Computer Science
Master Thesis

On-board Diagnostics Framework

Author:
Patrik Olsson

Supervisor/Examiner:
Jan Carlson

Industrial supervisor:
Johan Persson

Västerås, June 25, 2012

Mälardalen University
School of Innovation, Design and Engineering
Abstract

For as long as complex machines have been around there has also been a need for accurate diagnostic and prognostic capabilities. Over the last decades there have been many attempts to develop and implement systems with various degrees of diagnostic and prognostic abilities, some successful some not. This master thesis presents a software architecture that is used to implement an on-board diagnostic framework capable of advanced diagnostics and prognostics in industrial vehicles. The presented architecture consists of a set of modules that focuses on performance, scalability and simplicity that fulfils the demands of current and upcoming diagnostic and prognostic techniques.
## Contents

1 Introduction ......................................................... 7  
1.1 Problem formulation ........................................... 7  
1.2 Industrial context and limitations .............................. 7  
1.3 Contributions .................................................... 8  
1.4 Organization ..................................................... 9  

2 Background study .................................................. 10  
2.1 CrossControl software application platform .................. 10  
2.1.1 User interaction ............................................... 10  
2.1.2 Vehicle control ................................................ 10  
2.1.3 Mobile connectivity .......................................... 11  
2.1.4 Onboard diagnostics ........................................... 11  
2.1.5 Data engine .................................................. 11  
2.2 Hardware platforms ............................................. 11  
2.2.1 XM family ................................................... 11  
2.2.2 XA family ................................................... 12  
2.3 Diagnostic runtime engine ...................................... 13  
2.3.1 Logical layout ............................................... 14  
2.3.2 Configuration ................................................ 15  
2.3.3 Runtime ..................................................... 16  

3 Theoretical study .................................................. 18  
3.1 Data acquisition ................................................ 18  
3.2 Data processing ................................................ 18  
3.2.1 Data preprocessing ........................................... 18  
3.2.2 Signal processing ............................................ 19  
3.2.3 Value type data analysis .................................... 19  
3.2.4 Combined data analysis ..................................... 19  
3.3 Fault diagnosis .................................................. 20  
3.3.1 ISO 13374-1 .................................................. 20  
3.4 Fault prognosis .................................................. 21  
3.5 Diagnosis and prognosis techniques ............................ 21  
3.5.1 Commonly used data-driven techniques ...................... 21  

4 Suggested solution ................................................ 23  
4.1 Data engine — General .......................................... 23  
4.1.1 Communication ............................................... 23  
4.2 Data engine — Server .......................................... 23  
4.2.1 Data types ................................................... 24  
4.2.2 Data storage ................................................ 24  
4.2.3 Data engine server software architecture .................. 25  
4.3 Data engine — Client .......................................... 25  
4.3.1 Application programming interface ........................ 26  
4.3.2 Data engine client software architecture .................. 28
4.4 Diagnostic engine ......................................................... 28
  4.4.1 Individual scheduling ............................................. 29
  4.4.2 Reduced number of action blocks ................................. 29
  4.4.3 Standardized diagnostic blocks .................................. 29
  4.4.4 Chained diagnostic blocks ....................................... 29
  4.4.5 Diagnostic engine software architecture ...................... 30
4.5 Configuration environment ........................................... 31
  4.5.1 Integrated development environment plugin .................... 31
  4.5.2 Configuration environment software architecture ........... 34
  4.5.3 Diagnostic block development using MATLAB .................. 35
4.6 Prototype implementation ............................................. 35
4.7 Solution evaluation .................................................. 35
4.8 Alternative solutions ................................................ 36

5 Conclusions .................................................................. 37
  5.1 Summary ................................................................. 37
  5.2 Future work ............................................................. 37
List of Figures

1. CrossControl solution portfolio ........................................ 8
2. SAP concept .................................................................... 10
3. XM family ....................................................................... 12
4. XA family ....................................................................... 12
5. Overview of DRE concept .................................................. 13
6. DRE logical layout ............................................................ 14
7. DRE call graph .................................................................. 17
8. ISO 13374-1 ...................................................................... 20
9. Data engine overview .......................................................... 23
10. Data engine server architecture ............................................ 25
11. Data engine client architecture ............................................. 29
12. Diagnostic chains .............................................................. 30
13. Diagnostic engine architecture ............................................ 31
14. Block overview ............................................................... 32
15. Block creator dialog .......................................................... 33
16. Chain layout view ............................................................. 33
17. Chain debug view ............................................................. 34
18. Configuration plugin architecture ......................................... 35

List of Tables

1. Proposed data types ............................................................ 24
2. Standardized diagnostic blocks ........................................... 30
Abbreviations

API Application programming interface
CBM Condition based maintenance
DABE Diagnostic application builder environment
DRE Diagnostic runtime engine
GUI Graphical user interface
IDE Integrated development environment
ISO International standards organization
OBD On-board diagnostics
OEM Original equipment manufacturer
PHM Prognostic and health management
SAP Software application platform
Acknowledgements

I would like to thank all persons involved at CrossControl in Västerås for the opportunity to write my master thesis there. I’ve had a great time and will truly miss you all when I leave.

Especially I would like to thank Johan Persson at CrossControl for all the help and support you’ve given me during my stay at the office and my supervisor at Mälardalen University, Jan Carlson for excellent feedback on the report.

I would also like to thank Carl-Magnus Moon, Anders Svedberg and Jörgen Martinsson at CrossControl for great input regarding all the technical questions that have come up during my work with this master thesis.
1 Introduction

For as long as complex machines have been around there has also been a need for accurate diagnostic and prognostic capabilities. Over the last decades there have been many attempts to develop and implement systems with various degrees of diagnostic and prognostic abilities, some successful and some not. Recently, requirements have begun to arise primarily within the automotive, aerospace and defence industries regarding advanced prognostic and health management (PHM) capabilities implemented in almost all new development projects. The main motivation for these requirements is to enable the use of advanced condition based maintenance (CBM) logistics programs, where machine standstills are kept to a minimum. In industrial vehicles, which this thesis will focus on, the benefits of using CBM are to increase the reliability and availability of the vehicle and thereby also the revenue it generates. CBM enables this by allowing companies to replace many of the old pre-scheduled maintenance sessions which are very costly and labour intensive with small on-demand maintenance sessions to prevent a fault from occurring when the PHM system has detected deterioration in performance somewhere within the vehicle. This both increases the degree of utilization of the vehicle and keeps the running costs associated with the vehicle to a minimum.

The effectiveness of a CBM program is directly linked to the accuracy of the PHM system. Both the success rate of fault detection as well as fault isolation keeps improving through advancements in research around the subject and rapid development of new, faster hardware that can be used for running the systems. This has led to even more ambitious requirements being placed on new PHM systems which really challenge the developers. Prognosis is considered to be the most challenging part of a PHM system. It is also the part that if executed correctly will provide the greatest improvement in reduction of operation and support costs and total life-cycle cost of the vehicle. This is most often the factor that drives the development of new PHM systems forward.

1.1 Problem formulation

The purpose of this master thesis is to define a modular architecture for an on-board diagnostics framework consisting of a data repository, a diagnostic engine and a configuration environment for the diagnostic engine. Low CPU utilization, modularity, a simple interface and a short response time are key features that need to be included during the development process. Support should be given for all common techniques used for vehicle diagnostics as well as vehicle prognostics. If possible, backward compatibility with existing diagnostic systems is a benefit.

1.2 Industrial context and limitations

The work has been carried out at CrossControl, a company that develops and manufactures advanced control systems and human machine interfaces for leading industrial vehicle original equipment manufacturers (OEM) that manufac-
ture vehicles that operate in rough environments. They also provide a broad solution portfolio to that addresses all aspects of the life of a product or system. As seen in Figure 1 a part of this portfolio is called premium user interaction. If requested by a customer, CrossControl can provide their software application platform (SAP) as a part of combined hardware and software solution; this gives the customer access to advanced software that is specifically developed for their hardware. As a part of this SAP, CrossControl is looking into the possibility of providing an on-board diagnostic framework.

Figure 1: CrossControl solution portfolio

The goal of this master thesis is to find a suitable design for an on-board diagnostic framework that can act as a standalone application within CrossControl’s SAP. The suggested architecture must conform to the SAP concept and be able to execute on hardware with limited resources.

The suggested solution must be able to run in Ubuntu on hardware that is a member of CrossControl’s XA and XM family of products. There is no requirement that it should work on other operating systems. The implemented prototype doesn’t have to be feature complete, but it should be complete enough to prove the concept of the suggested architecture. The design of the suggested architecture itself should, however, be so complete that it can be implemented without significant changes.

1.3 Contributions

This master thesis has contributed to solving the problem described in Section 1.1 by suggesting a modular architecture for a data repository, a diagnostic engine and the configuration environment for the diagnostic engine that together forms the basis of a on-board diagnostic framework. The suggested architecture fulfils all the requirements regarding support for common techniques as well as providing high-performance, a simple yet powerful interface and scalability to support further extensions of the architecture.
1.4 Organization

The first chapter gives a short background to the problem that has been resolved. Chapter two presents CrossControls SAP, the targeted hardware and give a brief description of a diagnostic system currently developed by Cross-Control and in use at one of CrossControls customers. A technology survey looking at the different steps of the diagnostic and prognostic processes suitable for on-board use will be given in chapter three. Chapter four present the suggested architecture of the on-board diagnostic framework. Finally chapter five discusses the conclusions that can be drawn from the project and also gives some suggestions for future work that can be done regarding the subject.

The report is best read unabridged in chronological order. If you just want a general idea of the problem and the solution to it, it is possible to read only chapter 1 and 5.
2 Background study

This section gives a brief introduction to the SAP offered by CrossControl as a part of the Premium User Interaction concept and the different hardware platforms that is targeted by the SAP. It also gives a brief overview of a diagnostic system called diagnostic runtime engine (DRE) currently developed by CrossControl for one of its customers.

2.1 CrossControl software application platform

CrossControl provides a wide range of software products to meet the demands from their customers in the industrial vehicle automation industry. These products are based on industrial standards and are developed with a clear separation between the different applications. CrossControl is currently trying to package these products into separate applications that can be connected to a data repository as shown in Figure 2. This gives OEMs freedom of choice when making decisions regarding what applications to use off the shelf and what they need to develop them-self. A wide variety of operating systems are supported and the targeted hardware platforms are the XM and XA families presented in Section 2.2.

![Figure 2: SAP concept](image)

2.1.1 User interaction

The user interaction part gives the customer access to state of the art graphical user interface components which enables them to develop efficient machine interaction interfaces. In the end this makes the handling of machine interaction very easy and intuitive for the machine operator when he is given relevant information for different operating situations.

2.1.2 Vehicle control

The vehicle control part provides the fundamentals for building control applications. It comes with a set of standard hardware interaction interfaces. This enables the developer to develop the control system in a standard development environment that can be selected to fit the preferences and needs of the control application.
2.1.3 Mobile connectivity

The mobile connectivity part provides access to the WLAN, GPRS, GPS and Bluetooth interfaces that are available on the hardware platforms provided by CrossControl. This wide variety of wireless protocols turns the application into a telematics gateway between the on-board control system and a backoffice fleet management system, which makes it possible to e.g. send the exact location of a machine retrieved via GPS to a back-office system through a GPRS connection or let a service engineer download a large amount of diagnostic data to a laptop using a WLAN connection.

2.1.4 Onboard diagnostics

The on-board diagnostic part is responsible for performing diagnostics and prognostics on run-time data signals and notify the operator and/or a back-office system if something is about to break or already is broken. This enables the developers to add diagnostics to the machine without interfering with its control system. The on-board diagnostics concept is thoroughly described in Section 2.3.

2.1.5 Data engine

All communication between the SAP applications internally and with the hardware is going through a data engine. The data engine acts as a data repository where data that an application wants to share is registered, this data can then be retrieved by other applications that need to have access to the data.

2.2 Hardware platforms

The CrossControl hardware product range targets control solutions for industrial machines and vehicles. The portfolio represents robustness, reliability, industrial design and user friendliness; this has been accomplished by working closely with OEM manufacturers from the start. Display computers, main controllers, I/O controllers and infrastructure components are the different type of products that are offered.

2.2.1 XM family

The XM-family of products manufactured and sold by CrossControl targets applications where the focus lies on providing a premium user experience combined with high-performance system management and advanced communication interfaces.

CrossPilot XM  CrossPilot XM is an Intel Atom based high performance display computer and system controller in a very robust format for use in industrial machines, vehicles and vessels. The large amount of connection interfaces both
wired and wireless, makes it suitable for integration of various vehicle sub systems and external support and management systems with a human machine interface to support the operator in order to achieve higher machine utilization. It comes with a 10.4, 12.1 or 15 inch touch screen TFT display [6].

**CrossCore XM** CrossCore XM is an Intel Atom based high performance controller for industrial vehicle and machine applications. Due to a wide variety of interfaces, both wired and wireless and its high performance it’s suitable to use as a master in a distributed control network architectures or as a high performance standalone system controller [4].

### 2.2.2 XA family

The XA-family of products manufactured and sold by CrossControl targets applications where the cost needs to be kept at a minimum while still providing the customer with a premium user experience.

**CrossPilot XA** CrossPilot XA is an ARM based a display computer and system controller in a slim and rugged format. It’s suitable for use as e.g. a graphical user interface, service terminal for onboard control systems or as a video monitor connected to cameras around a machine or vehicle. It comes with either a 7 or 10.4 inch touch screen TFT display that offers good readability in all possible conditions [5].
CrossCore XA  CrossCore XA is an ARM based controller for industrial vehicle and machine applications. Due to a wide variety of interfaces, both wired and wireless, it’s both a controller and communications gateway suitable for use as slave in distributed control systems or as a standalone system controller [3].

2.3 Diagnostic runtime engine

CrossControl have implemented a diagnostic system called DRE for one of its customers, that is able to conduct advanced diagnostics of the system that it is used in. As shown in [1] the DRE consists of three parts, the diagnostic runtime, a database that contains diagnostic data produced by the diagnostic runtime and a system configuration tool. Communication between the diagnostic runtime and the control system that is being diagnosed is conducted through a block of shared memory that both the DRE and the control system have read and write access to, this enables the DRE to notify the control system if an error has been detected.

Even though it is possible to run the DRE as a separate system, it is often coupled with a graphical user interface (GUI) that presents the data that is stored in the database by the DRE. All communication between the DRE and the GUI goes through the database so there is no direct communication between them. This makes it possible to always treat the DRE as a separate system [2]. Figure 5 shows a system where one of the main parts is the DRE.

![Figure 5: Overview of DRE concept](image)

Three types of users are assumed to be using the system:

Service engineer  The service engineer is the engineer that receives notifications through email or SMS from the diagnostic system. The service engineer is also capable of monitoring the diagnostic data generated by the system through a web browser.
System engineer The system engineer is the engineer that is responsible for creating and developing the diagnostics blocks used in the system.

Application engineer The application engineer is the engineer responsible for setting up and configuring the system before delivery to the customer. Should the need arise this can also be done in the field after the delivery of the system.

2.3.1 Logical layout

The DRE is divided into four separate layers, a communication layer, a diagnostic layer, an action layer and a system layer. Communication between these layers is conducted through common value objects. These objects are always set up by the lower layer. Outside of these four layers lies the runtime scheduler that is responsible for executing the four layers in the correct order, an error manager responsible for error handling both during startup and execution and a configuration parser responsible for setting up the system based on the configuration file produced by the configuration tool. A graphical illustration of the logical layout can be seen in Figure 6.

![Diagram](attachment:image.png)

Figure 6: DRE logical layout

Communication layer The communication layer provides an interface for the diagnostic layer independent of what type of data source is used to retrieve the signals.

Diagnostic layer The diagnostic layer is where all the logic for the diagnosis process is implemented. Internally the layer is divided into several smaller
parts called diagnostics blocks. A framework for connecting the incoming signals to the diagnostic blocks and connecting the diagnostics blocks to action blocks also resides in this layer.

**Action layer** The action layer manages a number of action blocks listen in Section 2.3.2 that each has a specific action to perform. The application engineer is responsible for connecting a specific output from a diagnostics block to an action block. This enables the system to perform a predefined action when a diagnosis is made.

**System layer** The system layer contains modules that are used to perform actions depending on the output from the action layer.

**Error manager** The error manager is responsible for error handling within the system.

**Configuration parser** The configuration parser is responsible for reading and storing the information contained in the XML-file produced by the configuration tool.

### 2.3.2 Configuration

Configuration of the diagnostic system is done with a tool called Diagnostic Application Builder Environment (DABE) which is a graphical editor for configuration of the connections between diagnostic blocks and action blocks. The user interface is similar to the one used in National Instruments system design software LabView [14] where you connect blocks to each other with signals to perform certain tasks. When the configuration is done in DABE, an .xml file containing the configuration instructions is generated.

**Diagnostic blocks** Each diagnostic block is a function contained within a .dll file that consists of a number of inputs, outputs and parameters. The logic inside the diagnostic block decides if the values of the inputs are normal. If not, an output is generated that activates any action blocks connected to the diagnostic block.

The DRE doesn’t come with any predefined diagnostic blocks, these have to be developed. Diagnostic blocks are developed using C++ and is compiled to a file that is read by the DRE during the configuration phase.

**Action blocks** The DRE comes with the following predefined action blocks with corresponding system managers in the system layer:

**Create event** When the create event action block is executed an event I written to the database together with a parameter indicating whether it is a normal event, a warning or an alarm.

**Create trend** When the create trend action block is executed the last 100 values of a signal is stored in the database.
Create statistics  When the create statistics action block is executed, the current value of a signal is stored in the database.

Send e-mail and send SMS  When either send e-mail or send SMS is executed an e-mail or SMS is sent to a predefined address/number in the corresponding system manager.

Store black box  When the store black box action block is executed the latest 1024 values of a signal is written to a file on the hard drive of the computer where DRE is running.

Set signal in control system  When the set signal in control system action block is executed, the value of a signal within the control system is changed by the DRE.

2.3.3 Runtime

When the system starts, the runtime scheduler tells the configuration parser to read the configuration file and determine which diagnostic blocks and action blocks that should be instantiated and how these blocks should be connected. When the configuration is done the four layers are initialized. The system layer is initialized in four different threads that run concurrently with the main thread but at a lower priority. These four threads handle actions that have been queued by the action blocks. The queued actions can be database writings, back office communication, email and SMS messaging and black box writing.

During a normal execution cycle which occurs approximately every 10th millisecond, the runtime scheduler handles the invocation of each layer according to the sequence diagram in Figure 7. The communication layer has got two execute functions, one that reads signal values from the control system and one that writes signal values to the control system. The system layer has no main function; instead the four system layer threads do their work whenever they are queued from within an action block which has pointers to their corresponding system manager.

There is no synchronization between the control system and the diagnostic system other than that they both run at a 10 millisecond interval. Due to the lack of real synchronization the signal values retrieved from the control system might not be from the same execution cycle. This is not a problem since the control system receives its signals from sensors distributed throughout the system and there is no guarantee that they all will update their values at the same time.
Figure 7: DRE call graph
3 Theoretical study

This section gives a brief overview of the current technologies for retrieving the input data to the diagnostic and prognostic process as well as giving a short introduction to different techniques used to perform diagnostic and prognostic analysis.

3.1 Data acquisition

The purpose of the data acquisition step is to gather and store interesting and useful data from the targeted vehicle that is being analysed. The data gathered can be categorized into two types, event data and condition monitoring data. Event data consists of information regarding what has happened to the vehicle, e.g. scheduled maintenance and what was done during the maintenance session or a breakdown and what the cause of the breakdown was and how it has been resolved. Event data usually requires that the maintenance personnel enters the information manually into the system and is therefore due to the human factor more likely to contain errors. The number of errors in manually entered data can be reduced if the event data is entered into the system through the maintenance information system used by the vehicle. Condition data on the other hand is data regarding the current state of the vehicle gathered by sensors. Examples of condition data is engine coolant temperature, tire pressure and electrical system voltage. Even though event data and condition data contains very different types of information, they are equally important for the end result of the diagnostic and prognostic process [7, 10].

3.2 Data processing

Data processing is used to extract useful data from incoming sensor values and user input. The process is divided into two steps, data preprocessing and data analysis. The preprocessing part only needs to be performed when the data enters the system. Analysis can on the other hand be performed on the same data several times [15]. The analysis part is divided into three different categories that works on different types of data:

Value type A single data value
Waveform type Data values collected over time
Multidimensional type Multidimensional data, typically images

3.2.1 Data preprocessing

Data acquired in the data acquisition process may contain errors, event data from human errors and condition data may suffer from errors originating from broken sensors. During preprocessing of incoming data the data is cleaned in order to increase the chances of error free data, this reduces the risk of running
into a garbage in, garbage out situation during further analysis of the data. Preprocessing of data is a very complex subject and errors are quite hard to detect, especially errors originating from sensors. Most systems today only checks if the data is within a valid value range [9].

3.2.2 Signal processing

The goal of signal processing is to breakdown a signal, analogue or digital, into smaller units to make the characteristics of it more visible and easier to analyse [10].

**Time-domain analysis** The purpose of time-domain analysis is to analyse how the value of a signal changes with respect to time. Useful data that often is extracted during time-domain analysis is divided into two categories. Characteristic data like mean value, peak value and peak to peak interval. And higher order data like the root mean square and kurtosis of a signal.

**Frequency-domain analysis** The purpose of frequency-domain analysis is to analyse and isolate features of a signal with respect to frequency. Several methods exists for doing this, these methods are used to extract different types of features from a signal, the ones that are most commonly used are Fourier series, Fourier transform and Laplace transformation.

**Time-frequency analysis** When a fault occurs in a mechanical system a signal waveform that is stationary often turns into a non-stationary waveform. The non-stationary waveforms can’t be handled using either time-domain analysis or frequency-domain analysis, therefore time-frequency is used instead. Time-frequency analysis combines time-domain analysis with frequency-domain analysis to analyse the signal in both dimensions simultaneously. The most common method for conducting time-frequency analysis is Short-time Fourier Transforms.

3.2.3 Value type data analysis

Value type data consists of raw data collected from the system as well as the results obtained from signal processing. Compared to waveforms and multidimensional data value type data is quite simple. It is when a large of number of different values needs to be analysed and correlated to each other that it gets complex. Regression analysis and time series models are techniques that often are used to analyse value type data.

3.2.4 Combined data analysis

Since both event-data and condition-data is available in most PHM systems. They usually can be analysed together to provide a more accurate result of the current state of the analysed system. In order to enable this combined analysis
a accurate enough mathematical model that describes the mechanisms of the fault and failures that can occur in the system that is being analysed.

3.3 Fault diagnosis

Fault diagnosis has been the subject of several research programs in several types of industries during the past decade. All of them have aimed at developing methodologies that can be used to detect faults or deviations indicating faults in unmonitored parts of the system. The result of this research has so far been several techniques for detecting faults as well as a standard proposed by International Standards Organization (ISO).

3.3.1 ISO 13374-1

ISO 13374-1 is a standard that provides a conceptual framework that indicates how the processing model of a CBM/PHM system should be implemented [20]. It covers all steps from data acquisition to maintenance advice generation. The processing model of ISO 13374-1 is shown in Figure 8. Several large companies mainly in the aircraft industry, like Lockheed, Boeing and General Electrics have tried to implement the general concept described in ISO 13374-1 with varied results. As a compliment to ISO 13374-1, two more standards have been published, ISO 13374-2 that handles data processing and ISO 13374-3 that handles data communication. A new ISO standard ISO 13379-1 will be published in 2012, it discusses the different diagnostic techniques that can be used in diagnostic systems.

![Figure 8: ISO 13374-1](image-url)
3.4 Fault prognosis

During fault prognosis the remaining useful life of a component is predicted. This needs to be done accurately and precisely which has turned out to be the Achilles heel of CBM and PHM systems. The core of the problem lies within the fact that all vital parameters of the component that the prognosis is conducted for are most likely not monitored. This leads to a uncertainty regarding their actual state and assumptions have to be made [17].

3.5 Diagnosis and prognosis techniques

There are two main categories of techniques for performing diagnostics and prognostics, model-based and data-driven [10]. Model-based techniques rely mainly on an accurate mathematical model of the system that is being diagnosed and is very useful for detecting unanticipated faults. It works by comparing the current state of the system against a mathematical model and if large deviations are detected a fault can be assumed to have happened. Data-driven technique on the other hand works by comparing the current state of the machine to known fault models [21, 24], if the state matches one of the fault models a fault can be assumed to have happened. A third technique called probability based is used mainly for prognostics and relies on statistical methods. Of the three techniques mentioned data-driven techniques are the most common and produces the best results in terms of accuracy and efficiency for on-board diagnostics and prognostics. The other two methods need to much computing power to efficiently work in on-board systems and are better suited for off-board diagnostics in field terminals for technicians or back office systems. Three different data-driven techniques are suitable for use in a on-board diagnostic framework running on limited hardware, a brief description of these three techniques is given below [10].

3.5.1 Commonly used data-driven techniques

**Alarm bounds**  During diagnostics, alarm bounds techniques utilizes minimum and maximum values for a sensor signal. Should the sensor signal exceed or fall below the predefined values a fault can be determined to have happened. In the case of prognostics, the signal is monitored to see if it approaches one of the values rather than crossing it. This is the simplest and least resource demanding technique, and therefore also the most commonly used technique.

**Fuzzy logic**  Fuzzy logic is what is called many-valued logic, it doesn’t provide true or false as a answer like boolean logic does. Instead a degree of truth is provided [23]. For example, the statement, today is sunny, might be 100% true if there are no clouds, 80% true if there are a few clouds, 50% true if it’s hazy and 0% true if it rains all day. Fuzzy logic is especially useful for prognostics since deteriorating performance for a component easily can be detected by evaluating several parameters at once.
Case based reasoning  Case based reasoning relies on a database of states that has previously led to failure or deteriorated performance of a component in order to detect if the monitored component is headed the same way. Every time a new state that has lead to an error or deteriorated performance is detected it is added to the database. This makes the accuracy of the technique more accurate over time as more erroneous states are added to the database that the current state can be compared to.
4 Suggested solution

This section will present the solution to the problem described in Section 1. Since no data engine currently has been developed by CrossControl, a proposed architecture will be presented for both the data engine and the diagnostic framework. It should be noted that the proposed architecture fulfills the requirements of SAP but reduces the performance of the diagnostic process compared to the current DRE. Time critical diagnostics therefore have to be performed within the control system as proposed by [16]. And non critical diagnostics as well as all prognostics can be performed by the diagnostic framework.

4.1 Data engine — General

The data engine is supposed to act as the backbone of CrossControls SAP, therefore it needs to provide a general interface that can be used by all applications that want to connect to it. Such a interface is described in Section 4.3.1. It also needs to support a wide variety of data types, this master thesis describes the ones necessary for performing diagnostics and prognostics in Section 4.2.1. More data types might need to be added in the future. The general concept of the data engine is shown in Figure 9.

4.1.1 Communication

All communication between the data engine server and data engine clients connected to it will be conducted via TCP using messages formatted in a regular XML format. The decision to use the combination of TCP and XML is based on reliability. UDP was considered instead of TCP to increase performance but the risk of lost information that could affect the accuracy of the diagnostic system was too high. As an alternative to XML, Protocol Buffers [11] developed by Google was considered, but due to the fact that it isn’t supported by as many programming languages as XML is, it was rejected. If Protocol Buffers had been used, it would probably have increased the communication performance.

![Figure 9: Data engine overview](image)

4.2 Data engine — Server

The data engine server acts as the hub in the data engine. It handles the storage of all shared variables available in the system as well as the communication
between all the data engine clients that are connected to the data engine server.

4.2.1 Data types

Currently the DRE can only handle signed and unsigned integers as input parameters. This is a limitation that restricts the users from implementing diagnostic blocks that uses the more advanced techniques discussed in Section 3. Therefore the data-types described in Table 1 have been found to be necessary to support in the diagnostic framework. This conclusion is drawn by looking at the results of Section 3.1, 3.2 and 3.3 that shows the demands put on modern diagnostic and prognostic systems. The first ten data types listed in Table 1 will be used as in-parameters to the diagnostic blocks and the last two will be used as outputs from the action blocks.

4.2.2 Data storage

Two methods of storing variables have been selected both due to data engine performance and storage medium durability. Performance is gained if data requested by a client can be retrieved from a internal data structure instead from a relational database, which is quite time consuming, this means that a client will get a more rapid response to a request in most cases since only a few variables within a system needs to be permanent. The performance issue alone could have been resolved using a realtime database like Mimer SQL Real-Time [19] to store all variables as concluded by [13] and [22]. But this would still lead do decreased durability of the Compact Flash memory card used by the hardware described in Section 2.2 for permanent storage since they only can withstand a relatively limited amount of write cycles on each memory cell. The expected life of a memory card would only be a couple of months even when a normal amount of signals are handled using a traditional relational database.
4.2.3 Data engine server software architecture

A overview of the proposed data engine architecture is shown in Figure 10. The data engine is based around five main modules, their functionality is described below.

**Client connection** Operates in its own thread and handles communication with the connected clients, one client connection object per connected client is created.

**Message handler** Converts received XML formatted messages into actual data that can be handled by the server. Also works the other way around, it can convert data into XML formatted messages that can be transmitted to the clients.

**Data engine server** The main execution loop of the data engine server, it contains all logic that ties the different modules together.

**Permanent variable storage** A regular relational database that is used to store information that needs to be permanently stored. All data types listed in Table 1 has to be supported.

**Regular variable storage** A wrapper object around a set of lists that is used to store variables that do not need to be permanently stored. All data types listed in Table 1 has to be supported.

![Figure 10: Data engine server architecture](image)

4.3 Data engine — Client

The data engine client provides the application using it with a clean and easy to use interface for accessing the functionality provided by the data engine server.
4.3.1 Application programming interface

A simple application programming interface (API) is needed for the data engine client to get access to the functionality of the data engine server. An proposal of how such a API can be implemented is described below together with an example of how it is intended to be used. Corresponding methods should be available for all data types where it is applicable.

Public methods

int Connect(const string ip, const int port)

Summary: Connects the data engine client to a data engine server.

Parameter ip: The IP address of the data engine server.

Parameter port: The port of the data engine server.

Return value: A integer that shows if the operation was successful or not.

int Disconnect()

Summary: Disconnects the data engine client from the data engine server.

Return value: A integer that shows if the operation was successful or not.

int Add(const string name, const bool persistent, const int value)

Summary: Adds a variable to the data engine.

Parameter name: The name of the variable that is added.

Parameter persistent: Selects if the variable should be persistent or not.

Parameter value: The initial value of the variable that is added.

Return value: A integer that shows if the operation was successful or not.

int Set(const string name, const int value)

Summary: Sets the value of a variable in the data engine.

Parameter name: The name of the variable that is going to be set.

Parameter value: The value that the variable is going to be set to.

Return value: A integer that shows if the operation was successful or not.
int Get(const string name, int &value)  

Summary: Retrieves a variable from the local data engine buffer.

Parameter name: The name of the variable that is going to be retrieved.

Parameter value: A reference to a variable where the value of the retrieved variable should be stored.

Return value: A integer that shows if the operation was successful or not.

int Subscribe(const string name)  

Summary: Requests a subscription on a variable from the data engine.

Parameter name: The name of the variable that should be subscribed.

Return value: A integer that shows if the operation was successful or not.

Usage example

The following code shows a example of how the API for the data engine is supposed to be used. It demonstrates where a client connects to a server, adds a variable, then retrieves, display and update the variable 10 times before disconnecting from the server.

```cpp
1 int main()
2 {
3    int a = 0;
4    int b = 0;
5    DataRepositoryClient dataRepositoryClient;
6
7    dataRepositoryClient.Connect("127.0.0.1", 4242);
8    //Add variable a
9    dataRepositoryClient.Add("A", false, a);
10   //Subscribe to variable a
11   dataRepositoryClient.Subscribe("A");
12
13   while(1)
14   {
15     //Get variable a from the data engine
16     //store the result in variable b
17     dataRepositoryClient.Get("A", b);
18
19     //Print a and b to the console
20     cout << "A: " << a << endl;
21     cout << "B: " << b << endl;
```
// Exit when b is 10
if (b == 10) {
    break;
}

a ++;

// Set the new value of a
dataRepositoryClient.Set("A", a);

Sleep(1000);

dataRepositoryClient.Disconnect();

return 0;

4.3.2 Data engine client software architecture

A overview of the architecture of the client is shown in Figure 11. The client side of the data engine is based around four modules, their functionality is described below.

Server connection Operates in its own thread and handles communication with the server.

Message handler Converts received XML formatted messages into actual data that can be handled by the client. Also works the other way around, converting data into XML formatted messages that can be transmitted to the server.

Data engine client The main method of the data engine client. Handles all calls to the client API.

Variable buffer Variables requested from the server will be stored in a local variable buffer. Once a variable has been requested, any updates to the value of the variable will automatically be transmitted to the client from the server. All data types listed in Table 1 need to be supported.

4.4 Diagnostic engine

The diagnostic system already in use by CrossControl together with the improvements suggested by [12] makes a great platform for performing diagnostic
and prognostics on the limited hardware resources available, with only minor improvements. But in order to work with the SAP it needs to utilize the data engine client for retrieving and storing data instead of the system described in 2.3. Some small points of improvements are suggested below that make the system more future proof.

4.4.1 Individual scheduling

By making it possible to specify the periodicity at which a chain of diagnostic blocks executes, many CPU cycles can be saved due to the fact that normally only a few chains need to execute at a very high frequency. Another similar approach would be to execute a chain when the input values to the chain are updated so that multiple executions with identical input values can be avoided.

4.4.2 Reduced number of action blocks

In the current system only a subset of the predefined action blocks are actually used by the customers. Therefore a reduction is suggested. Create event, create trend, store black box and set signal in control system are the only ones that are used today. Of these only create event, create trend and store black box should be kept. Set signal in control system should for security reasons be replaced by some sort of messaging so that the diagnostic system can tell the control system to change a value instead of changing it without notifying the control system.

4.4.3 Standardized diagnostic blocks

The lack of predefined diagnostic blocks has been addressed in a previous master thesis [12] done at CrossControl. In order for standardized diagnostics blocks to be useful the system has to be changed so that it can use several layers of diagnostic blocks before the action block layer as described in Section 4.4.4. The four types of blocks described in Table 2 have been suggested by [12] as the standard blocks that should be provided with the configuration tool.

4.4.4 Chained diagnostic blocks

As concluded in [12] the blocks mentioned above are rather useless with the current way of setting up diagnostic chains, where only one diagnostic block can precede an action block as shown in Figure 12(a). A better way would
<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test functions</td>
<td>Tests whether a signal behaves as expected or not</td>
</tr>
<tr>
<td>Observers</td>
<td>Performs analytic operations on a signal</td>
</tr>
<tr>
<td>Filters</td>
<td>Signal filters</td>
</tr>
<tr>
<td>Boolean functions</td>
<td>Regular boolean functions</td>
</tr>
</tbody>
</table>

Table 2: Standardized diagnostic blocks

be to allow several diagnostic blocks to be connected to each other and then connected to an action block in the end as shown in Figure 12(b). This would enable the user to create very advanced diagnostic chains without having to write diagnostic blocks on their own.

![Diagnostic chains](image)

Figure 12: Diagnostic chains

### 4.4.5 Diagnostic engine software architecture

The core of the diagnostic engine for the diagnostic framework is based around the architecture described in [1], this is a proven architecture that has been used in a real industrial application for several years with good results. Some changes have been introduced to make it compatible with the data engine and to improve the flexibility regarding the execution of diagnostic blocks. A overview of the architecture of the client is shown in Figure 13 and a description of the different modules is given below.

**Configuration parser** Parses the .xml created by the configuration tool containing the configuration settings and returns instructions on how to setup the diagnostic chains to the scheduler.

**Scheduler** Loads all diagnostic blocks from a .dll file and arranges them according to the instructions from the parser and dispatches them to the diagnostic layer when it is time for them to execute.

**Diagnostic layer** Executes diagnostic blocks.
**Action layer** Runs in a separate thread, executes action blocks when ordered to do so by a diagnostic block.

**Data engine client** An instance of the data engine client described in Section 4.3.2.

---

### 4.5 Configuration environment

The configuration environment needs to be able to perform four tasks. Create skeleton code for new diagnostic blocks with the settings the user wants, let the user create chains of diagnostic blocks, generate a .xml file with all settings regarding the diagnostic blocks and chains that can be loaded by the diagnostic engine and finally let the user debug and inject erroneous values into a running diagnostic system.

#### 4.5.1 Integrated development environment plugin

By creating a plug-in for QT Creator all of the things mentioned above can be accomplished. Before deciding to use QT Creator both Visual Studio and Eclipse CDT were considered as possible alternatives, but the choice to use QT Creator was made due to the fact that it is going to be used as the integrated development environment (IDE) of choice throughout the SAP and that the extension capabilities of QT Creator fulfills the requirements of the plug-in.

**Project template** Since no suitable project type is available in QT Creator that fulfils the needs of the configuration plug-in, a custom project template needs to be created that contains all necessary files by default. The necessary files have been identified as the .h and .cpp files containing the base class for all diagnostic blocks, the .h and .cpp files for all standardized blocks and the .xml file used for configuration of the diagnostic engine. A suitable existing project template to use as a base for the extension is the default C++ class library project that already is a part of QT Creator.
Plug-in layout  The plug-in itself needs to consist of three different GUI views where the different tasks can be performed as well as dialog used to create and edit diagnostic blocks. The functionality of the views and the dialog will be described below.

Block overview  In the block overview the users will be able to get an overview of the blocks that are currently available. It will also let the user manage the blocks by calling the block editor.

![Block overview](image)

**Figure 14: Block overview**

Block editor  In the block editor the users will be able to create and edit their own custom diagnostic blocks. The type and number of inputs and outputs will be specified by the user, and from this a code skeleton for the diagnostic block will be generated in the QT Creator project view where the user adds the functionality of the block using C++. The blocks and their inputs and outputs will be added to the configuration .xml-file that will be loaded by the diagnostic engine. A sketch of what the block editor dialog roughly could look like is shown in Figure 15.

Chain creation  In the chain creation the user will be able to configure chains of diagnostic blocks and action blocks. All blocks created using the block creator mode as well as the predefined ones will be available for use in the chains. The inputs and outputs of the blocks can be assigned to arbitrary values that will be available in the data engine as well as inputs or outputs from other blocks. A sketch of what the chain creation view could look like is shown in Figure 16.

Debugging  In order to debug a diagnostic chain, there is a need to know what is going on inside the diagnostic chain during runtime. This can be solved by connecting the configuration tool to the data engine and let the diagnostic
system send all data from the currently executing chain to the data engine. This data can then be presented in a special debug view of the configuration tool so that the diagnostic chain developer gets a clear view of the value of all signals in a diagnostic chain. Fault injection could also be implemented as a feature, which would be useful for testing the functionality of the different diagnostic blocks. Fault injection is a more efficient way then trying to get the target equipment to generate signals useful for testing the blocks. It would also be good if every state change in a diagnostic chain could be recorded so that it could be stepped through in a slower pace later. This would be especially beneficial in the case where the time between the state changes are so small that it is hard to follow the signal values in real time. All of these improvements will greatly reduce the time it takes to debug a diagnostic block. A sketch of what the chain debugging view could look like is shown in Figure 17.
4.5.2 Configuration environment software architecture

If implemented as a plug-in for Qt Creator the plug-in only needs to consist of a few basic modules, the rest of the functionality can be retrieved from Qt Creator’s extensive extension API that is provided just for this kind of plug-ins. A overview of the architecture of the plug-in is shown in Figure 18 and a description of the different modules is given below.

Configuration plug-in Provides the necessary connections between the plug-in and QT Creator.

Block overview The block overview module lists all available diagnostic blocks and lets the user call the block creation module to add, edit or remove diagnostic blocks.

Block creation The block creation module lets the user create, edit diagnostic blocks.

Chain creator The chain creator module lets the user organize diagnostic blocks and action blocks into diagnostic chains.

Debugging The debugging module lets the user see the values present in a diagnostic chain during runtime, the values can also be changed in order to provide fault injection that can be used for testing the diagnostic chains. Additionally state storage has to be implemented here to support playback of signal values in systems with rapidly changing signals.

Data engine client A instance of the data engine client described in Section 4.3.2.
4.5.3 Diagnostic block development using MATLAB

As described in [18] MATLAB can be used to generate C++ code for dynamically linked libraries. This feature could also be used to generate diagnostic and prognostic blocks. Using this method, blocks could be created by defining a mathematical model in MATLAB for the blocks and then generate code for them. The functionality of the block would be less prone to contain errors if this method is used, assuming that the mathematical model has been verified. It would also lead to a more optimized development process since it often is easier to develop a mathematical model conforming to the technical specification of a system then to implement it in C++.

4.6 Prototype implementation

A prototype of the data engine concept has been implemented in order to try out and verify different solution approaches to problems that have come up during this project. It has proved to be very useful when trying out the communication in terms of both data transmission protocols and serialization techniques. The prototype has also been of great help when designing the hybrid storage system consisting of both data structures and a relational database. All of the concepts mentioned in Section 4.1, Section 4.2 and Section 4.3 has been tested and verified using the prototype. No prototype has been developed for the diagnostic engine and configuration environment due to lack of time.

4.7 Solution evaluation

The proposed solution fulfils all the high level demands specified in Section 1.1 and Section 1.2. Modularity is automatically achieved when using the server client architecture that the data engine is designed with. Extra applications can be connected up until the connection limit of the TCP connection is reached. Within the diagnostic engine, modularity has been achieved by reusing the block...
concept that is already being used in the DRE. This block concept also enables almost all diagnostic and prognostic techniques to be used in the system since they run within their own independent modules that are loaded from the outside, and thereby the diagnostic engine itself doesn’t impose any limitations on what types of techniques that can be used.

The suggested improvements to the diagnostic engine regarding scheduling of block execution will enable the CPU utilization to be kept at a desired level, which usually is low to not interfere with more important tasks like e.g. the control loop. The data engine is event driven by the TCP communication. This means that the CPU utilization of the data engine will be kept low as long as the amount of data that is being transferred is kept within reasonable levels, it also helps in reducing the overall response time within the system since all communication is handled on-demand. Usually communication in-between the applications have a quite large negative impact on response times.

The proposed configuration is integrated in the main development environment that is going to be used for SAP. If implemented in the way that is proposed in the solution it will be able to configure all configurable parts of the diagnostic engine.

There are however some drawbacks with this solution. The hybrid storage solution is quite complex and could be replaced by a real-time database running in the RAM on the target hardware. The GUI of the configuration environment is not as good as it could be if a person with adequate interface design experience would have designed it. The configuration environment in the way it looks in this solution is the part that is least likely to be present in a production version of the on-board diagnostic framework. The functionality will probably be there but in a more intuitive GUI.

4.8 Alternative solutions

Alternative platforms for implementing an on-board diagnostic framework were considered during this master thesis. The strongest candidate was National Instruments LabVIEW [14], which would have supported all the necessary communication interfaces as well as providing a very intuitive way of developing the diagnostic chains. Furthermore it could have been used to present the data gathered during runtime, making it a excellent tool for debugging. It is however very resource demanding, therefore it isn’t possible to run it on the hardware described in Section 2.2 which is why it isn’t suggested as the main alternative for the on-board diagnostic framework.

There are two main diagnostic systems that are in use in vehicles around the world today that were thought of as solutions to the problem. On-board diagnostics (OBD) that is used mainly in small vehicles and the FMS-standard that are used in heavy industrial vehicles. A description and a overview of the advantages and disadvantages of both technologies can be found in [8]. These systems are however not sufficient as a solution to the problem since they have a predefined information set to conduct diagnostics and prognostics on.
5 Conclusions

This section will summarize the work done during this master thesis and also give some suggestions to what can be done in the future to continue the development of the on-board diagnostic framework.

5.1 Summary

CrossControl initiated this project to investigate and evaluate how an on-board diagnostic framework could be integrated into their SAP. This master thesis has presented a suitable architecture for both the data engine and the on-board diagnostic framework. The data engine architecture is based around a hybrid storage solution that focuses on performance and extendibility. A simple API has also been suggested that makes it simple for application developers to interact with the data engine. This enables future applications added to the SAP to integrate without any major problems. The data engine concept has been proven to work in a feature limited prototype.

This master thesis also concludes that CrossControl already have a good diagnostic engine that has been tested in the real world with good results and with a few improvements suggested both by [12] and this master thesis could make the diagnostic engine future proof for quite some time. The challenge thereby lies within how the diagnostic engine could be connected to the data engine that forms the core of the SAP, and how the diagnostic framework should be configured. A revised architecture for the diagnostic engine that utilizes the data engine API has been suggested as a solution to the problem. The only drawback to the architecture is that the diagnostic performance is reduced due to the communication overhead introduced by the data engine. Time critical diagnostics therefore has to be performed within the control system as suggested by [16].

An architecture for a configuration environment that is integrated into QT Creator where the configuration of the on-board diagnostic framework is conducted has been suggested. This architecture is based around the block creator and the diagnostic application builder environment, two tools that already exist for configuring the diagnostic system currently developed by CrossControl. The two tools together with a environment that shows and records the execution of diagnostic blocks within the on-board diagnostic framework for debugging purposes are joined together to form the plug-in.

5.2 Future work

The solution presented in this master thesis is limited in performance by the hardware resources available in the targeted hardware platforms. There are however un-utilized resources in the form of GPUs. A good area for further research around this topic would therefore be how to utilize the GPUs to accelerate the execution of the diagnostic blocks in order to free up some capacity in the form of execution cycles on the CPU. Another area of future work could

37
be to look at how the communication between the data engine server and the clients can be improved. The current solution with XML-based messages fulfils all the requirements regarding communication speed that exists today, but I’m confident that it can be improved a lot. Finally, a way of storing data that is both persistent when needed, quick and that doesn’t cause excessive wear on the storage medium used in the hardware platforms should be found.
References


