UI thread (Dispatcher). This is a naive approach for time keeping in realtime systems, however for fast prototyping and this proof of concept tool is good enough. Another nice tool in Bombardier repository is IoTool\textsuperscript{25}, this tool give the ability to manipulate the simulated memory location for SoftTCMS and give access to the simulated CCU tickers. The tick counter provided by this tool has a better resolution and accuracy and can be considered in the future work as a complementary clock for the monitor system.

\textsuperscript{24}It a service class that can provide a mechanism to manage different work in a thread both synchronously and asynchronously

\textsuperscript{25}Bombardier internal tool to manipulate simulated memory locations for SoftTCMS
6 Result

This chapter presents the obtained results for this thesis, and it is organized into two sub sections. Firstly the tangible results of applying the stimuli to the system will be presented and it will be followed by an analysis of the results to evaluate the outcome.

6.1 Results of applying stimuli

Figure 20 shows the result of speed signals reacting to the stimuli in the mixed CPU load mode. Data sampling rate of DCUTerm depends on the MCU in which the code is running. According to the documentation of DCUTerm data sampling can be performed on the fastest cyclic MCU task which is normally 4ms. Mitrac CC DCU2 may also contain a fast log engine in the Digital Signal Processor (DSP). This makes it possible to measure samples at full DSP speed which is 50us. The X axes in this figure shows the time line of sampling which is from 12:17:50 to 12:20:00 (130 seconds, the tool takes system time as point of reference for x axes). Y axes will be unique to each signal and will oscillate between possible values of that particular signal. For instance for those signal representing speed (2 to 7) it will be oscillating between minimum to maximum speed. For signal 8 it will be either 1 or 0 that indicates the validity of calculated speed.

Figure 20: DCUTerm axles’ signal values

There are eight signals labeled from 1 to 8 using black font. Signal 1, is representing the CPU load of the targeted system. In this case since it is a simulated environment (SoftTCMS) this signal shows a constant 0 throughout the entire time line. Signal 2 to 5 are connected to the axle 4 to 1 respectively. The lowest speed among axles considered to be 0kmh and the maximum 45kmh (only the integer part will be of importance and the fraction is negligible). Signal 2 displays a speed of 0kmh in the start, and then after each stimulation (three) it shows a spike in the speed to the maximum of 45kmh, each spike lasts for five seconds. Signal 3 also has a 0kmh speed in the beginning which jumps to 36kmh in each stimulation for a total of 15 seconds (five for each). Next, signal 4 also starts with 0kmh and tops to 22kmh each time it is stimulated. The last signal for axles (signal 5) follows the trend of 0kmh start like previous signals and build up towards 13kmh for each stimulation. Signal 6 outlines the maximum calculated speed among all axles. In the beginning it has a value of 0kmh then the first stimulation will charge up axle 1 (signal 5) to 13kmh and then after five seconds another stimulation will take place which increases the speed of axle 2 (signal 4) to 22kmh, coming next is another stimulation for axle 3 (signal 3) which causes an increase of 14kmh in the speed for another five seconds and lastly axle 4 (signal 1) will maximize the speed to 45kmh; all these changes (gradual increase) in the speed is visible in the ladder-like line of signal 6. Signal 7 tracks the final calculated train speed, it is fed by signal 6’s output hence
the similarity in the signal shape. The train speed will be the maximum calculated axle’s speed in any given time. The last signal (8) corresponds to a boolean value in the system which indicates whether the calculated speed is valid or not. On those occupations that stimulation takes place this signal is overridden to echo the correctness of the calculation. There are many sub functions in the system that checks the integrity of the calculation for each individual axle and for the entire system as well, most often the forced-simulated calculated speed causes this flag to negate the validity, hence this is overridden via the automated scrip to prevent invoking the supervisory parts of the system to safety checks.

6.2 Analysis of results

In figure 20, the plot can be divided into three different zones. The first 40 seconds (from 12:17:50 to 12:18:30) happens when CPU is only serving the tasks specified in the simulation (zone 1). The second 40 seconds (from 12:18:30 to 12:19:10) takes place when CPU is heavily loaded with other tasks and then the stimulation is triggered (zone 2). The third 40 seconds starts from 12:19:10 and ends at 12:19:50, during this zone (zone 3) amid stimulation CPU usage varies to effect the response time. Figure 21 shows the mentioned zones differentiated by colors.

The first zone comes about the time when there is no CPU load on the simulated environment. The changes of speed in axles (signal 2 to 5) are projected simultaneously to signal 6 (calculated maximum axle speed) and signal 7 (overall train speed). This response time indicates that there is no significant delay between updating the train speed using axles calculation under normal CPU load. This fact is shown in figure 22. This figure (22) is a zoomed version of zone 1. The speed is at 0kmh at 12:17:40 an it immediately raises to 13kmh once axle 1 raises at 12:17:50 and this speed is kept for the next five seconds. This relationship is connected by a vertical dashed line. The next increase in speed happens at 12:18:00 by axle 2 which pushes up the speed to 22kmh and keeps it there for the next five seconds. The third and forth jump in speed also lasts for five seconds each. Once the axles speed are manipulated signal 6 will respond to it, by updating signal 7, as it is depicted in the figure 22 all the raises in speed are reflected concurrently by signal 7 (train speed). This can be more clear by zooming in at the scale of 50ms steps timeline in figure 23. It is divided into four different sub plots, green color line represents axle speed, red represents maximum axle speed and blue represent train speed. In top left sub plot, the raise of speed happens at 250ms which correspond to the train speed being updated at the same time. Similarly top right sub plot shows axle 2 and train speed raise time which is the same. Last but not least both bottom sub plots prov the same finding.
The plot in second zone suggests that there is some delay in the response time of train speed being updated, and this is due to the high CPU usage simulation which was forced on the targeted system. Figure 24 shows a zoomed version of this zone while speed is getting updated. Like zone 1 the speed gets updated whenever there is a change in axles' speed calculation, it starts from 0kmh and reaches 45kmh at the end of cycle. However it is visible from the graph that the updating throttles at few points. At 12:18:45 and again at 12:18:53. These are the places that there is a chance that speed calculation has invalid (it means it has the previously calculated speed rather the newest calculation which we consider invalid) value for a specific period of time. This phenomenon is more clear in figure 25.

Figure 22: Zone 1 signals zoomed

Figure 23: Zone 1 signals split-ted and zoomed
Figure 24: Zone 2 signals zoomed

Figure 25: Zone 2 signals split-ted and zoomed

The green line indicates the axles' value, the red line shows the maximum calculated axle speed and blue line represents the train speed. It is divided into four sub plots; top left shows the speed increase from 0kmh to 13kmh without any issue. Top right plot shows that axle 2's speed is raising toward 22kmh at exactly 12:18:44.850, surprisingly the train's speed has already started dropping speed 50ms ago and only picks up after being stimulated again by axle 2. This effect shows that during this 50ms (12:18:44.800 to 12:18:44.850) the displayed speed is in invalid mode. This happens because of the injected high CPU load into the system and affecting the schedule to distribute time slots properly or simply CPU is too busy to service the tasks. The implementation details such as how OS's scheduler divides time or how efficient is speed calculation is out of scope of this research, and we are only interested of finding such scenarios that time violation might
happen or a value is invalid based on predefined specifications. The bottom sub plots in figure 25 indicates that response time goes back to normal at these points. Figure 26 depicts zone 3. This zone suffers from the same issue as zone 2 did. The CPU load in the simulated environment varies dramatically during this zone which is affecting it in few places when the speed is being calculated. Figure 27 pin point this issue. Top left sub plot shows normal response time whereas top right sub plot hints a 50ms for updating the speed. Bottom sub plots acts normal. This happens for the second and third stimulation in zone 3 and each time it is the second axle that comes down with this delay. We can extract that depending on the work load in the system, there are scenarios that the calculated speed is at least invalid for a minimum of 50ms.

Figure 26: Zone 3 signals zoomed

Figure 27: Zone 3 signals split-ted and zoomed
7 Delimitation

This chapter is divided into two main sections. Firstly, a discussion about this thesis will be given which narrates the process of doing the work. Secondly, the limitation that we have faced throughout this work will be discussed and the threats to validity will be mentioned in this section as well.

7.1 Discussion

Amid discussion with Bombardier Transportation, one of the highlighted area was the amount of time being spent on testing and preparing test environment for different configurations. And since it is a safety critical system it indeed requires tremendous number of safety checks before acquiring safety certificates, which in turn compel Bombardier to put in place a continuous effort to make testing and testing process more efficient and reliable. The current process used in Bombardier for testing covers specific test cases and user requirements (a must for safety critical systems), however the benefits of a passive way to constantly monitors running system and checks it for conformance is obvious. The methods and techniques proposed in this thesis can bridge the gap between requirement specification and implementation (specifically for PLC codes). Furthermore the results gathered for this thesis were able to detect scenarios that calculated values can be invalid for a specific amount of time (the previous value will be used), which can be catastrophic for safety critical systems. However it should be stated that the results and analysis were obtained using a simulated environment and stimuli. Albeit the same process can be used for real environment and hardware. In addition the proof of concept tool provided with this thesis can connect to the same tooling system for evaluating the data and displaying violations. Conducting an industrial case study will provide a great exposure to get familiarize with the contemporary technologies and practical processes and techniques to tackle and solve real world problems. However it offers its own challenges and it is inevitable, for instance few challenges that we have faced during this thesis and hindered development process were caused on account of not being very familiarized with the tooling and perhaps not a concrete communication channel to find necessary documentations, tools and resources. This can be justified owing to the lack of manpower and heavy loads of work on the company. Nevertheless it cannot undermine all the benefits and exposure that were achieved by working with real world data and system, and for that we thank Bombardier Transportation.

7.2 Limitation

Whilst doing any sort of development or research there are always obstacles that can hinder the process. Furthermore there are always challenges for validating and analyzing the obtained results. In software engineering there are methods and techniques that can be used to categorize these limitations and validity threats. Wohlin et al examine four main types of validity threats including: conclusion validity, internal validity, construct validity and external validity. Each of which contributes their own characteristics to the general and total limitation that can plague a system. However Per Runeson and Martin Höst introduce the same threats to validity with minor changes to make it more suitable for flexible design studies, in their version of threats to validity conclusion validity phrase is replaced by reliability. In the following sub sections each threat is explained and its relation to this particular case study is highlighted using [40].

7.2.1 Construct validity

Construct validity concerns to what extend the research idea is related to the operational measures that were studied, in other world if the interpretation of research questions is according to the study. The data gathered for this study were extracted using application level API and a limitation of accessing OS level API were present. OS level API for measuring accurate information such as timing, cyclic values and so forth are necessary. The operating system that runs the TCMS boxes are based on a Real Time Operating System (RTOS) named VxWorks, and there is a shell developed by Bombardier on top of that, so any attempts that requires tapping into the OS, needs

\footnote{VxWorks is a proprietary software which is developed by Wind River and used heavily in medical devices, industrial equipment, robotics, energy, transportation, network infrastructure, automotive, and consumer electron-}
to be addressed by third part and then the shell developers. The results obtained for this thesis were generated on a Windows machine which is NOT a realtime operating system, so the timing and clock cycle are affected by many processes that are being served by the OS.

7.2.2 Internal validity

Internal validity get details on how confident we can be that the outcome was in fact caused by the stimulation. For example when the study focuses if a factor affects the results without being aware that a third factor also affects the outcome. The results which were used in this case study were obtained in a simulated environment (SoftTCMS), all the MCU cycles, network IOs, memory locations and internal interaction were also simulated. This will affect time critical calculation as there is no guarantee of what aspect of simulated environment is effecting the system at any given time.

7.2.3 External validity

External validity answers the question weather the obtained results can be generalized outside the scope of the study. This case study was specifically conducted for PLC environment in Bombardier site, and most of the tooling were proprietary software developed and maintained by the company which yields that the results are specific to this case study hence cannot be popularized for general purposes. However there are phases that can be utilized outside the scope of this thesis, for instance formal specification phase and how they were saved and stored can be extended and potentially used outside this case study albeit similar domain (PLC).

7.2.4 Reliability

Reliability concerns about how dependent is the obtained results and analysis to a specific researcher. For instance if another researcher conduct the same study, will the produced result be the same. This threat to validity happens under circumstances that for example there is no clear way of collecting data, producing stimuli and such things. The process of collecting data in this thesis is straightforward and specification of requirements also is well outlined. One of the key factors of gathering reliable results is to have consistent and effective stimuli. This is the utmost important factor and can affects the results dramatically if administered in a different way. Other small notes can be software versions used, network settings and target system environment.
8 Conclusion

This chapter concludes the case study and summarizes the final thoughts. First section provides a summary from the start of the thesis to the outcome of the project, and in the second section some future work and enhancement will be introduced that outlines the steps that assist reducing limitations.

8.1 Summary

This thesis has explored the possibility of exercising runtime verification in an industrial environment that employs PLC systems. The idea of mitigating and detecting anomalies during runtime of a system which otherwise would be easy to miss were touched and a proof of concept tool (MVTerm) were introduced to alert the user for any deviation and misbehaviour. The runtime verification process used in this thesis were organized from an introduction to different approaches that are possible for embedded environment to the easiest way of displaying information to the end user. During the thesis different methods and techniques of formal specification were discussed as well, after careful consideration and comparisons, PLCSpecif were used as the main method of formal specification language for this project. The usage of PLCSpecif were reasoned on account of being specifically useful for PLC environment, coding style and theories behind it. In the end the target system were stimulated and enough samples were gathered, these sample data were used in the RV process that were introduced in this thesis checking the conformance of the system. Albeit there are rooms for improvement of this project and some of the limitations that were mentioned in section 7.2 can be overcome and soften in the future work.

8.2 Future work

Future work includes a more solid and robust validation method as well as a well established communication with the PLCSpecif vendors to attain a copy for generation of real code. Furthermore the test and analysis conducted for this work has been done in the application level of the system, a more solid and systematic communication with OS level that provides realtime scheduling values such as deadline, priority and switching between tasks will enhance the quality of results as well as accuracy. The current tooling provided with this thesis (MVTerm) is communicating with a version of system that does not provide OS level calls to retrieve details about execution hence as a future work both side of communication need to be updated to address this issue. Defining and investigating more cases of stress and manipulation of the system both in component and system level will generate more raw data for enhancing the analysis. Another area to improve in future work can be the notion of time, most RV system utilize some sort of logical clocks or temporal logic (specially Linear Time temporal Logic (LTL)) for considering order and timeline in the system.
References


A MVTerm Interaction

A UI: MVTerm

Figure 28: MVTerm main UI
B UI: Setting

Figure 29: MVTerm main UI
C  UI: Plotting and warning

![MVTerm Plotting and evaluation area](image)

Figure 30: MVTerm Plotting and evaluation area
B DCUTerm Interaction

A Signal analysis

Figure 31: DCUTerm signals' plot for axles
B Axle manipulation script

; Disconnects everything
FLC *

; Enable execution of speed calculation module
  o DBC_SV_TrnSpdCalc_1_Enable /1/.
  o SpdCalc_T2_Enable /1/.
  o T2_Enable /1/.

; Attach signals to fast logger
FLC DBC_SV_TrnSpdCalc_1_DBC_SV_Xv_TrnSpd
FLC DBC_SV_TrnSpdCalc_1_DBC_SV_Vv_TrnSpd
FLC DBC_SV_TrnSpdCalc_1_W10_Xv_TrnSpd
FLC DIAG_DV_X_CpuLdAvg
FLC DBC_SV_TrnSpdCalc_1_DBC_SV_Xv_BcuSpd_1
FLC DBC_SV_Xv_BcuSpd_2
FLC DBC_SV_Xv_BcuSpd_3
FLC DBC_SV_Xv_BcuSpd_4

; Update signals
  o TC_BI1BCU12_Vv_Ax1 /1/.
  o DBC_RV_Xd_WhDm_1_1000/.
  o TC_BI1BCU12_Vv_Ax1_5000/.
  o TC_BI4BCU12_Vv_Ax1_1/.
  o DBC_RV_Xd_WhDm_2_1000/.
  o TC_BI1BCU12_Xv_Ax2_5000/.
  o TC_BI4BCU12_Vv_Ax2_1/.
  o DBC_RV_Xd_WhDm_15_1000/.
  o TC_BI4BCU12_Xv_Ax2_5000/.
  o TC_BI4BCU12_Vv_Ax1_1/.
  o DBC_RV_Xd_WhDm_16_1000/.
  o TC_BI4BCU12_Xv_Ax1_5000/.

; Start fast logging on T2 (task2)
FLS T2
C IoTool automation script

```xml
<DocumentElement>
  <ToXml>
    <Signal x0020_Name="DBC_SV_TrnSpdCalc.1. Enable"/>
    <Override x0020_Value>1</Override>
    <Override x0020_Time x0022=",x0028,1,x002F,10s,x0029;">300</Override>
    <StartTime x0020_Offset x0022=",x0028,1,x002F,10s,x0029;">0</StartTime>
  </ToXml>
  <ToXml>
    <Signal x0020_Name="Enable_DBC_S"/>
    <Override x0020_Value>1</Override>
    <Override x0020_Time x0022=",x0028,1,x002F,10s,x0029;">300</Override>
    <StartTime x0020_Offset x0022=",x0028,1,x002F,10s,x0029;">0</StartTime>
  </ToXml>
  <ToXml>
    <Signal x0020_Name="SpdCalc_T2. Enable"/>
    <Override x0020_Value>1</Override>
    <Override x0020_Time x0022=",x0028,1,x002F,10s,x0029;">300</Override>
    <StartTime x0020_Offset x0022=",x0028,1,x002F,10s,x0029;">0</StartTime>
  </ToXml>
  <ToXml>
    <Signal x0020_Name="TC_BI1BCU12_Vv_Ax1"/>
    <Override x0020_Value>1</Override>
    <Override x0020_Time x0022=",x0028,1,x002F,10s,x0029;">50</Override>
    <StartTime x0020_Offset x0022=",x0028,1,x002F,10s,x0029;">50</StartTime>
  </ToXml>
  <ToXml>
    <Signal x0020_Name="DBC_RV_Xd_WhDm.1c"/>
    <Override x0020_Value>1148846080</Override>
    <Override x0020_Time x0022=",x0028,1,x002F,10s,x0029;">50</Override>
    <StartTime x0020_Offset x0022=",x0028,1,x002F,10s,x0029;">50</StartTime>
  </ToXml>
  <ToXml>
    <Signal x0020_Name="TC_BI1BCU12_Xv_Ax1"/>
    <Override x0020_Value>300</Override>
    <Override x0020_Time x0022=",x0028,1,x002F,10s,x0029;">50</Override>
    <StartTime x0020_Offset x0022=",x0028,1,x002F,10s,x0029;">50</StartTime>
  </ToXml>
</DocumentElement>
```