Design Plan
Diagnosis of ADAPT system

Version 1.0

Author: Daniel Eriksson
Date: December 4, 2009

Course name:	Control Project	E-mail:	diagnos2009@googlegroups.com
Project group:	FFF	Document responsible:	Daniel Eriksson
Course code:	TSRT10	Author's E-mail:	daner963@student.liu.se
Project:	Diagnosis	Document name:	Designplan1_0.pdf

Status

Reviewed		09-10-14
Approved		09-10-14

Project Identity

Group E-mail:	diagnos2009@googlegroups.com
Homepage:	http://www.isy.liu.se/edu/projekt/tsrt10/2009/
Orderer:	Erik Frisk, Linköping University
	Phone: +46 (0)13 28 2035 , **E-mail:** frisk@isy.liu.se
Customer:	The Division of Vehicular Systems, Linköping University
	Phone: +46 (0)13 28 1000 , **E-mail:** Vehicular.Systems@isy.liu.se
Course Responsible:	David Törnqvist, Linköping University
	Phone: +46 (0)13 28 1882, **E-mail:** tornqvist@isy.liu.se
Project Manager:	Niklas Wahlström
Advisors:	Mattias Krysander, Linköping University
	Phone: +46 (0)13 - 28 2198 , **E-mail:** matkr@isy.liu.se

Group Members

Name	Responsibility	Phone	E-mail
Niklas Wahlström	Project manager	0705-122349	nikwa148@student.liu.se
Daniel Eriksson	Document manager	073-4405730	daner963@student.liu.se
Erik Almqvist	Software manager	0705-149935	erija952@student.liu.se
Emil Nilsson	Test manager	073-6766558	emini550@student.liu.se
Andreas Lundberg	Design manager	0704-061227	andlu549@student.liu.se

Document History

Version	Date	Changes made	Sign	Reviewer
0.1	09-09-11	First draft.		Daniel Eriksson
0.2	09-10-11	Second draft.		Daniel Eriksson
1.0	09-10-14	First release.	OK	Daniel Eriksson

Contents

1 Introduction

This document is a system design plan for the diagnosis system that will be developed in the project "Diagnosis of ADAPT system", by project group **FFF**[1]. This document is the base from which the project, schedule and tasks, are defined. This document contains definitions on the software design, suggestions of component models and test quantities. The diagnosis system will be implemented as a computer program that given measurement data generates a diagnosis of the real system.

1.1 Background

NASA is interested in analyzing different ways to monitor whether or not systems that are sent into space are working properly, and also in finding out what the faults are when there are faults present in the system. It is of course beneficial to know exactly which faults that are present in e.g. a satellite before you send someone to repair it. It may also be the case that detecting a fault, and smoothly shutting down the system or limit its activities, can prevent other parts of the system to get damaged. The reasons above illustrates why NASA together with Palo Alto Research Center (PARC) have started an annual competition called the Diagnostic Challange Competition (DCC). The developed diagnois algorithm is intended to participate in the DCC'10, in the Industrial Track System Tier 2 challenge.

1.2 Goals

The goal is to create a diagnosis system that performes as good as possible in the Diagnostic Challange Competition (DCC)[**?**], which primarily means that the diagnosis algorithm should get as high final score as possible, and secondarily a high final rank, in the competition.

2 System overview

Advanced Diagnosis and Prognosis Test Bed (ADAPT) is a facility developed at NASA Ames for testing diagnostic tools and algorithms. The real system that is going to be monitored and diagnosed by our diagnosis system is an electrical power system that is set up in a NASA laboratory. The facilities hardware contains several components, and are intended to illustrate a typical electrical power system in a satellite. This electrical power system has components such as batteries, circuit breakers, resistors, relays, fans, inverters, light bulbs and water pumps. To analyse and observe the circuits there are over 100 sensors which produces data that NASA records. The recorded data from the sensors will be sent in a data sequence together with a commands to a diagnosis algorithm to detect faults. The appendix contains a schematic overview of the ADAPT testbench (figure 5) aswell as a photograph of the physical testbench (figure 6).

2.1 Project division

A reasonable division of the work in the project is into the following three subdivisions: system modeling, diagnostic algorithm and software. Note that these subdivisions are not separate modules of the system, but rather different work divisions. Also these divisions

[1] Finn Fem Fel

are not completely separated from each other, since for example the diagnostic algorithm is based on the system model, and will be implemented in the software.

3 System modelling

A mathematical model of the real system will be crated, based on the real systems circuit diagram [**?**] provided on the DCC homepage. To esitmate model parameters the sample data[**?**], available on the DCC homepage will be used. The model will be used as the basis for the diagnosis system algorithm. The different components of the system may be modeled differently (i.e. by more or less complex models) and parameters in the component models will be determined using the sample data[**?**]. This data will also be used for validation of the models.

3.1 Battery

To determine if a battery is degraded or not the internal resistance of the battery is estimated, since it increases when the battery degrades. The battery is modelled as an ideal voltage generator, with output voltage V_0 ("open circuit voltage"), which depends on the battery's charge level, in series with a resistance R_i, the internal resistance. The voltage generated by the battery is called V, and the current drawn from the battery is called I. In this model V and I are input signals while V_0 and R_i are parameters. The internal restitance can be calculated according to Equation 1.

$$R_i = \frac{V_0 - V}{I} \tag{1}$$

It turns out that R_i varies dynamically with I. To get around this one compare stationary values R_i with its expected values given by the function $R_i^{exp}(I)$, a function that is estimated from the training data sets [**?**]. Different functions,$R_{i,b}^{exp}(I)$, $b \in \{BAT1, BAT2, BAT3\}$, may have to be created for each battery (or maybe for each battery model since BAT3 is not of the same brand and model as BAT1 and BAT2). It is likely that due to noise in the training data sets one will have to have a lower limit of the domain of definition for $R_i^{exp}(I)$, thus limiting the situations in which the battery model can be used.

For some unknown reason, current out from one battery affects the voltage output of other batteries, even though they according to the system lay-out and relay configurations not are connected to each other. Because of this phenomenon V_0 has to be determined for all of the three batteries at times when there is no current drawn from any of the batteries. Fortunately (as stated in the README.txt file in [**?**]) all relays are open at the start of the experiments, so the experiments always starts in a situation where V_0 can be determined for each battery.

3.2 Inverter

An inverter converts direct current (DC) to alternating current (AC), and the model of this system[2] has an input voltage of 24 V (DC), an output power of 1000 W, an output voltage of 120 V (AC) and an output frequency of 60 Hz. In the electrical power system there are two inverters located at different places, based on which load bank that is in use. The behaviour of the inverter is quite static and a simple model could look like

[2]Xantrex prosine 1000, part no. 806-1051.

$$u_{ut}(t) = 120H(u_{in}(t) - 22) \tag{2}$$

where H is the Heaviside function (this equation is based on the Table 2). To create a model of the inverters, the signals above is needed. The signals needed is also described in Table 1.

Signal	Description	Sensors affected(INV1)	Sensors affected(INV2)
i_{out}	The current out from the inverter.	IT167	IT267
u_{in}	The voltage in to the inverter.	E161	E261
u_{out}	The voltage out from the inverter.	E165	E265

Table 1: Useful outputs and inputs.

Each inverter has three different modes and to characterise these data from sensors that especially measure the voltage input and output, but also the current input, is needed. To recognize which mode the inverter is in, the following table will be very useful

Mode	$u_{in} > 22$	$i_{out} > 0$	$u_{out} > 120$
NominalOn	True	–	True
NominalOff	False	False	False
FailedOff	True	False	False

Table 2: The different signals characterize which mode the inverter is in.

In the NominalOn mode the inverted is expected to work well, with an output voltage around 120 V (AC) and an input voltage around 24 V (DC). The inverter switches off when input voltage drops below 22 V. In the NominalOff mode every affected signal should has a value around zero and the inverter switches on when the input voltage rises above 22 V. In the FailedOff mode the inverter does not transmit current or voltage, even if it is supplied with voltage over 22 V.

3.3 Load

The loads can be devided into two groups: the AC loads and the DC loads. The loads can be light bulbs, fans, water pumps or resistors. The signals affecting the AC loads are given in Table 3.

Name	Description
$U_{RMS}(t)$	The RMS value of the voltage across the load
$I_{RMS}(t)$	The RMS value of current through load
$\phi(t)$	Phase shift by which the current is ahead of the voltage.
$P(t)$	The output power from the load.

Table 3: Signals influencing an AC load

According to given data, a good model for all loads is to assume that their impedances are constant. The voltage, the impedace and the current are obeying Ohm's law

$$\tilde{U}(t) = \tilde{Z} \cdot \tilde{I}(t) \tag{3}$$

where $\tilde{U}(t)$, $\tilde{Z}$ and $\tilde{I}(t)$ are the complex represantations of these quatities. By representing the impedance with its magnitude Z and phase θ we get $\tilde{Z} = Ze^{j\theta}$. Ohm's law then gives the realations:

$$U_{RMS}(t) = Z \cdot I_{RMS}(t) \quad (4)$$
$$\phi(t) = \theta \quad (5)$$

and for the output power we have

$$P(t) = U_{RMS}(t) \cdot I_{RMS}(t) \cdot cos(\theta) \quad (6)$$

For two loads connected in parallel, the voltage across each of them is the same; the ratio of currents through any two elements is the inverse ratio of their impedances. The total impedance is given by the formula

$$\frac{1}{\tilde{Z}_{tot}} = \frac{1}{\tilde{Z}_1} + \frac{1}{\tilde{Z}_2} \quad (7)$$

Since the loads are connected in parallel, they will be modelled by its admittance $\tilde{Y}$, which is the reciprocal of the impedance $\tilde{Y} = \tilde{Z}^{-1} = Z^{-1}e^{-j\theta}$ in order to make the calculation of the total admittance easier (only have to sum up the admittances). The modell parameters can be found in Table 4.

Name	Description
Y	The magnitude of the admittance
$\angle Y$	The phase of the admittance

Table 4: Modell parameter of an AC load

and the equations coupling the signals and the parameters will be given by:

$$Y = \frac{I_{RMS}(t)}{U_{RMS}(t)} \quad (8)$$
$$\angle Y = -\phi \quad (9)$$
$$P(t) = U_{RMS}(t) \cdot I_{RMS}(t) \cdot cos(\angle Y) \quad (10)$$

For DC loads one have the same signals, parameters and eqations without the phase influencing (or phase equal to 0).

Each mode of each load has a characteristic set of these model parameters (for the mode FailedOff the admittace is zero).

3.3.1 Power characteristic systems

Some of the loads also have sensors measuring quantities which are caused by the power output of the load. These are light and temperatur sensors for some light bulbs, speed transmittors for some fans, and flow transmitters for the pumps.

The relations between these quantities and the power output will also be described with models. The signals affecting the system are described in Table 5.

These systems usually have a dynamical behaviour, which can be modelled with the ODE defined in 11.

$$y'(t) = k_t(y_P(P(t)) - y(t)) \quad (11)$$

Name	Description
$y(t)$	The measured quantity (light, temperature, speed or flow)
$P(t)$	The output power of the corresponding load

Table 5: Signals influencing power characteristic systems.

where k_t is a proportionality constant (deciding the swiftness of the system) and $y_P(P(t))$ is the working point of the measured quantity as a function of the output power. For the relation between the working point and the output power a quasi-linear relation can be used.

$$(y(t) - y_0)^p = k_0 \cdot P(t) \tag{12}$$

where k_0 is a proportionality constant, y_0 the value of the measured quantity without any power output from the corresponding load, and p is a characteristic exponent coupling the measured quantity with the output power (presumably 2 for the speed and flow transmittor). All model parameters are summerized in Table 6.

Name	Description
k_t	Proportionality constant describing the swiftness of the system
k_0	Proportionality constant between output power and mesured quantity.
y_0	Measured quantity without power from the load.
p	The characteristic exponent of the power characteristic system

Table 6: Model parameter of the power characteristic systems.

Some of the systems do have a very fast dynamical behaviour (more or less all system expept the power - temperature system) and for these systems it could be relevant to consider not to include the dynamical part of the modelling.

3.4 Relay

The relay is a commandable component, which has two boolean signals relatered with it: the command (0=open, 1=closed) and the actuator measuring the position of the relay. The relay has four different modes, given in table 7. Note that some input signal combinations can be explained by two different modes.

Mode	Command	Actuator
NominalClosed	closed	closed
NominalOpen	open	open
StuckOpen	-	open
StuckClosed	-	closed

Table 7: Modes of the relay

Futhermore, the relay doesn't have any model parameter.

When the relay is closed its resistance is virtually zero, resulting in that the voltage drop across the relay is approximately zero. This result can be used to create test quantities that compares voltage measurements from different voltage sensors that are separated by closed relays only.

3.5 Circuit breaker

There are two types of circuit breakers in the system, commandable and non-commandable. The difference between the two is that the commandable has an input (the command), and an additional mode (StuckClosed).

Input to the model is the current $I(t)$ through the circuit breaker, the circuit breaker's actuator position and for the commandable circuit breaker also an open/close command. The only parameter in the model is the rated current I_n of the circuit breaker.

Let $I_{max}(t) = \max_{\tau \leq t} I(\tau)$ be the greatest current that (during the present measurement series) so far has passed through the circuit breaker. A low sampling frequency for $I(t)$ may result in that we have $I(t_1) > I_n$ but get $I_{max}(t_2) \not> I_n$, although $t_2 > t_1$. In reality this event is unlikely since circuit breakers react rather slowly unless the current is many times greater than I_n. Because of its unlikeliness one may disregard the above mentioned event and using the provided description of the modes (AdaptDXC.xml in [**?**]) getting Table 8 and Table 9 for deciding which mode the circuit breaker is in. For those input signal combinations not described by the two tables, you can't tell which mode the circuit breaker is in.

Mode	$I_{max} < I_n$	Actuator
Nominal	true	closed
Tripped	false	open
FailedOpen	true	open

Table 8: Non-commandable: Relationship between mode, maximal current and actuator position.

Mode	$I_{max} < I_n$	Actuator	Command
Nominal	true	closed	closed
Tripped	false	open	–
	true	open	open
FailedOpen	true	open	closed
StuckClosed	true	closed	open

Table 9: Commandable: Relationship between mode, maximal current, actuator position and command.

When the circuit breaker is closed it has a certain resistance, which is a seen as a model parameter. By measuring the voltage before and after the circuit breaker, and the current through it, creating a test variable using Ohm's law is possible.

3.6 Sensors

The sensors can be devided into two groups: boolean sensors measuring boolean signals (actuator position sensor measuring the position of the relay) and scalar sensors measuring real numbers (all other sensors).

The boolean sensors have two modes according to Table 12 and they don't have any model parameter.

Scalar senors measure a certain quantity together with noise. The signals influencing the sensor are given in Table 11.

The sensor could then be described with the equation

Mode	Description
Nominal	Reads 1 (true) if actuator is closed, 0 (false) if open.
Stuck	Reading is stuck at open or closed.

Table 10: Modes of the boolean sensors

Signal	Description
y(t)	The measured signal.
x(t)	The signal to be measured.
n(t)	Measurement noise.

Table 11: Signals influencing scalar sensors

$$y(t) = x(t) + v(t), \text{ where } v(t) \sim N(0, \sigma) \tag{13}$$

where σ is the standard deviation of the (white) noise and a parameter of the model.

Mode	Description
Nominal	The sensor measures the scalar quatity according to equation 13.
Offset	The sensor measures according to equation 13 together with an added unknown constant value.
Stuck	The sensor measures an unknown constant value (without noise).

Table 12: Modes of the boolean sensors

3.7 The flowing current

Kirchoff's current law will be useful to explain the circumstances between the flowing current in the electrical power system:

$$\sum_{k=1}^{n} I_k = 0 \tag{14}$$

All the current into a node is equal to all the current out of the node.

4 The diagnostic algorithm

The diagnosis algorithm will be able to detect and isolate faults in the system based on the modell of the real system. For this algorithm to work it must get measurement data from the sensors of the real system, and data for the inputs to the real system (e.g. ambient temperature and commanded relay positions).

4.1 Test variables

The following section describes the different testvariables used in the diagnostic algortihm.

4.1.1 Battery

From the model in chapter 3.1, the following test variable can be given for a battery:

$$T = R_i - R_i^{exp}(I) = \frac{V_0 - V}{I} - R_i^{exp}(I) \tag{15}$$

Assuming correct sensor values:
For $T > 0$ R_i is greater than expected, i.e. the battery is degraded. Because of noise and disturbances the battery is concidered to be degraded when $T > J$, where $J > 0$ is some threshold.

The sensors may be faulty, so it is always (regardless of the value of T) possible that the current sensor and/or the voltage sensor are damaged.

4.1.2 Inverter

A natural test variable for the model in equation 2 is:

$$T = |u_{out}(t) - u_{exp}| = |u_{out}(t) - 120H(u_{in} - 22)| < J \tag{16}$$

Where J is a threshold, greater than zero, based on noise, off-sets and other disturbances. In the ideal case the test variable should be zero, but it will never be the case because of different disturbances in the measurements. When the test variable is greater than the threshold an alarm is given as the inverter supposedly doesn't work well (or that one or both of the voltage sensors are faulty). Note that Xantrex, according to their data sheets [**?**], guarantees that the output of this component does not differ more than three procent. There are three modes for the inverter (IN) and those are NO (NominalOn), NF (NominalOff) and FO (FailedOff). Here the mode UF is introduced, which represent fault mode for the two voltage sensors (U). To decide any kind of sub-diagnosis for the inverter, the following table will be useful:

$u_{in} > 22$	$u_{out} > 120$	Statement
True	True	IN $\in \{$NO$\}$ $\vee$ U $\in \{$UF$\}$
True	False	IN $\in \{$FO$\}$ $\vee$ U $\in \{$UF$\}$
False	True	IN $\in \{$NF$\}$ $\vee$ U $\in \{$UF$\}$
False	False	-

Table 13: Sub-diagnosis statements for the inverter

It is possible to create more test variables for the inverter if needed. There is sensors that measure the output for the frequency, both the input and output for the current.

4.1.3 Load

In order to check whether a load works as expected or not, one wants to know how much current it draws. Since there are no current sensors for each load (only one for each load bank), that won't be possible. However, together with the voltage sensor and the phase angle transducer the total admittance on that load bank could be calculated with equations 8 and 9. Each load combination (and their modes) would theoretically has its corresponding total admittance, which can be represented as a point in the complex plane. This total admittance can be calculated by simply adding the admittances of the loads according to equation 7. By calculating the actual total admittance, we know which load combination(s) that could sum up to the observed total admittance.

Since the measurements also have noise, a kalman filter will be applied in order to make the confidence interval of a specific load combination smaller.

After studing the data some problems have been identified with this diagnostic approach:

- The phase angle transducer doesn't react as fast as the other sensor sensors resulting in inconsistent behaviour for some samples at abrupt changes. This problem will be solved by making two test variables, one for the admittance magnitude and one for its phase. Then there will be one fast and one slow test variable, instead of having one that measures false (as would be the case if the total admittance were one test variable).
- The motion model in the kalman filter has to be slow enough to achieve a good filter performance, however then it will be too slow by abrupt changes. To solve this, the filtered value will be set to the measured value if it is outside a certain confidence interval and the covariance matrix will be set accordingly.
- Even though the admittance of the load was stated to be constant in chapter 3.3, it has in some cases a small slow dynamical behaviour. By including this in the admittance uncertainty for each load, the confidence intervals of the load combinations will be unnecessary large. Better is to measure the abrupt change in admittance and compare this with the admittances of the single loads (or groups of them).

To explain this in more concrete terms, the following simple motion and sensor models will be used:

$$\begin{aligned} Y[n+1] &= Y[n] + e_1[n] \\ \angle Y[n+1] &= \angle Y[n] + e_2[n] \\ Y^{measure}[n] &= Y[n] + v_1[n] \\ \angle Y^{measure}[n] &= \angle Y[n] + v_2[n] \end{aligned} \tag{17}$$

,where $Y[n]$ and $\angle Y[n]$ are the magnitude and phase of the total admittance at time nT, where T is the sample time, $Y^{measure}[n]$ and $\angle Y^{measure}[n]$ are the measured magnitude and phase of the total admittance, given by equations 8 and 9, $e_i[n]$ and $v_i[n]$ are white noise. The process noise $e_i[n]$ is small and should only correspond to the small changes that occurs for a certain load configuration. Even though the admittances are modelled to be constant, a small random walk do occur, which here is taken into account by the process noise $e_i[n]$.

To this model two kalman filters will be applied observing the magnitude and phase of the total admittance. The filtered value of these quantities will be called $\hat{Y}[n]$ and $\angle\hat{Y}[n]$ and are in practise a low-pass version of $Y[n]$ and $\angle Y[n]$.

The kalman filter algorithm also gives a predicted value of the next sample $Y_p[n]$ and $\angle Y_p[n]$, and a variance of this prediction P_p^{mag} and P_p^{phase} based on all samples until sample $n-1$. The variance of the difference between the new measured value $Y^{measure}[n]$ and the predicted value $Y_p[n]$ can be calculated as

$$\begin{aligned} Var(Y^{measure}[n] - Y_p[n]) &= Var((Y^{measure}[n] - Y[n]) - (Y[n] - Y_p[n])) \\ &= Var(Y^{measure}[n] - Y[n]) + Var(Y[n] - Y_p[n]) \\ &= Var(v_1[n]) + P_p^{mag} \end{aligned} \tag{18}$$

If $(Y^{measure}[n] - Y_p[n])^2 >> Var(v[n]_1) + P_p^{mag}$ this big change can't be explained by the model and the conclution is that the load configuration has been changed. By this event three things will happen:

- This m^{th} abrupt change of the magnitude of the total admittance will be registered as $\Delta\hat{Y}_m[n] = Y^{measure}[n] - Y_p[n]$
- The new filtered value will be set to the measured, i.e. $\hat{Y}[n] = Y^{measure}[n]$
- The variance of the new filtered value will be set to the measurement noise $P_p^{mag} = Var(v_i[n])$

The last two steps are done in order to make the filter adapt the abrupt changes quickly.

Furthermore, there will be an equivalent calculation for the phase.

The following two quantities in Table 14 will be used in our tests.

Name	Description
$\Delta\hat{Y}_m[n]$	The magnitude of the m^{th} abrupt change of the total admittance.
$\Delta\angle\hat{Y}_m[n]$	The magnitude of the m^{th} abrupt change of the total admittance.

Table 14: Usable quantities for designing tests of the loads.

Each of them will be compared with the magnitude Y_i and phase $\angle Y_i$ of a specific load i at the load bank. To make this comparisons the test variables can be defined as follows

$$
\begin{aligned}
T_{im}^{+mag} &= +\Delta\hat{Y}_m[n] - Y_i && (19)\\
T_{im}^{-mag} &= -\Delta\hat{Y}_m[n] - Y_i && (20)\\
T_{im}^{+phase} &= +\angle\Delta\hat{Y}_m[n] - \angle Y_i && (21)\\
T_{im}^{-phase} &= -\angle\Delta\hat{Y}_m[n] - \angle Y_i && (22)\\
& && (23)
\end{aligned}
$$

These test variables will be created at the m^{th} registration of a new abrupt change in the total admittance. If the corresponding relay to the load i is considered to be open only T_{im}^{+mag} and T_{im}^{+phase} will be created, if it is considered to be closed, only T_{im}^{-mag} and T_{im}^{-phase} will be created. If one don't know if it is open or closed, all of them will be created. Since one only want to detect faulty behaviours, no residuals at all will be created if there recently has been a change in command affecting the admittance.

Each load (L_i) has at least the following modes N = Nominal and FO = FailedOff and the corresponding relay (R_i) also has different modes: NC = NominalClosed, NO = NominalOpen, SO = StuckOpen, SC = StuckClosed.

Based on this discussion the following equations describing which statements could be defined

$$
\begin{aligned}
|T_{im}^{+mag}| < J_i &\rightarrow P_{im}^{+mag} = \mathrm{R}_i \in \{\mathrm{SC}\} && (24)\\
|T_{im}^{-mag}| < J_i &\rightarrow P_{im}^{-mag} = \mathrm{R}_i \in \{\mathrm{SO}\} \vee \mathrm{L}_i \in \{\mathrm{FO}\} && (25)\\
|T_{im}^{+mag}| < J_i &\rightarrow P_{im}^{+phase} = \mathrm{R}_i \in \{\mathrm{SC}\} && (26)\\
|T_{im}^{-mag}| < J_i &\rightarrow P_{im}^{-phase} = \mathrm{R}_i \in \{\mathrm{SO}\} \vee \mathrm{L}_i \in \{\mathrm{FO}\} && (27)\\
& && (28)
\end{aligned}
$$

Some loads also have more faulty modes than FailedOff. Each possible admittance change that could be caused by a mode transit from Nominal to another faulty mode apart from FailedOff have to be analysed.

Let $\tilde{Y}_i$ be the admittance for load i in mode Nominal and $\tilde{Y}_{ij}$ the admittance in faulty mode F_j. The admittance change of a mode transit for the load i from mode Nominal to faulty mode F_j can be expressed as

$$\Delta\tilde{Y}_{ij} = \tilde{Y}_{ij} - \tilde{Y}_i \tag{29}$$

In the same whay as before, the following test variables can be defined in order to check if there is a match

$$\begin{aligned} T_{ijm}^{mag} &= \Delta\hat{Y}_m[n] - |\tilde{Y}_{ij}| && (30)\\ T_{ijm}^{phase} &= \angle\Delta\hat{Y}_m[n] - \angle\tilde{Y}_{ij} && (31)\\ & && (32) \end{aligned}$$

with the following statements

$$\begin{aligned} |T_{ijm}^{mag}| < J_{ij} \quad &\rightarrow \quad P_{ijm}^{'mag} = \mathrm{L}_i \in \{\mathrm{F}_j\} && (33)\\ |T_{ijm}^{phase}| < J_{ij} \quad &\rightarrow \quad P_{ijm}^{'phase} = \mathrm{L}_i \in \{\mathrm{F}_j\} && (34)\\ & && (35) \end{aligned}$$

If two tests belonging to the same abrupt change m (for example $|T_{1m}^{-mag}| < J_1$ and $|T_{22m}^{mag}| < J_{22}$) have alarmed, statement P_{1m}^{-mag} *or* $P_{22m}^{'mag}$ could explain this behaviour (this is the case if $|\tilde{Y}_1| \approx |\tilde{Y}_{22}|$)

The final test statements must then be a disjunction of all test statement within the same abrubt change m

$$P_m^{mag} = \vee_i(P_{im}^{+mag}) \vee_i (P_{im}^{-mag}) \vee_{ij} (P_{ijm}^{'mag}) \vee \mathrm{I} \in \{\mathrm{F}\} \vee \mathrm{E} \in \{\mathrm{F}\} \tag{36}$$

$$P_m^{phase} = \vee_i(P_{im}^{+mag}) \vee_i (P_{im}^{-mag}) \vee_{ij} (P_{ijm}^{'mag}) \vee \mathrm{X} \in \{\mathrm{F}\} \tag{37}$$

where I, E, X are the current, voltage and phase angle sensors and F = sensor reading is not reliable, i.e. the sensor is stuck or has an offset.

4.1.4 Power characteristic systems

For each power characteristic systems (described in 3.3.1) a test could be designed since there is a sensor measuring the output quantity of that system. However, to decide the present mode of the corresponding load, one wants to observe the power output of that load. To achieve this, a (kalman) observer will be used, where its motion model is given by the equation 11. It will observe the working point of the measured quantity, and the power can be calculated with the equation 12.

The observed output power P^{obs} can be used together with the command signal of the relay to make the statments in Table 15. In order not to make a false statment after a command change, the tests will not be executed a short period of time after such an event.

Command	Observed output power	Statement
open	$P^{obs} > J_1$	R $\in$ {SC}
closed	$P^{obs} < J_2$	R $\in$ {SO} $\vee$ L$\in$ {FO}
open	$P^{obs} < J_3$	R $\in$ {NO, SO}
closed	$P^{obs} > J_4$	R $\in$ {NC, SC} $\wedge$ L $\in$ {NF,F$_j$}

Table 15: Sub-diagnosis statements for a load/relay

If the load has more faulty modes than FailedOff (FO), one wants to be able to detect and isolate even such a mode. Since the Nominal mode (NF) and each faulty mode F$_j$ has a characteristic power output (P and P$_j$, they can be calculated with equation 10, the statements in Table 16 can be made.

Observed output power	Statement
$\lvert P^{obs} - P\rvert < J_5$	L $\in$ {NF}
$\lvert P^{obs} - P_j\rvert < J_5$	L $\in$ {F$_j$}

Table 16: Sub-diagnosis statements for the special faulty modes of a load

All test statements have to be combined with the test statement S $\in$ {F}, where S is the sensors and F = sensor reading is not reliable, i.e. the sensor is stuck or has an offset.

4.1.5 Relay

The relay (R) has the following modes: NC = NominalClosed, NO = NominalOpen, SO = StuckOpen, SC = StuckClosed. Also, the actuator sensor (A) has fault mode AF = actuator reading is not reliable, i.e. the actuator sensor is stuck. Based on the relay model we get table 17, mapping input signal combinations to sub-diagnosis statements.

V_1 and V_2 are measurements from two voltage sensors that are separated by relays (R) only. Create the test variable

$$T = |V_1 - V_2| \tag{38}$$

and alarm when $T > J$ *and* all relays in R are closed, $J > 0$ is the alarm threshold. In case of an alarm the sub-diagnosis statement is that either any of the voltage sensors or any of the relays are faulty. If case of no alarm, the test quantity does not produce any sub-diagnosis statement.

Command	Actuator	Statement
closed	closed	R $\in$ {NC, SC} $\vee$ A $\in$ {AF}
closed	open	R $\in$ {SO} $\vee$ A $\in$ {AF}
open	closed	R $\in$ {SC} $\vee$ A $\in$ {AF}
open	open	R $\in$ {NO, SO} $\vee$ A $\in$ {AF}

Table 17: Sub-diagnosis statements for a relay

4.1.6 Circuit breaker

The circuit breaker (C) has in the non-commandable case the following modes: N = Nominal, T = Tripped, FO = FailedOpen. The commandable circuit breaker has the mode

SC = StuckClosed in addition to the three modes mentioned above. Also, the current sensor (I) has fault mode IF = current sensor reading is not reliable, i.e. the sensor is stuck or has an offset, and the actuator sensor (A) has fault mode AF = actuator reading is not reliable, i.e. the actuator sensor is stuck. Based on the circuit breaker models we get table 18 and table 19, mapping input signal combinations to sub-diagnosis statements.

Provided voltage measurements V_1 and V_2 before and after the circuit breaker, the current I through it, and its resistance R, the following test variable is created:

$$T = (V_1 - V_2) - R \cdot I \tag{39}$$

The alarm goes off if $T > J$, where $J > 0$ is a threshold, *and* the circuit breaker is closed. In case of an alarm the sub-diagnosis statement of this test is that any of the voltage or current sensors, or the circuit breaker actuator position sensor is faulty. When the alarm has not gone off the test provides no sub-diagnosis statement.

$I_{max} < I_n$	Actuator	Statement
true	closed	C $\in$ {N} $\vee$ I $\in$ {IF} $\vee$ A $\in$ {AF}
true	open	C $\in$ {FO} $\vee$ I $\in$ {IF} $\vee$ A $\in$ {AF}
false	closed	–
false	open	C $\in$ {T} $\vee$ I $\in$ {IF} $\vee$ A $\in$ {AF}

Table 18: Sub-diagnosis statements for a non-commandable circuit breaker

$I_{max} < I_n$	Actuator	Command	Statement
true	closed	closed	C $\in$ {N} $\vee$ I $\in$ {IF} $\vee$ A $\in$ {AF}
true	closed	open	C $\in$ {SC} $\vee$ I $\in$ {IF} $\vee$ A $\in$ {AF}
true	open	closed	C $\in$ {FO} $\vee$ I $\in$ {IF} $\vee$ A $\in$ {AF}
true	open	open	C $\in$ {T} $\vee$ I $\in$ {IF} $\vee$ A $\in$ {AF}
false	closed	–	–
false	open	–	C $\in$ {T} $\vee$ I $\in$ {IF} $\vee$ A $\in$ {AF}

Table 19: Sub-diagnosis statements for a commandable circuit breaker

4.1.7 Sensors

The sensors have two or three different modes depending on which type of sensor it is. The boolean sensors have the modes Nominal (N) and Stuck (S), and for the scalar sensors it resides another mode called Offset (O).

The best way of deciding the mode character for the scalar sensors is to observe the measured values over an arbitrary time interval Δt.

$$T = \Delta y = y(t_1) - y(t_2) \tag{40}$$

where $y(t) = x(t) + n(t)$.

The sensor is in the mode Nominal (N) when $\Delta y(t) < n(t)$ over an arbitrary time interval.

The sensor is in the mode Stuck (S) when $\Delta y(t) = 0$ over an arbitrary long time interval.

The sensor is in the mode Offset (O) when $\Delta y(t) > n(t)$, i.e. the noise is greater than the difference between two values over an arbitrary short time interval.

For the boolean sensors we cannot create any test variable that is sensitive to faults in the sensor only.

4.2 Diagnosis decision logic

To isolate the faults we have to consider all information that we get from the test quantities. The diagnosis will finally be the fault combinations that are consistent with the sub-diagnoses. When a sub-diagnosis is altered, e.g. as a result of a test quantity alarm, the set of diagnoses will be updated using the new information. If a new sub-diagnosis is added (i.e. before it did not say anything, but now it says something), we can update the diagnoses by just plug in this new sub-diagnosis into the algorithm below. In the other case, which is when a sub-diagnosis that did say something before now is saying something else (or nothing at all), we have to restart the algorithm below, plugging in all the current sub-diagnosis statements into it([**?**]).

1. Given old diagnoses D_{old} and a new sub-diagnosis P_i. D are the new diagnoses. Let $D = \emptyset$.
2. If D_j does not imply P_i:
   (a) Remove D_j from D_{old}.
   (b) Extend D_j according to P_i to create diagnoses.
   (c) Delete new diagnoses which imply any old diagnose in D_{old} and add the rest to D.
3. Add the diagnoses in D_{old} to D.

This can be very time consuming if the algorithm have to isolate faults of higher order. An assumption is that multiple faults of higher order is less probable comparing to faults of lower order. Therefore if a possible diagnosis includes at least a certain number of faults, we remove it because it is relatively unlikely. We can add this fault dimension limit into the algorithm above:

1. Given old diagnoses D_{old} and a new sub-diagnosis P_i. D are the new diagnoses. Let $D = \emptyset$. Add also a upper limit for the dimension for multiple faults l.
2. If D_j does not imply P_i:
   (a) Remove D_j from D_{old}.
   (b) Extend D_j according to P_i to create diagnoses.
   (c) *If D_j has higher dimension than l then remove from D_{old}.*
   (d) Delete new diagnoses which imply any old diagnose in D_{old} and add the rest to D.
3. Add the diagnoses in D_{old} to D.

4.3 Detectability and isolability analyses

In order to determine how well the test quantities enable detection and isolation of the faults in the electrical power system analyses the fault detectability and isolability for the diagnosis system will be performed. These analyses are performed in parallell with creating the test quantities, so poor performance in terms of isolability or detectability

is discovered as early as possible in the diagnosis system creation process. If such poor performance is discovered, the basic idea for increasing the performance is to create and add new test quantities to the diagnosis system.

5 Software

This section describes how the diagnosis algorithm will be implemented in terms of structure, classes and communication with the DxC framework provided by NASA.

5.1 Integration with the DxC Framework

5.1.1 Background

In order to take part of the DCC'10 there is a strong requirement that the diagnosis algorithm is fully integrated into the framework of ADAPT, called the DxC.

Although it is possible to develop the diagnosis algorithm in any language, there are two languages recommended by NASA. Those languages are C++ and Java. Other languages have to communicate with the DxC framework using lower level TCP-IP communication, instead of being able to use some of the classes for message passing provided by NASA.

The chosen language for the diagnosis algorithm is C++. The main reason for choosing C++ ahead of Java is mainly that the knowledge of C++ is greater within the developing group.

5.1.2 Communication with the DxC Framework

The DxC framework takes care of both input to and output from the diagnosis algorithm. Figure 1 gives a overview of the different classes provided by the DxC and how they integrate with each other.

The Scenario Loader loads data into the Scenario Data Source, which provides the diagnosis algortihm with data. The data comes from previously recorded data sets that's available for download from dx-competition.org. Several scenarios with different injected fault is available for testing the algorithm, as well as some competition data from the DCC'09 where the injected fault is unknown. The DxC framework also records the output from the diagnosis algoritm (through the Scenario Recorder). Briefly described it records the output from the diagnosis algorithm, for example the current error state if there is one. The DxC also handles storage of the whole scenario. It also evaluates the results. This is done through the last two classes in Figure 1. The results from the scenario is stored in Scenario Results and evaulated and calculated into points that can be used to compare the diagnosis algorithm towards other algortithms in a competition using the Evaluator. How this evaluation is made in a competition is listed in section four of the Diagnostic Challange Competition Announcement[**?**].

All communication made between the modules in Figure 1 is made using a message based TCP-IP protocol. There are classes provided by the DxC for this communication. The communication is made using a Connector and a callback class, handling all kinds of messages to and from the diagnosis algorithm. The diagnosis algorithm only need to send two kinds of messages: a ScenarioStatusData message signalling they are ready to receive data, and DiagnosisData. DiagnosisData is a message containing information on when the system is believed to be in a faulty state. The diagnosis algorithm will also need to handle three kinds of datatypes as input. Once the initial ScenarioStatusData is sent

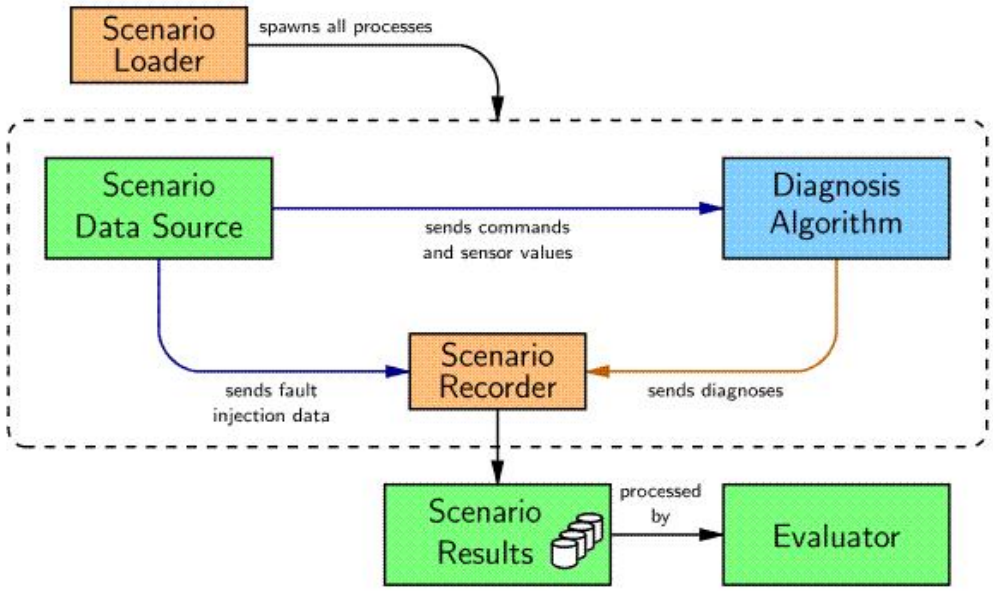

Figure 1: An overview of the ADAPT framwork[**?**].

by the diagnosis algorithm, the DxC will begin sending data of tree types: SensorData, CommandData, and ScenarioStatusData.

The different messages are inherited from a parent class called DxC::DxcData. The structure can be seen in Figure 2.

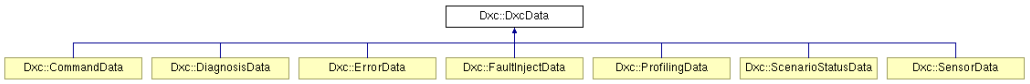

Figure 2: The messages to and from the DxC are inherited by the parent class DxcData as follows.

The datatype sent from the DxC is beeing inherited from a class called DxC::Value. (Read more about internal data storage in section 5.2.4). Here follows a description of all the different messages that are being sent to and from the DxC framework. From the constructors and functions one can see how a message of a certain type can be handled and how a output messages should be passed. The following section describes each of the messages that beeing sent to and from the DxC.

Dxc::CommandData	
Description: Message containing the relays on / off values and other signals that can be set in the diagnostic algorithm.	
Public Functions:	
	CommandData (long long timestamp, const std::string &commandID, Value *commandValue) *Constructor.*
virtual ostream &	put (ostream &) const *Prints DxcData in standardized, parseable format.*
std::string	getCommandID () const *Get command ID string.*
const Value *	getCommandValue () const *Get command Value.*
virtual CommandData *	clone () const *Virtual copy constructor.*

Dxc::ScenarioStatusData	
Description: Message that's beeing sent when the diagnosis algorithm is ready to recieve data, and when the diagnosis algorithm signals that it's finished.	
Public Functions:	
	ScenarioStatusData (const std::string &status) *Constructor.*
	ScenarioStatusData (Timestamp timestamp, const std::string &status) *Constructs with timestamp set to current time.*
std::string	getStatus () const *Returns status.*
virtual ScenarioStatusData *	clone () const *Virtual copy constructor.*
virtual ostream	put (ostream &) const *Prints DxcData in standardized, parseable format.*
Static Public Attributes:	
static const std::string	DA_READY *A Diagnosis Algorithm sends DA_READY to indicate it's prepared to receive data.*
static const std::string	SDS_ENDED *Signals scenario end. DAs must finalize and exit properly or risk termination.*

Dxc::ErrorData	
Description: Error message that's can be sent from the diagnosis algorithm to the DxC(more information in section 5.2.5).	
Public Functions:	
	ErrorData (Timestamp timestamp, const string &error) *Constructor..*
string	getError () const *Returns the error message string.*
virtual ostream &	put (ostream &) const *Prints DxcData in standardized, parseable format.*
virtual ErrorData *	clone () const *Virtual copy constructor.*

Dxc::DiagnosisData	
Description: Message that's being sent when a fault is found, containing a candidate list of faulty components and weights for each of those components.	
Public Types:	
typedef std::set ⟨ Candidate,ltCandidate ⟩	CandidateSet *Candidate set typedef.*
Public Functions:	
	DiagnosisData (Timestamp timestamp, bool detectionSignal=false, bool isolationSignal=false, const CandidateSet &isolation=CandidateSet(), const std::string &notes="") *Constructor to initialize the DiagnosisData with timestamp.* DiagnosisData (bool detectionSignal=false, bool isolationSignal=false, const CandidateSet &isolation=CandidateSet(), const std::string &notes="") *Constructor to initialize the DiagnosisData with current time as timestamp.*
bool	getDetectionSignal () const *True if, according to the diagnosis, the system is believed to be in a faulty state.*
bool	getIsolationSignal () const *True if faults have been isolated, i.e. candidates exist.*
virtual DiagnosisData *	clone () const *Virtual copy constructor.*
virtual ostream &	put (ostream &) const *Prints DxcData in standardized, parseable format.*
Classes:	
struct	Candidate *Maps a set of component IDs to (hypothesized) faulty states.*

Dxc::SensorData	
Description: Sensordata sent by the DxC loader contains following message.	
Public Types:	
typedef std::map ⟨ std::string, const Value * ⟩	CandidateSet *Candidate set typedef.*
Public Functions:	
	SensorData (Timestamp timestamp, const SensorValueMap &sensorMap) *Constructor.*
virtual ostream &	put (ostream &) const *Prints DxcData in standardized, parseable format.*
SensorValueMap	getSensorValueMap () const *Returns the map from sensor to Value.*
virtual SensorData *	clone () const *Virtual copy constructor.*

5.2 Implementation

This section covers how the diagnosis algorithm will be developed and how the structure of the diagnosis algorithm will look like.

There is a demand that the developed diagnosis algorithm is generic and flexible to changes in parameters and in sensor configuration. This suits the object oriented programming style very well, and as a result of these demands on the software the diagnosis algorithm will be object oriented. An overview of how the diagnosis algorithm will work can be seen in Figure 3.

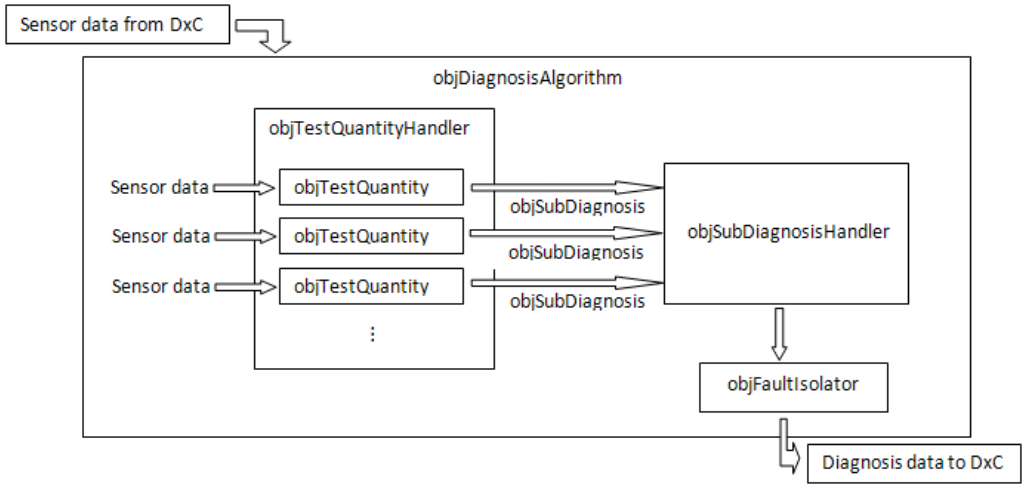

Figure 3: An overview of how the diagnosis algorithm will be implemented.

The diagnosis algorithm can be divided into a few major parts, namely test quantities,a sub-diagnosis handler and a fault isolator. Among with help classes and containers, these class objects will form the core of the diagnosis algorithm.

A test quantity is a parent class. It takes in the sensor data and delivers a sub-diagnosis, containing information about what parts that might have a fault, or if some parts can be guaranteed not having a fault. In between the start and finish, test quantities can be quite different however. A test quantity for checking a relay might look very different from a test quantity that's checking a load. Test quantities will need to be programmed specifically for each type of test. This will be implemented so that each specific test quantity will be a subclass from the parent test quantity class. In this way you can specify how each algorithm will be designed. Exactly how these tests will be coded is a hard to predict before the modelling part of the project is done.

Test quantities take the sensor data it needs from the global sensor map and filter the data. What data is needed and what filtering will be done is based on which test is to be performed. Several tests will only need data from a few sensors, thus making it unwise to load data from all sensors into every test quantity. If possible the calculating of test quantities can be a subject of threading (using pthreads) for increased speed up.

The information gathered from all the test quantities will be placed in a sub-diagnosis container class. This container will store information from all test quantities. When all test quantities has delivered its result to the container, the container will be passed as input to the the fault isolator.

The fault isolator is a decision maker that is taking the subdiagnoses from all the available testquantities and finds the diagnoses of them as described in section 4. This diagnosis

will be presented back to the DxC through a message of type DxC::DiagnosisData as mentioned in section 5.1.2. Using this way of representing the test quantities, with a parent class that contains only a few general functions and specific subclasses for different types of tests, increases the generic and flexible touch of the diagnosis algorithm, due to the fact that the fault isolator class does not require a named test quantity to be able to produce a diagnosis. Of course, it might be hard to isolate a diagnosis without certain test quantities, but that will be the case even without an object oriented form of programming.

In order to make sure that the requirement that says that "The diagnosis algorithm shall be designed so that it is possible to handle a change in a model parameter for a specific component" (Requirement 23 in), it shall be mentioned that all model parameters shall be placed in a separate file containing definitions of all model parameters. Model parameters shall instead be defined like:

#define RELAY_1_RESISTANCE 140

5.2.1 Time limits during software execution

According to the Requirement List the start up time has a maximum limit of 30 seconds. There is also a maximum cycle time limit of 500 ms to make sure that everything is finished before the next cycle begins. During the implementation and testing of the software these time limits will be tested against and necessary adaptation of the software will be made.

5.2.2 Class structure

The implementation of the diagnosis algorithm is strongly based on the object-oriented class structure. Here follows a list of tables that describes the included class objects.

class objDiagnosisAlgorithm	
Description: Main class that holds the whole algorithm. This class communicates with the DxC Framework	
Public Functions:	
	objDiagnosisAlgorithm() *Constructor*
	˜objDiagnosisAlgorithm() *Destructor*
int	collectSensorData(map<string, DxVariable> sensorData) *Takes a map of sensor data where the map key is the sensor name. The function returns 1 if ok and 0 if error.*
int	collectCommandData(map<string, DxVariable> commandData) *Takes a map of command data where the map key is the relay name. The function returns 1 if ok and 0 if error.*
int	collectScenarioData(map<string, DxVariable> scenarioData) *Takes a map of scenario data where the map key is the relay name. The function returns 1 if ok and 0 if error.*
map<string, DxVariable> int	runTestQuantities(void) *The functions runs through the test quantity objects.* runFaultIsolator(void) *Applies the diagnosis calculator on the sub-diagnoses collected from the test quantities.*
Variables:	
objTestQuantityHandler * objFaultIsolator *	testQuantityHandler faultIsolator

class objTestQuantity	
Description: Superclass that handles a single test quantity. It takes the sensor data as input and returns a sub-diagnosis to the objTestQuantityHandler class	
Public Functions:	
	objTestQuantity() *Constructor*
	˜objTestQuantity() *Destructor*
objSubdiagnosis *	run() *Run the testquantity and return a objSubdiagnosis object*
void	addSensorDependencies() *Function to add a sensor dependency to a single testquantity*
void	getSensorData() *Function that fetches the correct sensor data from the big sensor map in the diagnosis algorithm class. This function gets the sensorvalues that a particular testquantity needs. the list of sensor*
string	getTestQuantityName() *Returns the name of the test quantity*
Variables:	
string $map < std :: string,$ $vector < Value* >>$	testQuantityName localSensors

class objTestQuantityHandler	
Description: Container class that holds all objTestQuantity objects.	
Public Functions:	
	objTestQualityHandler() *Constructor*
	˜objTestQualityHandler() *Destructor*
objSubdiagnosisHandler *	run(void) *Runs through all test quantities and returns a obj-SubdiagnosisHandler.*
int	addTestQuantity(objTestQuantity * newTestQuantity) *Add a objTestQuantity object to the handler.*
Variables:	
vector<objTestQuantity>	testQuantities

class objSubdiagnosisHandler	
Description: Stores the sub-diagnosis recieved from the objTestQuantity objects to be used later in the objDiagnosis object.	
Public Functions:	
	objSubdiagnosisHandler() *Constructor*
	˜objSubdiagnosisHandler *Destructor*
int	addSubdiagnosis() *Add a sub-diagnosis to the objSubdiagnosisHandler.* *The function returns 1 if of and 0 if error.*
vector<objSubdiagnosis>*	getSubdiagnosis() *Returns all the objSubdiagnosis objects as a vector.*
Variables:	
vector<objSubdiagnosis>	subDiagnoses

class objSubdiagnosis	
Description: Holds the sub-diagnosis from a test quantity.	
Public Functions:	
	objSubdiagnosis() *Constructor*
	˜objSubdiagnosis() *Destructor*
void	setTestQuantity(std::string testQuantityName) *Set the name of the test quantity*
std::string	getTestQuantity() *Returns the test quantity name.*
void	addFault(std::string fault) *Add a fault candidate to the sub-diagnosis*
void	addNonFault(std::string nonFault) *Add a non fault candidate to the sub-diagnosis*
vector<std::string>	getFaults() *Returns the faults as a vector<std::string>*
vector<std::string>	getNonFaults() *Returns the non faults as a vector<std::string>*
Variables:	
string vector<Fault> vector<string>	testQuantity faults nonFaults

class objFaultIsolation	
Description: Takes the faulty and non faulty components in the subDiagnosisHandler and returns a diagnosis.	
Public Functions:	
	objFaultIsolation() ~objFaultIsolation() returnDiagnosis(objSubdiagnosisHandler subdiagnosises) *Takes all the sub-diagnoses and calculates the diagnosis.*
Dxc::DiagnosisData	
Variables:	
-	-

struct Fault	
Description: Stores a fault in two strings, one for components name and one for fault mode.	
Variables:	
std::string component	Text string that identifies the component.
std::string mode	Text string that identifies the fault mode.

5.2.3 Fault Isolation

The object structure handles the sensor data and gives the test quantities this information as input and from these test quantities come the sub-diagnoses. How to interpret this information to decide the most probable diagnosis is handled in the objDiagnosisCalculator class.

The algorithm will be implemented as described in section 4.2.

5.2.4 Data storage

As mentioned in section 5.1.2 the software need to handle three kinds of input messages from the DxC framework. These three are scenario status messages, command messages and sensor data messages. The information from these messages needs to be stored somewhere in the software.

The diagnosis algorithm will get a callback signal whenever a command data, sensor data or a scenario status is recieved from the DxC framework. When a command data is recieved a series of parameters will be set. When a scenario status arrives the algorithm will be set ready to start. When sensor data arrives all sensor values will be stored in a global sensor map. After this the task of checking for faulty parts and calculating, a diagnosis will take place.

The handling of the typeid (ScenarioStatusData) message will be done directly in the callback class. The scenarioStatus message only contains a end of scenario tag that will stop our algorithm. This is done by setting a flag.

As multiple scenarios looks to be run one after another, the algorithm needs to clear out it's old scenariodata and send out a new message to the DxC telling the scenario that it's ready to go again.

The storage of the input signals recieved in a typeid (CommandData) message will be stored within the testQuantityHandler. The command data contains information on whether or not a certain switch or relay is set open or closed, and this information will be kept as a map containing the id of the relay and a bool value containing information about if the relay is open or not.

map<string,bool> commandType; commandType commandMap;

The sensordata is recieved from the DxC as a typeid(SensorData) message. This SensorData object contains a SensorValueMap that holds information about each sensors values. The structure of the SensorValueMap is

typedef map<std::string, const Value* > SensorValueMap

where the string contains a sensorID and the Value points to the sensors value. The sensorID is used by the test quantities when they query for the sensor values it needs. The Value in the SensorValueMap can be either a integer, a string, a boolean or a real (complex) value.

The Value parameter of the sensorValueMap is of the type DxcValue. DxcValue is a parentclass to that got subclasses for integer, string, boolean and real as seen in figure 4. The value inserted should be of the correct subclass to the DxcValue so that each sensor will get the correct datatype.

In order to be able to filter sensor signals and take averages it needs to store a number of sensor values from each sensor in the diagnosis algorithm. The number of values stored should be set as a parameter, and is an object for tuning later on. The sensor values

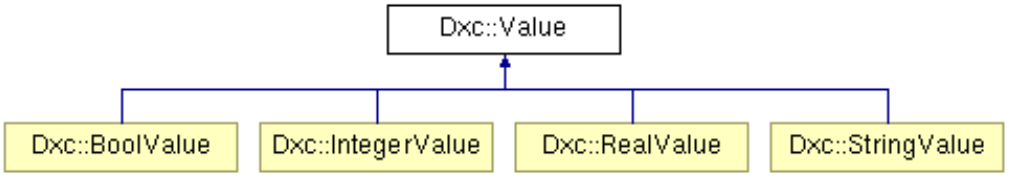

Figure 4: The values of the sensordata are beeing inherited from a parentclass called DxcValues.

received from the DxC will therefore be transformed into a map container that stores the latest arrived sensor values in a vector.

typedef map<std::string, const vector<Value4∗ >>
sensorValueTypeMap sensorValueTypeMap sensorMap;

For example. If a sensor measures an integer value, the look of the element in the sensorMap containing this item would be something like:

map<string,vector<int>>.

In general, sensor IDs and fault IDs is chosen to be stored as strings. However, this may not be the fastest way to calculate data. Comparison between strings do take more time than comparison between integers. With performance as an aspect, storing these values as a string might not be the best call. This decision is instead made as a try to ease the understanding of the code. Sensor ID will arrive to the diagnostis algorithm from the DxC as strings, thus making it natural to keep that chain through the diagnosis algorithm. The option here would be to create a table that maps all sensor IDs and possible diagnosises to an integer value, that would have to be casted back to a string once the fault isolator have calculated its intersection between the subdiagnosises. Mapping all faults and sensor to integers tends to be quite time consuming. That's why strings are kept as a primary option until performance becomes a factor.

5.2.5 Error handling in the software

Even if this project does not focus on errors and exeption handling, adding some mechanism for throwing and catching errors might make it easier to detect bugs inside the diagnostic algorithm itself. Figure 2 in section 5.1.2 shows that the DxC supports an additional message type called DxC::ErrorData. This class is a simple class that allows simple passing of error messages between the DxC and the Diagnostic Algorithm. The structure of the DxC::ErrorData can be seen 5.1.2.

By adding a simple exception class to the diagnostic algorithm and putting a try and catch block around the callbacks for handling sensordata commanddata and scenariostatusdata, one should get a simple error handling that still allows and extra degree of debugging possibilities. This error can be thrown wherever needed in the diagnostic algorithm, and can be used for debugging purposes.The exeptionclass in the diagnostic algorithm should for simplicitys sake be inherited by the runtime_error class as follows:

```
class diagnostic_error : public runtime_error {
public:
 diagnostic_error(const string& argument = " ") : runtime_error(argument){}
};
```

5.2.6 Software manual

In order to get an easy understanding of how to change or implement new test quantities in the software, an manual will be written. It will also hold a short tutorial of how fo find and change the different component parameters.

6 Coding standards

In order to make it easier to maintain a good overall impression in the code, this section introduces some coding standards that is to be used within the project. A good start is to always try to give functions and variables names that connects to its use, and to use indentation. In addition, the following can be taken as a recommendation for coding:

- Header files shall allways end with **.h** and implementation files should end with **.cc**.
- Header files should have a guard to prevent multiple inclusions
  #ifndef FOO_H
  #define FOO_H
  #endif
- Each class shall have it's own headerfile and cc file, containing it's functions.
- Objects shall allways be named with the prefix obj.
  Example:
  class objDiagnosis;
  would define a class called Diagnosis.
- Functions shall always be named with a lower-case letter for the first word, and then a capital letter for the remaining words. This is to be done without a separation with an underline.
  Example:
  void setName()
  is prefered ahead of
  void set_name()
- If the use isn't obvious a strategical comment is to be place before a code section, for example before a function or a module, that describes what use the following section does.
- A tactical comment usually explains the use of a certain row. It's to be placed at the end of the row if possible, otherwise just before the row.
- When creating functions and statements, allways place the { on a separate row. This is to support the unified look with the rest of the DxC framework.

```
if (value)
{
       return (1);
}
```

is prefered ahead of:

```
if (value) {
       return(1);
}
```

In addition, all files should start with a tag that *briefly* explains the use of the file and who made the file. This section will look abit different in header files and in implementation files. Simply copy the following section to each header file:

```
/*
 * CDIO DIAGNOSIS ALGORITHM - PROJECT FFF
 *
 * IDENTIFY
 * Filename:   foo.h
 * Type:       Module declaration
 * Written by:
 *
 * DESCRIPTION
 * Brief description
 */

#ifndef FOO_H
#define FOO_H

/*
 * USED LIBRARIES AND MODULES
 */

#include <*.h>
#include "*.h"
#endif
```

and the following into each implementation file. The // commands in the end is just to clarify order of placement for local variables and functions:

```
/*
 * CDIO DIAGNOSIS ALGORITHM - PROJECT FFF
 *
 * IDENTIFY
 * Filename:   foo.cc
 * Type:       Definitions that belongs to module Foo, non inline
 * Written by:
 */

/*
 * USED LIBRARIES AND MODULES
 */

#include "Foo.h"

// LOCAL OBJECTS
int localInt;

//LOCAL DECLARATIONS
void fie(...);

//LOCAL DEFINITIONS
void fie(...)
{
        ...
}
```

7 ADAPT figures

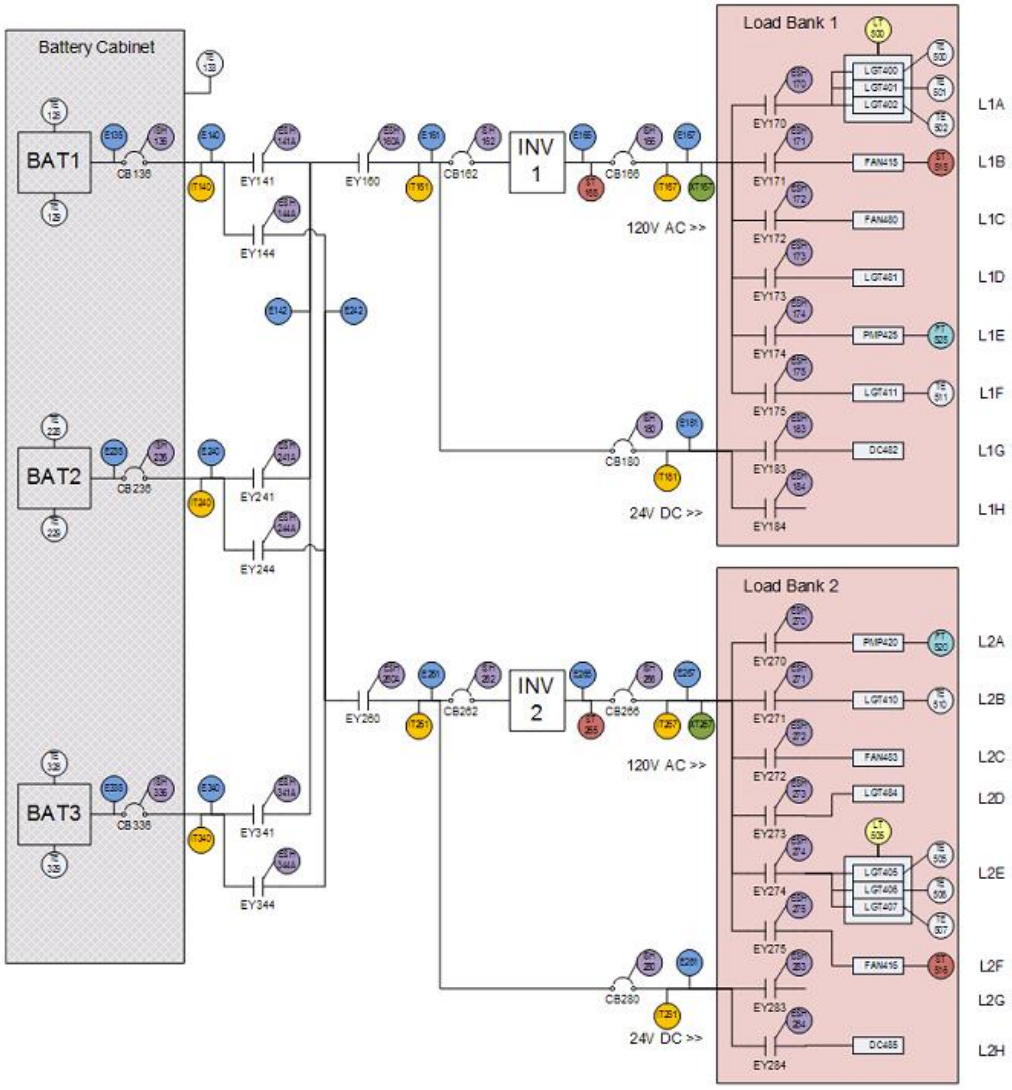

Figure 5: An overview of the ADAPT system and its components[**?**].

Figure 6: A picture of the ADAPT system[**?**].