# Technical Documentation
# Diagnosis of ADAPT system

Version 1.0

Author: Daniel Eriksson
Date: December 15, 2009

**Status**

| | | |
|---|---|---|
| Reviewed | | |
| Approved | | |


| | | | |
|---|---|---|---|
| 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: | TechnicalDoc1_0.pdf |

## 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 | 09-12-02 | First draft. | | Daniel Eriksson |
| 0.2 | 09-12-04 | Second draft. | | Daniel Eriksson |
| 1.0 | 09-12-07 | First release. | | Daniel Eriksson |

# Contents

# 1 Introduction

This document is a technical documentation for the diagnosis system that has been developed in the project "Diagnosis of ADAPT system" at Linköping University by project group **FFF**[1]. This document contains a thorough description and analyses of this project and the diagnosis algorithm. The code of the diagnosis algorithm is not included in this document.

## 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 Challenge Competition (DCC). The developed diagnosis algorithm is intended to participate in the DCC'10, in the Industrial Track System Tier 2 challenge.

## 1.2 Goals

The goal of the project was to create a diagnosis system that performs as good as possible in a Diagnostic Challenge Competition (DCC)[dxp09], 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 has been 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 is sent in a data sequence together with commands to a diagnosis algorithm to detect faults. The appendix contains a schematic overview of the ADAPT testbench (figure 16) as well as a photograph of the physical testbench (figure 17).

## 2.1 Project division

The work within the project was divided into the following three subdivisions: system modeling, diagnostic algorithm and software. These subdivisions are not separate modules of the system, but rather different work divisions. Also these divisions are not completely separated from each other, since for example the diagnostic algorithm is based on the

[1] Finn Fem Fel

system model, and is implemented in the software. In this document the chapters will be divided in the same way, with the addition of an analysis section.

# 3 System modelling

A mathematical model of the electrical power system system is created, based on this systems circuit diagram [sys09] provided on the DCC homepage. The model is used as the basis for the diagnosis system algorithm. The different components of the system is modelled differently (i.e. by more or less complex models) and parameters in the component models is determined using the sample data[tes09]. This data is also 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 $V_0$ is a parameter. The internal resistance can be calculated from measurements according to Equation 1.

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

The internal resistance $R_i$ varies dynamically with $I$, and to get around this only stationary values of $R_i$ are used. The stationary dependence of $R_i$ by $I$ is modeled according to Equation 2, where $A$ and $e$ are model parameters.

$$R_i = \frac{A}{I^e} \tag{2}$$

Although there are three batteries, one of which is of another brand and model than the other two, their internal resistances (modeled according to Equation 2) can all be described by the same model parameters. The parameters are $A = 0.23$ and $e = 0.51$. This model is only considered valid for currents above 2.0 A, since no sample data[tes09] with lower currents was available.

Figure 1 shows stationary internal resistance measurements, calculated using the sample data[tes09], and the internal resistance model. It also shows two measurements for the battery from the AdaptLite system (this battery is of the same brand and model as BAT1 and BAT2). These measurements are included partly to illustrate how degradation in a battery manifests itself.

The open circuit voltage is affected by the charge level of the battery, so as a battery is discharged during the course of a scenario $V_0$ decreases. The model used for the change in $V_0$ from time $t = t_1$ to $t = t_2$ is given by Equation 3, where I(t) is the current out from the battery, K is a model parameter, $T_{abs}$ is the absolute battery temperature (considered to be constant), given in degrees Rankine (which is the Fahrenheit scale but with absolute zero as 0 degrees), and $Q_{nom}$ is the nominal battery capacity. The diagnosis program uses $K = 0.005\ V/R$, $T_{abs} = 530\ R$ and $Q_{nom} = 360000\ C$ (100 Ah: BAT1 and BAT2) or

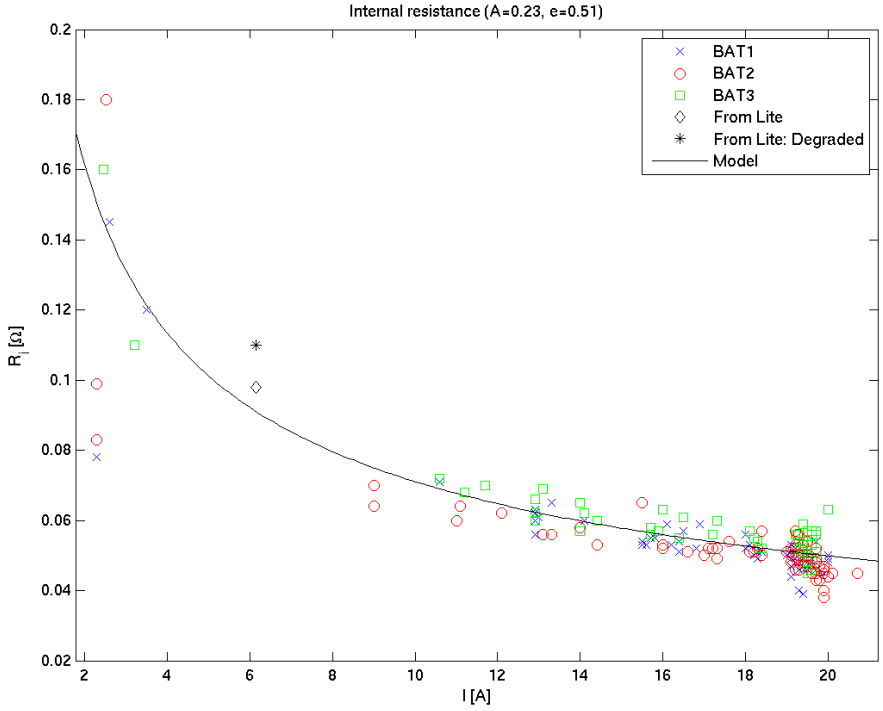



Figure 1: Measurements and model of the batteries internal resistances

$Q_{nom} = 180000\ C$ (50 Ah: BAT3).

$$V_0(t_2) = V_0(t_1) - K \cdot T_{abs} \cdot \frac{1}{Q_{nom}} \int_{t_1}^{t_2} I(t)dt \tag{3}$$

For some unknown reason (possibly electromagnetic fields from the wires and/or other components), current out from one battery sometimes affects the voltage output of other batteries (voltage gets lower), 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 (current readings below 0.3 A are considered as "no current"). Fortunately (as stated in the `README.txt` file in [tes09]) all relays are open at the start of the experiments, so the experiments always start 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 these inverters[2] in the given electrical power system has an input voltage of 24 V, an output voltage of 120 V 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.

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

### 3.2.1 Model based on the efficiency

To determine if there are any losses of energy in an inverter it is a good idea to look at the power before and after the inverter. In the direct current case the instantaneous power can be expressed as

$$P_{in}(t) = U_{in}(t) \cdot I_{in}(t) \tag{4}$$

where $U_{in}$ and $I_{in}$ represents the voltage and the current in to the inverter. The average power for sinusoidal voltage and current is

$$P_{out}(t) = U_{RMSout}(t) \cdot I_{RMSout}(t) \cdot cos(\phi) \tag{5}$$

where $U_{RMS}$ and $I_{RMS}$ represents the root mean square values of the sinusoidal alternating voltage and current out of the inverter. The phase angle between the voltage and the current sine functions is denoted as $\phi$. Furthermore, one can calculate the energy conversion efficiency, $\eta$, to get a ratio that describes the relationship between the input and the output power.

$$\eta = \frac{P_{out}(t)}{P_{in}(t)} = A(1 - e^{-kP_{in}(t)}) \tag{6}$$

Equation 6 is the model of the inverters behaviour, according to the loss of energy, where $P_{in}$ and $P_{out}$ are input signals while $A$ and $k$ are parameters. The parameter k describes the exponential gradient and the parameter A describes the ratio between the input and the output power for the specific inverter. In other words, how much energy the inverter loses. The estimated parameters differ between the two inverters, "INV1" and "INV2". For "INV1" the parameters is $A_1 = 0.78$ and $k_1 = 0.025$ and for "INV2" the parameters is $A_2 = 0.90$ and $k_2 = 0.017$. In other words the loss of energy is approximately 22 % for the inverter one and 10 % for the inverter two, as can be seen in Figure 2.

The red squares in Figure 2 represents the maximal efficiency and the black circles represents the minimal efficiency in different work areas, estimated from the given test data. The curve is estimated to lie between the squares and the circles.

### 3.2.2 Model based on the voltage

One can also express the behaviour of the inverter in a more simple model. In the ideal case, the relation between the input and output voltage can be expressed as

$$U_{out}(t) = 120 \cdot H\big(U_{in}(t) - 24\big) \tag{7}$$

where H is the Heaviside function. This says that if the voltage input is $\geq$ 24 VDC then the inverter have an output of 120 VAC, otherwise the output is zero. This equation is based on the specific type of inverter that is given in the electrical power system.

Each inverter has three different modes and to characterise these, data from sensors that measure the voltage input and output is needed. To recognize the present mode of the inverter, Table 1 can be used.

In the NominalOn mode the inverter 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 have a value around zero and the inverter switches on when the input voltage rises above

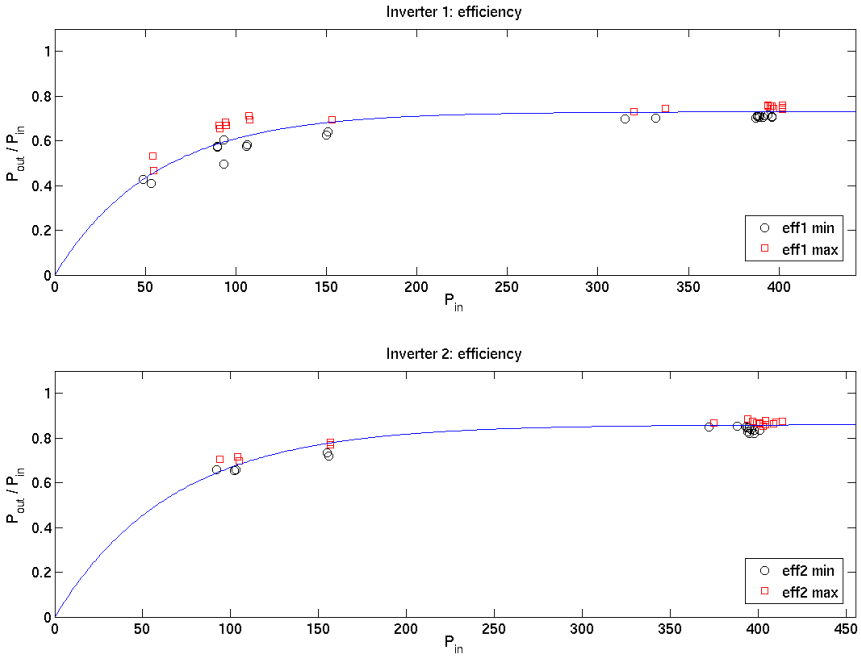



Figure 2: The estimated model for the both inverters.

Table 1: The different signals characterize which mode the inverter is in.

| Mode | $U_{in} > 22$ | $U_{out} > 120$ |
|---|---|---|
| NominalOn | True | True |
| NominalOff | False | False |
| FailedOff | True | False |

22 V. In the FailedOff mode the inverter does not transmit current or voltage, even if it is supplied with a voltage over 22 V.

## 3.3 Load

The loads can be divided 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 2.

Table 2: Signals influencing an AC load

| 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. |

According to given data, a good model for all loads is to assume that their impedances

are constant. The voltage, the impedance and the current are obeying Ohm's law

$$\tilde{U}(t) = \tilde{Z} \cdot \tilde{I}(t) \tag{8}$$

where $\tilde{U}(t)$, $\tilde{Z}$ and $\tilde{I}(t)$ are the complex representations of these quantities. By representing the impedance with its magnitude $Z$ and phase $\theta$ one gets $\tilde{Z} = Ze^{j\theta}$. Ohm's law then gives the relations:

$$U_{RMS}(t) = Z \cdot I_{RMS}(t) \tag{9}$$
$$\phi(t) = \theta \tag{10}$$

and for the output power it follows that

$$P(t) = U_{RMS}(t) \cdot I_{RMS}(t) \cdot cos(\theta) \tag{11}$$

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

$$\frac{1}{\tilde{Z}_{tot}} = \frac{1}{\tilde{Z}_1} + \frac{1}{\tilde{Z}_2} \tag{12}$$

Since the loads are connected in parallel, they are modelled by their 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 (the admittances only need to be summed up). The model parameters can be found in Table 3.

Table 3: Model parameter of an AC load

| Name | Description |
|---|---|
| $Y$ | The magnitude of the admittance |
| $\angle Y$ | The phase of the admittance |
| $Var(Y)$ | The variance of the magnitude |
| $Var(\angle Y)$ | The variance of the phase |
| $Cov(Y, \angle Y)$ | The covariance of the magnitude and the phase |

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

$$Y = \frac{I_{RMS}(t)}{U_{RMS}(t)} \tag{13}$$
$$\angle Y = -\phi \tag{14}$$
$$P(t) = U_{RMS}(t) \cdot I_{RMS}(t) \cdot cos(\angle Y) \tag{15}$$

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

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

In the load characteristic scenarios `Exp_599-Exp_603` in [tes09] only one load is turned on at a time and these datasets have been used to estimate the admittances. Also information

from other scenarios where a certain load is tuned on/off or fails off have been used. The admittance change of that action corresponds to the admittance of that load. For each admittance the covariance matrix has been estimated. Since the covariance matrix $\Sigma(Y, \angle Y)$ is symmetric

$$\Sigma(Y, \angle Y) = \begin{pmatrix} Var(Y) & Cov(Y, \angle Y) \\ Cov(Y, \angle Y) & Var(\angle Y) \end{pmatrix} \tag{16}$$

only the variances of the magnitude and phase, and their covariance have to be regarded as model parameters and they are included in Table 3. All estimated admittances and their confidence intervals are presented in Figure (3). According to this figure there are good chances to isolate most of the loads, whereas all resistive loads, i.e. the light bulbs, seems to have similar admittances.

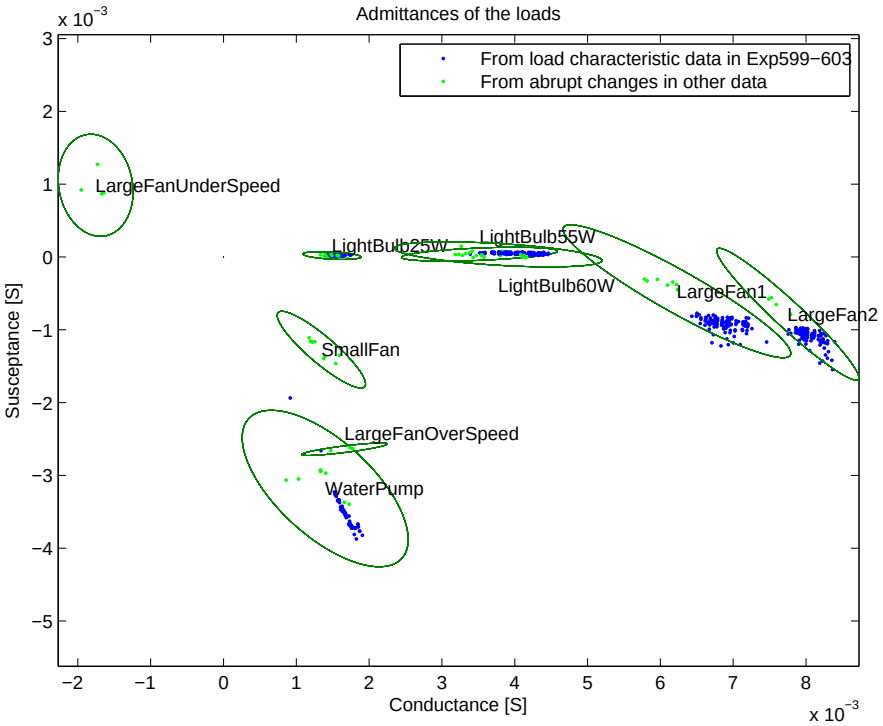



Figure 3: Measurements and models of the admittances for the AC loads with a confidence interval of four standard diviations

### 3.3.1 Power characteristic systems

Some of the loads also have sensors measuring quantities which are affected by the power output of the load. These are light and temperature sensors for some light bulbs, speed transmitters 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 4.

Table 4: Signals influencing power characteristic systems.

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

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

$$y'(t) = k_t(y_P(P(t)) - y(t)) \tag{17}$$

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.

$$(y_P(P(t)) - y_0)^p = k_0 \cdot P(t) \tag{18}$$

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. Since the power has a quadratic relation to speed and flows, $p = 2$ has been chosen for these quantities. For other quantities $p = 1$, i.e. there is assumed to be a linear relation between the power and the measured quantity. All model parameters are summarized in Table 5.

Table 5: Model parameter of the power characteristic systems.

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

In order to estimate the parameters, a discrete version of Equations (17) and (18) is needed. To achieve this, the Euler-approximation has been used:

$$y'(kT) = \frac{y[k+1] - y[k]}{T} \tag{19}$$

Where $T$ is the sample time, $y[k] = y(kT)$, and the dynamic Equation (17) can be described as:

$$y[k+1] = (1 - k_t T)y[k] + k_t T \cdot y_P(P[k]) \tag{20}$$

Some of the systems do have a very fast dynamical behaviour and for these systems it is relevant not to include this, since that only would result in numerical problems in the diagnosis algorithm later on. This is done for the flow transmitter and the light sensors by setting $k_t = 1/T$, since that will cancel out the $y[k]$ term in equation (20).

## 3.4 Relay

The relay is a commandable component, which has two boolean signals related 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 6. Note that some input signal combinations can be explained by two different modes.

Table 6: Modes of the relay

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

Furthermore, the relay doesn't have any model parameter because the modes are only described by the logical states of the command and the actuator.

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 a command input, 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 largest 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 in reality $I(t_1) > I_n$ but $I_{max}(t_2) \not> I_n$ is received 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 use the provided description of the modes (AdaptDXC.xml in [sys09]). This will result in Table 7 and Table 8 for deciding which mode the circuit breaker is in. Those input signal combinations which are not described by the two tables are impossible (provided that the sensors are working properly).

Table 7: Non-commandable: Relationship between mode, maximal current and actuator position.

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

There is a voltage drop $V_{CB}(t)$ over the circuit breaker. When the circuit breaker is open $V_{CB}(t)$ can vary widely, but when the circuit breaker is closed, $V_{CB}(t)$ is fairly small. One natural approach is to consider the closed circuit breaker to be a resistor with constant resistance, but this model does not describe the behaviour of the circuit breakers in the ADAPT system. Instead, the model used for $V_{CB}(t)$ is the one given by Equation 21, where $V_{CB,min}$ and $V_{CB,max}$ are model parameters, and each circuit breaker has its own

Table 8: Commandable: Relationship between mode, maximal current, actuator position and command.

| 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 |

model parameters.

$$V_{CB,min} \leq V_{CB}(t) \leq V_{CB,max} \tag{21}$$

### 3.6 Sensors

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

The boolean sensors have two modes according to Table 10 and they don't have any model parameter.

Table 9: Modes of the boolean sensors

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

Scalar sensors measure a certain quantity together with noise. The scalar sensors has an additional offset mode according to Table 10. The sensor test quantities are modelled using the sensors working area as described in section 4.1.7. Where the stuck mode is separated from the offset mode by counting consecutive equal values in a row.

Table 10: Modes of the boolean sensors

| Mode | Description |
|---|---|
| Nominal | The sensor measures the scalar quantity. |
| Offset | The sensor measures the scalar quantity together with an added unknown constant value. |
| Stuck | The sensor measures an unknown constant value (without noise). |

## 4 The diagnostic algorithm

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

## 4.1 Test variables

While designing the test quantities used within the algorithm focus has been placed on designing sufficient test quantities per component rather than metodically going through the system from the beginning to the end, creating *all* possible test quanties for each and every component thus making sure that isolability is preserved throughout the system. While designing test quantities, one have instead looked upon a single component and then focused on creating on what you can get from this specific component.

In the process of creating the test quantities focus has also been to avoid test quantities giving false positive alarm. This is due to the mechanisms of the diagnosis algorithm (see Section 4.2) where it's impossible to withdraw a given diagnosis. Therefore it's important that no test quantity delivers a non correct diagnosis to the algorithm that's calculating the total diagnosis.

The following subsections describes the different test variables used in the diagnostic algorithm.

### 4.1.1 Battery

From the model in chapter 3.1, the test variable for a battery is given by Equations 22-23, where $J = 0.07$ is a threshold parameter. $A$ and $e$ are the model parameters from the modeling chapter. The $R_i$-measurements, model and threshold are shown in Figure 4.

$$T = R_i - R_{i,thr}(I) = \frac{V_0 - V}{I} - R_{i,thr}(I) \tag{22}$$

$$R_{i,thr}(I) = \frac{A + J}{I^e} \tag{23}$$

$V_0$ is calculated during the start of each scenario, and it is the average of the battery voltage measurements until the time when any of the batteries' current is above 0.3 A.

Assuming correct sensor values, $T > 0$ means that $R_i$ is significantly greater than expected, i.e. the battery is degraded.

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

The internal resistance is not calculated if $I < 2.0$, since the model is only valid for currents above 2.0 A. Furthermore, the battery test quantity is only active when the internal resistance is stationary. In this section, the descrete time index $n$ will be used instead of the continuous time variable $t$. The method for testing stationarity is the following:

First $R_i$ is low-pass filtered according to Equation 24, where $\alpha = 0.1$ is a filter parameter.

$$R_i^{filt}[n] = (1 - \alpha)R_i^{filt}[n - 1] + \alpha R_i[n] \tag{24}$$

Thereafter a difference measurement $\Delta R_i^{filt}[n]$ is calculated according to Equation 25, where $N = 20$ is a filter parameter that states how many samples to use for calculating $\Delta R_i^{filt}[n]$.

$$\Delta R_i^{filt}[n] = \frac{1}{N/2} \sum_{i=0}^{N/2-1} R_i^{filt}[n - i] - \frac{1}{N/2} \sum_{i=N/2}^{N} R_i^{filt}[n - i] \tag{25}$$

Finally, $R_i$ is considered to be stationary if $|\Delta R_i^{filt}[n]| < J_{diff}$ holds for all of the $M = 20$ last values of $\Delta R_i^{filt}$. $J_{diff} = 0.005$ is the threshold for the differences.

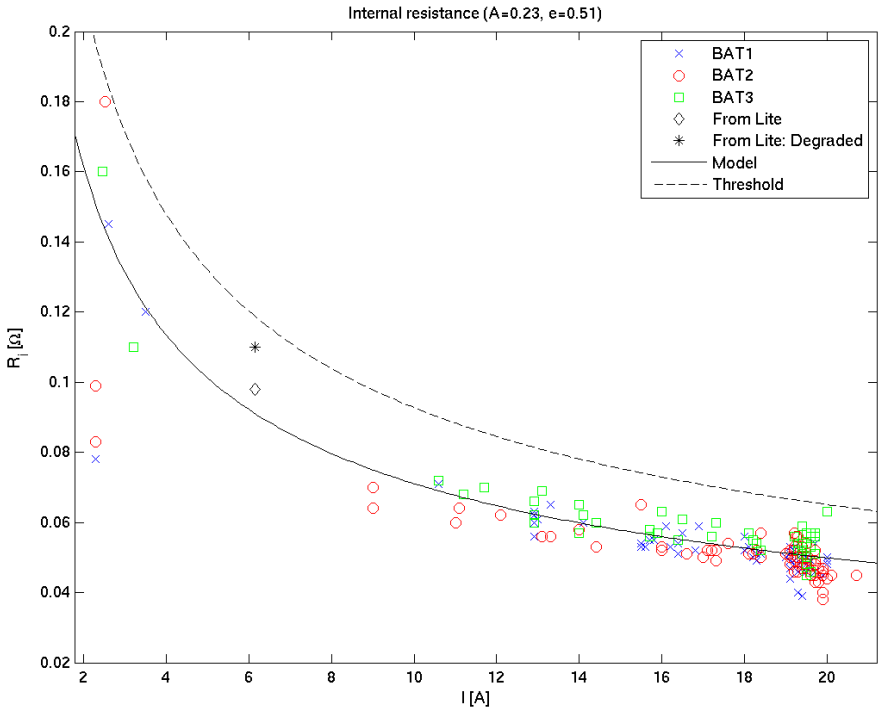



Figure 4: Measurements, model and threshold for the batteries internal resistances

### 4.1.2 Inverter

To determine whether the inverter behaves as it should when it is active, these two thresholds, based on the model in equation 6, have been derived

$$J_u = A_u(1 - e^{-k_u P_{in}(t)}) \quad (26)$$
$$J_l = A_l(1 - e^{-k_l P_{in}(t)}) \quad (27)$$

where $J_u$ and $J_l$ are the upper and the lower limit, for the inverters normal working area, see Figure 5. When creating the test variable exactly as the given model in Equation 6, the allowed working space is $J_l < T < J_u$. If the test variable, $T$, goes outside the allowed limits, assuming stationarity, an alarm is given and a subdiagnois containing that the inverter(INV) is in the mode FailedOff(FO) is sent to the diagnosis algorithm. There are a several sensors that is in use when creating this test variable, so the subdiagnosis also contain the possible faults that these sensors represents, see Table 11 below:

Here are the faults in the sensors that measure voltages (UF), currents (IF) and the phase angel (PF) present.

Note that this test variable only gives an alarm if:

- The current in to the inverter is over a certain threshold (14 A).

Table 11: Sub-diagnosis statements for the inverter

| Scenario | Statement |
|---|---|
| $T > J_u > J_l$ | INV $\in$ { FO } $\vee$ $U_{in} \in$ { UF } $\vee$ $U_{out} \in$ { UF } $\vee$ $I_{in} \in$ { IF } $\vee$ $I_{out} \in$ { IF } $\vee$ $\phi \in$ { PF } |
| $T < J_l < J_u$ | INV $\in$ { FO } $\vee$ $U_{in} \in$ { UF } $\vee$ $U_{out} \in$ { UF } $\vee$ $I_{in} \in$ { IF } $\vee$ $I_{out} \in$ { IF } $\vee$ $\phi \in$ { PF } |

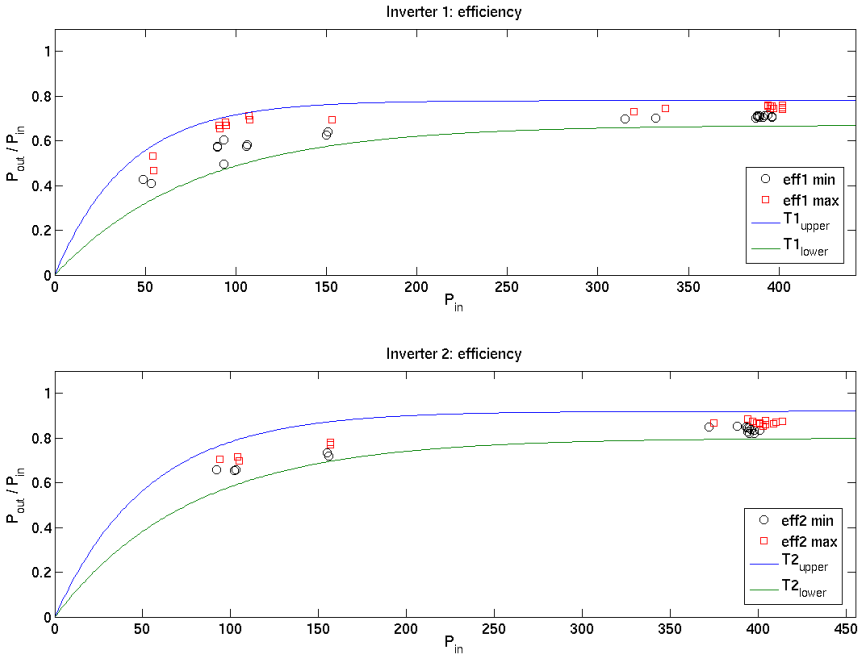



Figure 5: The upper and lower limits for the test variable.

- The voltage in to the inverter is over a certain threshold (22 V).
- The current in to the inverter is in a stationary state/working area. This is determined by measuring the difference between the present and the previous sample(not greater than 0.4 A) in a number of samples (20).
- The test variable is outside its allowed working area in 20 samples.
- The circuit breaker before the inverter is closed, see the attached figure 16.

The advantage of this test variable is that it is reliable but on the other side it is reacting slowly. An alarm is not given until detected a fault during 20 samples (i.e. 10 seconds).

The other test variable, based on the voltage input and output, is more simple. The decision logic for this test variable is calculated by the table 12. There are three modes for the inverter (INV) and those are NO (NominalOn), NF (NominalOff) and FO (FailedOff).

In Table 12 the mode UF represents the fault mode for the two voltage sensors (U). As can be seen in this table there is only during the second case that a sub-diagnosis (FO) is

Table 12: Sub-diagnosis statements for the inverter

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

sent. One should also know that the voltage sensor before the inverter is only read if the circuit breaker is closed, analogous to the test variable for the power.

Note that Xantrex, according to their data sheets [xan09], guarantees that the output of this component does not differ more than three procent.

### 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 sensor for each load (only one for each load bank), that is not possible. However, together with the voltage sensor and the phase angle transducer the total admittance on that load bank can be calculated with Equation 13 and Equation 14. Each load has its admittance, which can be represented as a point with a belonging confidence interval in the complex plane. This representation is shown in Figure 3 and are parameterized with the model parameter in Table 3.

When an abrupt change in the measured admittance has been detected, this change will correspond to the admittance of the load that has been added or removed from the load bank, since one know that parallel coupling of admittances obey the equation 12. This admittance change will be compared with all admittances of all loads that are turned on at that load bank.

Since the measurements also have noise, two Kalman filters (one for magnitude and one for the phase) has been applied and the following simple motion and sensor models will be used:

$$\begin{aligned} Y[n+1] &= Y[n] + e_1[n] \\ \angle Y[n+1] &= \angle Y[n] + e_2[n] \\ Y^{measure}[n] &= Y[n] + v_1[n] \\ \angle Y^{measure}[n] &= \angle Y[n] + v_2[n] \end{aligned} \tag{28}$$

$Y[n]$ 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 13 and 14, $e_i[n]$ 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 have been applied observing the magnitude and phase of the total admittance. The filtered value of these quantities will be called $\hat{Y}_f[n]$ and $\angle\hat{Y}_f[n]$ 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:

$$\begin{aligned} Var[Y^{measure}[n] - Y_p[n]] &= Var[(Y^{measure}[n] - Y[n]] - (Y[n] - Y_p[n]) \\ &= Var[Y^{measure}[n] - Y[n]] + Var[Y[n] - Y_p[n]] \\ &= Var[v_1[n]] + P_p^{mag} \end{aligned} \tag{29}$$

One gets that:

$$T = \frac{Y^{measure}[n] - Y_p[n]}{\sqrt{Var[v[n]_1] + P_p^{mag}}} \sim N(0,1) \tag{30}$$

If $|T| > J_1$, which can not be explained by the model, and the conclusion is that the load configuration has been changed. By this event three things will happen:

- The last filtered value of the admittance before the abrupt change will be saved as $\tilde{Y}_{old}$.
- The new filtered value will be set to the measured, i.e. $\hat{Y}_f[n] = Y^{measure}[n]$.
- 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. See also Figure 6(b).

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

The following two complex stochastic variables in Table 13 will be used in our tests.

Table 13: Usable quantities for designing tests of the loads.

| Name | Description |
|---|---|
| $\tilde{Y}_{old}$ | The filtered value of admittance before the abrupt change. |
| $\tilde{Y}_f[n]$ | The last filtered value of admittance. |

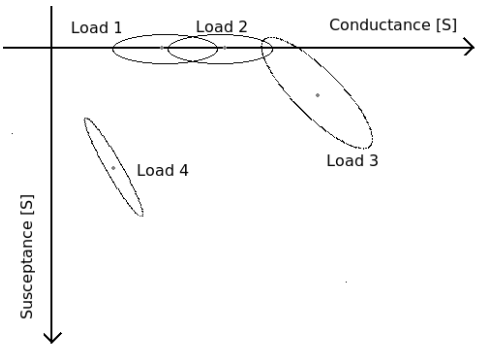
According to Equation (12) the admittance of two loads connected in parallel is hold simply by summing up the admittances of the loads. Therefore the admittance of a load bank when a load is removed from that load bank is found by subtracting the admittance of the load from the total admittance of the load bank. Thus, for example, to check if load $i$ has failed off, the admittance $\tilde{Y}_{old} - \tilde{Y}_i$ is compared with $\tilde{Y}_f[n]$.

If only a change in the magnitude has been detected the magnitudes will be compared (see Figure 6(c). If only a change in the phase has been detected, only the phases will be compared, and when a change in both of them has been detected, their admittances will be compared (see Figure 6(d))

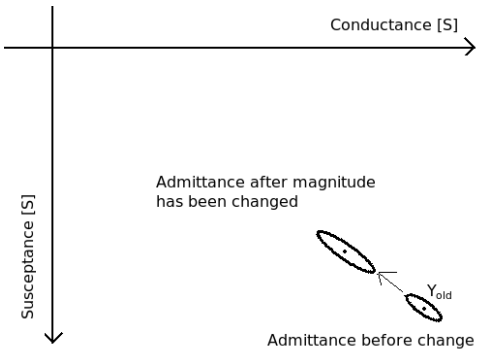
To make this test graphically described in Figure 6(c) and 6(d), set

$$\tilde{Y}_i^* = \tilde{Y}_{old} - \tilde{Y}_i \tag{31}$$

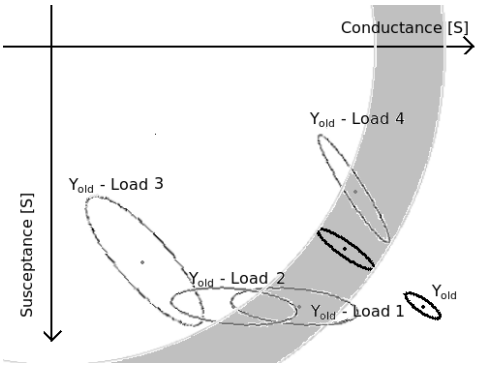
$$\Delta\tilde{Y}_i = \tilde{Y}_i^* - \tilde{Y}_f[n] \tag{32}$$



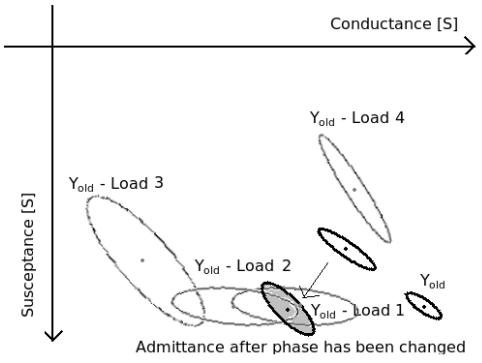

(a) The modelled admittances of the loads




(b) An abrupt change in the magnitude measurement has been detected. The old value of the admittance is saved




(c) All loads will be subtracted from $Y_{old}$. Since only the magnitude is known of the new measurement, only this will be compared with the calculated differences. Load 1,2 or 4 could be the one that has failed off.




(d) Also the phase measurement has made an abrupt change. The new measured admittance compares with the calculated differences. Load 1,2 could be the one that has failed off.

Figure 6: An abrupt change detects in the magnitude measurement, and shortly afterwards in the phase measurement. With use of the models of the loads a statement can be done about which loads that have failed off.

for all loads $i$ and define the following test variable.

$$T_i^{FO} = \begin{cases} \frac{|\tilde{Y}_i^*| - |\tilde{Y}_f[n]|}{\sqrt{Var[|\tilde{Y}_i^*|] + Var[|\tilde{Y}_f[n]|]}} & \text{, when only magnitude measurement has been changed} \\ \frac{\angle \tilde{Y}_i^* - \angle \tilde{Y}_f[n]}{\sqrt{Var[\angle \tilde{Y}_i^*] + Var[\angle (\tilde{Y}_f[n])]}} & \text{, when only phase measurement has been changed} \\ \sqrt{\Delta \tilde{Y}_i^T (\Sigma[\Delta \tilde{Y}_i])^{-1} \Delta \tilde{Y}_i} & \text{, when both measurements have been changed} \end{cases} \tag{33}$$

where $\Sigma[\Delta \tilde{Y}_i]$ is the covariance matrix of $\Delta \tilde{Y}_i$.

With the normalization in equation block 33, one gets that $T_i^{FO} \sim N(0,1)$ if the load $i$ actually had failed off and one single threshold $J_2$ can be used for all tests. Therefore the

following test statements can be defined.

$$|T_i^{FO}| < J_2 \quad \rightarrow \quad P_i^{FO} = \mathrm{L}_i \in \{\mathrm{FO}\} \tag{34}$$

where each ($\mathrm{L}_i$) has at least the following modes N = Nominal and FO = FailedOff.

Furthermore, some loads have more fault modes $\mathrm{F}_k$ than only FO = FailedOff with a corresponding characteristic admittance change. In the same manner as above, tests can be generated testing this changes

$$|T_{ij}^{F}| < J_2 \quad \rightarrow \quad P_{ij}^{F} = \mathrm{L}_i \in \{\mathrm{F}_j\} \tag{35}$$

for all loads $i$ and faults $j$.

In the same way, it can be tested if a load relay ($\mathrm{R}_i$) is SO = StuckOpen or SC = StuckClosed by summing up all admittances at this relay and compare this admittance with the detected admittance change. This will generate tests

$$|T_k^{SO}| < J_2 \quad \rightarrow \quad P_k^{SO} = \mathrm{R}_k \in \{\mathrm{SO}\} \tag{36}$$
$$|T_k^{SC}| < J_2 \quad \rightarrow \quad P_k^{SC} = \mathrm{R}_k \in \{\mathrm{SC}\} \tag{37}$$
$$\tag{38}$$

for all load relays $k$.

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

$$P = \vee_i(P_i^{FO}) \vee_{ij} (P_{ij}^{F}) \vee_k (P_k^{SO}) \vee_k (P_k^{SC}) \tag{39}$$

If only the magnitude or phase measurement have been changed, also an Offset or Stuck in the current respectively the phase sensor could explain the behaviour and these will be added to the test statement. However, if both measurement have been changed only an admittance change is assumed to have taken place.

The test (39) will only be created if a load relay recently has not been changed. In that case, one would expect an admittance change. By a relay configuration change, if no admittance change is detected the changed relays are considered to be StuckOpen or StuckClosed respectively. If a change has been detected everything is considered to be normal (see Table 14)

Table 14: Sub-diagnosis statements for by command change of a load relay

| Command change | Admittance change detected? | Statement |
|---|---|---|
| closed → open | no | $\mathrm{R}_k \in \{\mathrm{SC}\}$ |
| open → closed | no | $\mathrm{R}_k \in \{\mathrm{SO}\}$ |
| closed → open | yes | - |
| open → closed | yes | - |

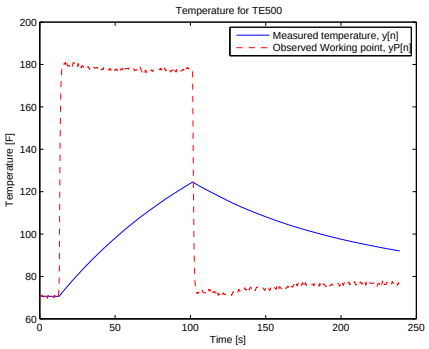
### 4.1.4 Power characteristic systems

For each power characteristic systems (described in Section 3.3.1) a test could be designed since there is a sensor measuring the output quantity of that system. However, to decide

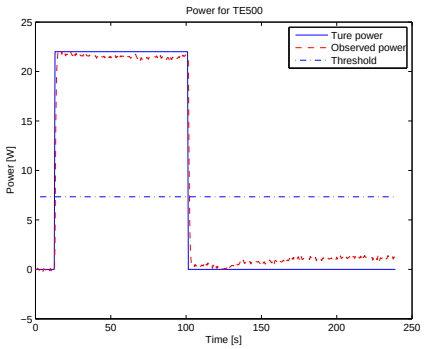
the present mode of the corresponding load, one wants to observe the power output of that load. To achieve this, a (Kalman) observer has been used, where its motion model is given by the equation 17. It will observe the working point of the measured quantity, and the power can be calculated with the equation 18.

If relay of load $i$ is closed, the observed output power $P_i^{obs}$ from the sensor at that load $i$ can be used to detect if the load $L_i$ is failed off (see statement in Table 17). The expected value of the power $P_i$ of $P_i^{obs}$ when the load in mode Nominal, can be calculated by Equation (15) and the threshold $J_i = P/3$ has been used.

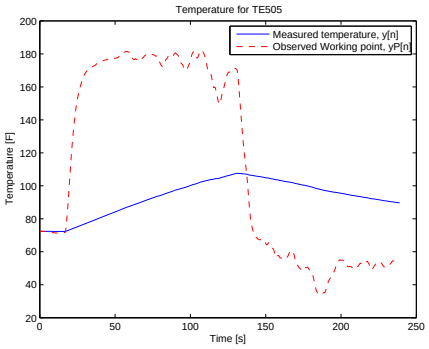
The tuning of this Kalman observer is based on the signal noise making observer for noisy signals more slower than for those with less noise. The performance of this kalman filter is presented in Figure 7 for the temperature sensors TE500, has not much signal noise, and for the temperature sensor TE505 which has significantly more signal noise.



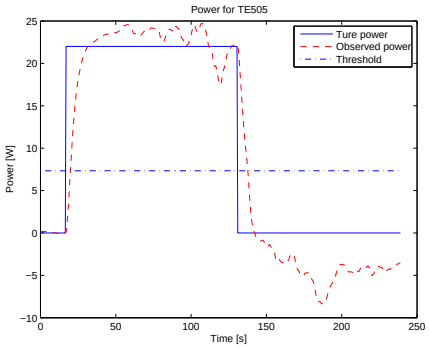

(a) Measured temperature and observed working point for sensor TE500




(b) Observed an true power for LGT400 based on measurements from TE500 with threshold




(c) Measured temperature and observed working point for sensor TE500




(d) Observed an true power for LGT405 based on measurements from TE505 with threshold.

Figure 7: Performance of the Kalman filter observing the power. The sensor TE500 has significantly more noise than TE505, which results in a slower Kalman filter. This must be the case, otherwise would the power observed with TE505 cause false alarm or the power observed by TE500 be unnecessary slow.

In order not to make a false statement after a command change, the tests will not be executed a short period of time after such an event.

If the load has more faulty modes than FailedOff (FO), one wants to be able to detect and isolate even such a mode. Since each faulty mode $F_j$ in load $i$ has a characteristic

Table 15: Sub-diagnosis statements for a load

| Command | Observed output power | Statement |
|---|---|---|
| closed | $P_i^{obs} < J_i$ | R ∈ L ∈ {FO} |

power output ($P_{ij}$), the statements in Table 16 can be made.

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

| Observed output power | Statement |
|---|---|
| $\lvert P_i^{obs} - P_{ij}\rvert < J_{ij}$ | L ∈ {$F_j$} |

In the same manner, it can be tested if a load relay ($R_k$) belonging to load $i$ is StuckOpen = SO or StuckClosed = SC.

Table 17: Sub-diagnosis statements for a relay

| Command | Observed output power | Statement |
|---|---|---|
| open | $P_i^{obs} > J_i$ | $R_k$ ∈ {SC} |
| closed | $P_i^{obs} < J_i$ | $R_k$ ∈ {SO} |

All test statements have to be combined with the test statement S ∈ {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 18, 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

$$T = |V_1 - V_2| \tag{40}$$

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. These controls are split into two tests. If the test alarms then there is either a fault in one of the voltage sensors or the relay is open. If the latest command says that the relay should be closed then the relay could be stuck open. If the position sensor says that the relay is closed then the sensor could be stuck. Here we have two tests: one that tests if the relay is stuck open and one that tests if the position sensor is stuck. These test quantities can only detect a fault if the relay is closed. It is only when the relay is closed that one know that there should be no voltage drop. If the relay is open we can not say how the voltage should differ before and after the relay.

Another test is to control if the position sensor, which is measuring the relays position, agrees with the last command to the relay. If the last command says that the relay should be closed but the sensor says that it is open then there is a problem. This test works all the time and do not require the system to have a specific setup like the earlier two tests. The resulting sub-diagnosis for each relay can be seen in Table 18.

Table 18: Sub-diagnosis statements for a relay

| Command | Actuator | Statement |
|---|---|---|
| closed | open | R ∈ {SO} ∨ A ∈ {AF} |
| open | closed | R ∈ {SC} ∨ A ∈ {AF} |

### 4.1.6 Circuit breaker

The circuit breaker (C) has, in the non-commandable case, the following modes: N = Nominal, T = Tripped and 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 19 and Table 20, mapping input signal combinations to sub-diagnosis statements. The test quantity does not send a sub-diagnosis statement if the statement contains a non-faulty mode assignment to the circuit breaker. This is because the CPU load increases very fast when nominal mode sub-diagnoses are sent to the fault isolator. (With only 5-6 circuit breaker test quantities sending nominal diagnoses the fault isolation takes so much time that the diagnosis program doesn't finish within the required 0.5 seconds. Also, the gain for the diagnosis program with using nominal sub-diagnoses has been observed to be small.)

Table 19: Sub-diagnosis statements for a non-commandable circuit breaker. Only the italicised statement may be returned by the test variable.

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

Table 20: Sub-diagnosis statements for a commandable circuit breaker. Only italicised statements may be returned by the test variable.

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

Provided voltage measurements $V_1$ before the circuit breaker and $V_2$ after the circuit breaker a test variable is created according to Equation 41.

$$T = \Delta V_{filt} \tag{41}$$

where

$$\Delta V_{filt} = \begin{cases} \text{mean}(\overline{\Delta V}) & \text{if } \Delta V > J_{amp} \text{ and } \text{std}(\overline{\Delta V}) < J_{std} \\ \Delta V & \text{otherwise} \end{cases} \quad (42)$$

where $\Delta V = V_1 - V_2$ and $\overline{\Delta V}$ is a vector containing the $N = 4$ previous $\Delta V$ values. $J_{amp}$ and $J_{std}$ are filter parameters, and they are are not the same for all circuit breakers.

The filter computing $\Delta V_{filt}$ completely filters out one-sample spikes (this is the purpose of the filter), and the small drawback is that step changes in $\Delta V$ are delayed by a few samples.

The alarm goes off if, when the circuit breaker is closed according to its position sensor, it does not hold that $J_{min} < T < J_{max}$, where

$$J_{min} = \min(J_1 \hat{V}_{CB,min},\ \hat{V}_{CB,min}) - J_2$$
$$J_{max} = \max(J_1 \hat{V}_{CB,max},\ \hat{V}_{CB,max}) + J_2$$

$\hat{V}_{CB,min}$ and $\hat{V}_{CB,max}$ are empiric estimates (using the sample data[tes09]) of the model parameters from the modeling chapter (chapter 4.1.6). $J_1 = 1.5$ and $J_2 = 0.2$ are threshold parameters.

In case of an alarm the sub-diagnosis statement of this test is that any of the voltage sensors or the circuit breaker actuator position sensor is faulty.

### 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).

While designing the test variables for the sensors, it is desirable to be able to detect faults in the sensor by using the sensor only, without the use of other components within the system. The reason for this is that once another component is introduced, if a fault is detected the fault may always be derived from the new component and not only the used sensors. For the boolean sensors we cannot create any test variable that is sensitive to faults in the sensor only. For the scalar sensors the test quantity for checking for sensor faults tries to use the tested sensor only by using the known work areas for each specific sensor. This is, as explained below, far from a perfect check and the sensors is an area to be improved in a future improvement of the diagnosis algorithm.

For the scalar sensors there is three different modes as described as above, Nominal (N) and Stuck (S) and Offset (O). To determine if a sensor is stuck the number of consecutive elements detected is counted for each sensor. This number is compared to a predefined value that holds the highest allowed consecutive number of values for a given sensor. This value has been determined by using MATLAB to count the highest detected amount of consecutive values in a row in all test scenarios that doesn't contain a fault in the tested sensor. In other words, a check has been made for all sensors that counts the occurrence of consecutive values in all scenarios that doesn't hold a fault for the tested sensor. If the diagnosis algorithm detects a larger number of consecutive values in a row than the modelled number, a stucked sensor has been found.

For the offset mode the work area of the sensors has been determined. Once again this has been done by using MATLAB and all test scenarios that doesn't contain a fault in the tested sensor itself. If a sensor is stationary outside its modelled working area, an offset diagnosis has been found.

This reasoning however does not cover all scenarios. Figure 8 and 9 shows the working area for two of the sensors in the ADAPT. For the first sensor, `E135`, a voltage sensor in the beginning of the system, the working area looks clean and nice. Offsets outside of the working area can easily be detected and alerted. The second sensor, `ST515` - a speed transmitter sensor in the end of the system, it is harder to determine the working area. This is because relays before `ST515` may open resulting in the speed measured by `ST515` dropping to zero. The working area of `ST515` will therefore be [0 960], and offsets withing this range in this sensor will not be discovered by this test quantity. There are other test quantities that might alarm on a offset fault in the area [0 960] and the detection of offset within this range is left to them. We also note that an offset within the working area can not be detected. There are no such cases discovered in the test data [tes09] however.

If the `ST515` sensor gets stuck at zero this can not be detected by the test quantity either. This is because the number of consecutive samples in a row may be very large if a relay before the sensor is open, and the value detected by the sensor goes down to zero without being stuck at zero. In order to address this matter, a lower threshold has been implemented for the sensor, where no stuck or offset faults can be detected below this threshold. For sensors where offset and stuck faults can be detected at zero or negative values(for example the `E135` sensor mentioned above), a negative threshold has been set.

The problem with very large working areas works for both the speed transmitter sensors (ST), phase sensors (XT), flow sensors (FT) and current sensors (IT). The sensors where the check of working areas works very well for temperature sensors (TE), the DC voltage sensors (E) and light sensors (LT).

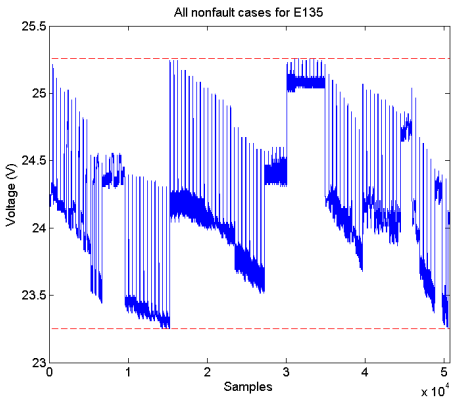



Figure 8: Working area for sensor E135 and all non fault scenarios.

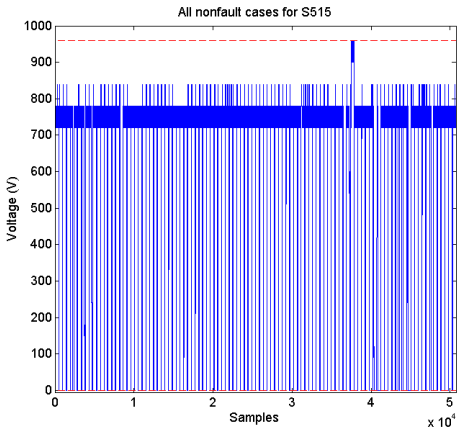



Figure 9: Working area for sensor ST515 and all non fault scenarios.

The sensor test quantity is a strong area for improvement in a future development of the diagnosis algorithm. To get accurate test quantities for all sensors without the use of another sensor is not possible for all types of sensors. To improve this, more time for modelling of the sensors and all relations between the sensors are needed. Time has not allowed this during the development of the diagnosis algorithm, thus the easier version of sensor tests has been implemented. An alternative would be to design new test quantities which could detect and isolate the fault modes of the sensors. Another option might be to work on a test quantity with variable working areas, adapting to the environment around them.

## 4.2 Diagnosis decision logic

To isolate the faults all information has to be considered from the test quantities. The diagnosis candidates will be the minimum sets of faults which 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 diagnosis candidates will be updated using this 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 ([Nyb06]). When the decision algorithm recieves a new sub-diagnosis it updates the diagnosis candidates. A description of the algorithm looks as follows:

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 $D$.

There are some special cases in the system that we have to handle in a proper way with our fault isolation algorithm. The first thing is that we can create test quantities that can say that a component is working properly. In our algorithm we handle all modes in the same way but when we have calculated our diagnosis candidates we don't want these to contain nominal modes. To correct this we remove all nominal modes in the candidates. Because of this it can happen that we get an diagnosis which results in an empty set.

All components have different modes, both nominals and faults, and a component can only be in one mode at the same time. Therefore there can be diagnosis candidates which says that a component is in two different modes at the same time which is a contradiction. The diagnoses which holds contradictions are not possible and are therefore removed. To handle these cases following steps are added to the algorithm:

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 $l$ remove it from $D_{old}$.
   (d) *If $D_j$ contains a contradiction remove it from $D_{old}$.*
   (e) Delete new diagnoses which imply any old diagnose in $D_{old}$ and add the rest to $D$.
3. *Remove all nominal modes in the diagnoses in $D$.*
4. Add the diagnoses in $D_{old}$ to $D$.

This is the adjusted decision algorithm that is implemented in the diagnosis algorithm.

# 5 Analyses

## 5.1 Introduction

The developed test quantities have different detection and isolation abilities of the faults in the electrical power system. In order to verify these and the detectability and isolability for the whole diagnosis system an analysis of their properties will be performed. These analyses are performed in parallel 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.

There are different properties which determines how a test quantity reacts to a specific fault. Different faults need different kinds of test quantities with different kinds of properties. Some faults might be difficult to detect or isolate and requires a long time before a test quantity can give a reliable diagnosis. The time interval could be shortened to get a faster result but then the reliability might deteriorate. There can be faults which are difficult to isolate or detect because of the behaviour of the different faults.

One noticeable aspect is the size of the faults. A test quantity is designed to react to faults but not if there are not any. This requires some kind of thresholds or requirements for the test quantity to react. Because of this there might be faults of small amplitudes which can not be detected by some test quantities. One could lower the thresholds or ease the requirements to be able to detects these faults as well, but at the same time you increase the risk of having false detections. Different test quantities have different thresholds and therefore different detectability properties. Two faults might not be isolable from each other. By designing test quantities with different properties, which react to different sets of faults, a diagnosis system at whole can have isolability and detectability properties enough to detect faults even if a single test quantity can not.

If one assume that there is a system with four faults $F1$, $F2$, $F3$ and $F4$ and a diagnosis system with three test quantities which are able to detects these faults as shown in Table 21. The 'X' represents that the fault can activate the test quantity but as discussed earlier this is not always assured. The only thing one know is that a test quantity does not react to a fault marked by '0'. As can be seen in Table 21, Test1 reacts to faults $F1$

and $F3$, but not $F2$. If this test reacts it can be said that there is a fault either in $F1$ or $F3$. The faults can not be isolated from each other with only this test.

By adding the test quantity Test2, which reacts to fault $F1$ and $F2$, it helps to make a better analysis depending on the reactions of both tests. If both test reacts the explanation can either be that $F1$ has happened or $F2$ and $F3$. If only the single faults are considered then $F1$ can be isolated from $F3$. But if only Test1 reacts, $F3$ can not be the only explanation because Test2 maybe not be as good as Test1 on reacting to $F1$. Therefore with these two tests $F1$ can be isolated from $F3$ but not the opposite.

Table 21: Example: Detectability matrix

| Test Quantity | $F1$ | $F2$ | $F3$ | $F4$ |
|---|---|---|---|---|
| Test1 | X | 0 | X | 0 |
| Test2 | X | X | 0 | 0 |
| Test3 | X | X | X | 0 |

When Test3 is added, which reacts to all three faults, one see that $F3$ still can not be isolated from $F1$. Even if a new test quantity is added one can not be sure that it gives a better isolability or detectability of the diagnosis system. By using a fault detection matrix one can analyse which faults that can be detected but there also have to be made a deeper analysis to check isolability. For this purpose an isolability matrix is used which is easily calculated by help from the detectability matrix.

If you analyse which other faults a specific fault can be isolated from, you get the isolability matrix. If there is an assumption that a specific fault has occurred you can see which test quantities that can react to this fault. Then, by analysing all faults the test quantities can react to, it can be seen if there are any other fault that also reacts to the same combination of test quantities. The isolability matrix is a matrix with the same number of rows and columns as detectable faults and for each row all faults that can not be isolated from the row-specific fault are marked.

Table 22: Example: Isolability matrix

| | $F1$ | $F2$ | $F3$ |
|---|---|---|---|
| $F1$ | X | 0 | 0 |
| $F2$ | X | X | 0 |
| $F3$ | X | 0 | X |

If all faults can be isolated then there will only be 'X' in the diagonal of the isolability matrix. If one continue with the earlier example and create an isolability matrix from the three test quantities results in what you can see in Table 22. As realised before you can see that $F1$ is isolable, from the other faults, but $F2$ and $F3$ can not be isolated from $F1$. From this matrix it can now be seen that it is needed to create test quantities which reacts to $F2$ or $F3$ but not to $F1$ to be able to isolate these faults[NF09].

If you analyse a small system with only a few faults these matrices can easily be created by hand. But if there is a system with a large amount of possible faults and a large amount of test quantities then there is needed some kind of help program or algorithm to create these matrices.

## 5.2 Tools for analysis

To be able to make the detectability and isolability matrices a program was made in MATLAB. The program is coded with MATLAB's object oriented programming syntax.

The code is presented in Appendix. It takes the test quantities as input an auto-generates these matrices for analysis. You can also analyse if a new test quantity add any new information to the diagnosis algorithm or improve the isolability matrix. The information is presented as a graph where all 'X' marks are represented by dots. The MATLAB code is presented in Appendix D.

## 5.3 Detectability

For the developed diagnosis system a detectability matrix for the test quantities has been made. This gives an optimistic estimation of which test quantities that will react when a specific fault enters the system. You can't expect that the test quantities always will react if there is a fault because, like mentioned earlier, the fault maybe not be large enough or if the fault does not make the system leave an expected behaviour. One example is the test quantities that tests if the real valued sensors has an offset. If the offset is outside of the sensors normal working interval a fault can be detected but if the offset still is within the expected interval you can not say if there is an offset or if the system is behaving as expected.

The detectability matrix can only say that one can detect some faults of a certain type, but not promise that they are always detected because of different circumstances stated earlier.

Our developed diagnosis system can detect all fault modes defined for the system. Some faults detectability have a redundancy because several tests can react to the same fault. For example are most of the sensors used by several test quantities so there are several chances that these faults can be detected. But again as stated earlier some faults may not be detected. There are test quantities which controls if the sensors are within their normal working intervals. If not then there is an offset fault detected but if the offset does not make the sensor leave its working interval then the fault is not detected. Faults in the batteries and the inverters are detectable by only one and two respectively per each component which requires a good detectability from these test quantities.

Test quantities which use relays require that the relays are in specific modes for the tests to work. For example the tests which controls if the relay is 'Stuck' compares the voltage sensors before and after the relay. If the relay is closed then the voltage should be the same before and after the relay. If it is not closed then no statement can be made about the relation of the voltage sensors. These tests require that the relays say that they are closed to see if there are any faults. This gives two tests for each relay: one which can react if last command to relay says that it should be closed and one which can react if the relay position sensor says that it should be closed.

A conclusion of the detectability of the diagnosis system can be seen in Figure 10. The diagnosis system holds 315 different test quantities (different possible sub-diagnoses) and can detect 222 different faults which is the total amount for the electrical power system.

## 5.4 Isolability

After the development of the the test quantities and analysis of the detectability of our diagnosis algorithm one can continue with the isolability analysis. There is still a question if the developed test quantities are enough to isolate the faults that can be detected. Just as stated earlier for the detectability matrix the isolability matrix is an optimistic estimation of the properties of the diagnosis system. The isolability matrix says that if all test quantities reacts to their expected faults then we can isolate a fault according to the matrix. If a test quantity is not responding to a fault then the isolability properties

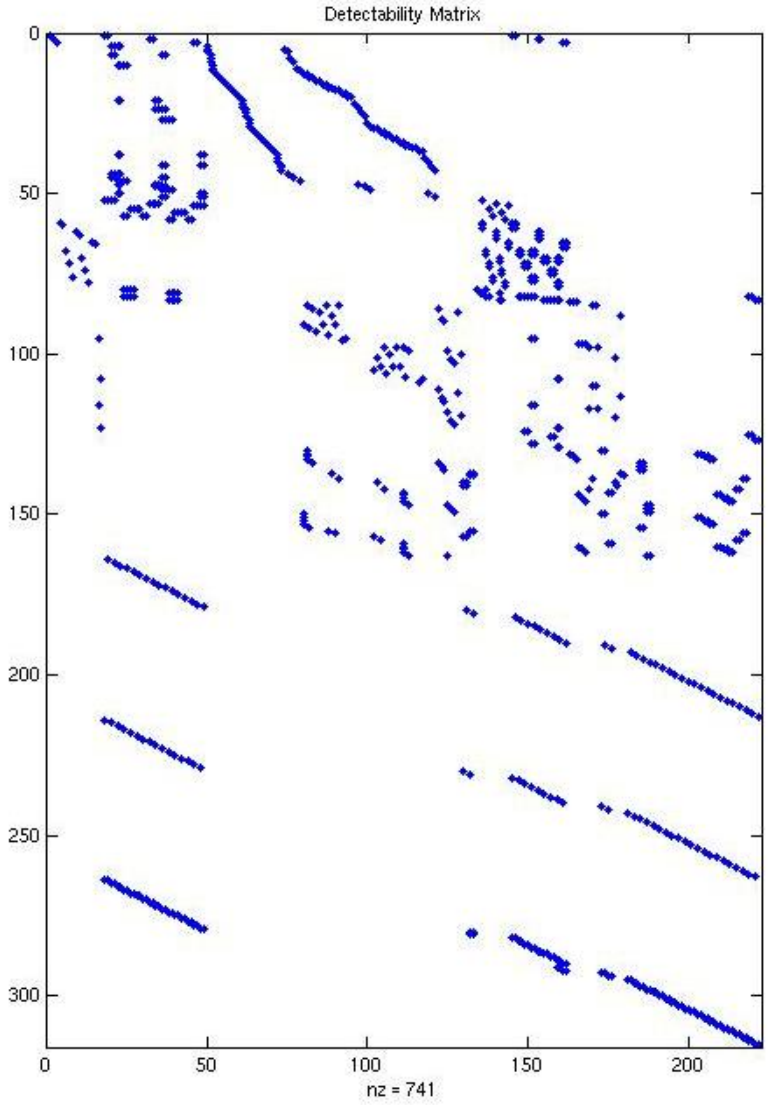



Figure 10: The detectability matrix of the diagnosis system

might not be as good. But one can say that if two faults can not be isolated from each other, according to the isolability matrix, then you can never isolate them with our present diagnosis system.

A conclusion of the isolability of the diagnosis system can be seen in Figure 11. Ideally this $222 \times 222$ isolability matrix should be the identity matrix, since that would imply that all fault were uniquely isolable. However, this is not the case (even though it is close!) and the non-isolable faults can be divided into 9 different groups presented in Table 23 and Table 30. For each of these groups the main problem is that only one test quantity was designed detecting each of the non-isolable fault and an explanation will be given why that must be the case on basis of the model of that component. Note that this analysis is not complete, implementation of further tests is needed to improve properties of the diagnosis algorithm.

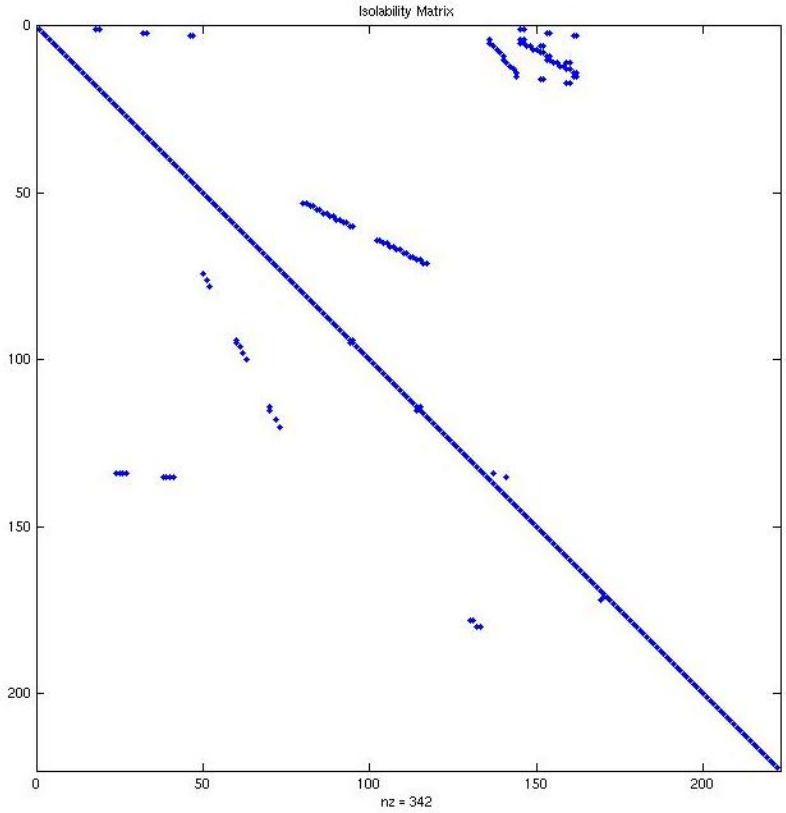



Figure 11: The isolability matrix of the diagnosis system

### 5.4.1 Non-isolable faults in battery

The battery test quantities use the batteries relating voltage and current sensors to compute the internal resistance of the battery. If it is higher than expected, the battery is assumed to be degraded (see Chapter 4.1.1). However, another explanation could be that sensors are not reliable (see Table 23). According to our model, a degraded battery will not affect any other quantities (like the temperature). Therefore no other test can be designed including the degraded battery. Thus it will not be isolable from sensor faults.

Table 23: Non isolable faults in battery

| Fault | Can not be isolated from |
|---|---|
| BAT[1/2/3]:Degraded | E[1/2/3]35:Offset/Stuck, IT[1/2/3]40:Offset/Stuck |

### 5.4.2 Non-isolable faults in the circuit breakers

In the circuit breakers the faults FailedOpen and StuckClosed (StuckClosed only possible for commandable circuit breakers) are not isolable (see Table 24). The problem is that there is only one possible test to make including the faults of the circuit breakers, namely a test using the current sensors. A test using voltage sensors will not be possible, since the circuit breakers have two Nominal modes: Nominal and Tripped and the mode Tripped only depends on the current and not on the voltage. Thus, only a test including the

current sensors can separate the nominal mode Tripped from fault modes. To summarize: the underlying problem is that the two nominal modes Nominal and Tripped in our model of the circuit breakers are represented as one nominal mode.

Table 24: Non-isolable faults in the circuit breakers

| Fault | Can not be isolated from |
|---|---|
| CB[1/2/3]36:FailedOpen/StuckClosed | ISH[1/2/3]36:Stuck, IT[1/2/3]40:Offset/Stuck |
| CB[1/2]62:FailedOpen | ISH[1/2]62:Stuck, IT[1/2][6/8]1:Offset/Stuck |
| CB[1/2]66:FailedOpen | ISH[1/2]66:Stuck, IT[1/2]67:Offset/Stuck |
| CB[1/2]80:FailedOpen | ISH[1/2]80:Stuck, IT[1/2]81:Offset/Stuck |

### 5.4.3 Non-isolable stuck in load relays

In the actuator position sensors of load relays the fault mode Stuck is not isolable from the faulty mode of the relays (see Table 25). The problem is that there are no unique current or voltage sensors for any of these sensors, as for all other relays. Thus, only one test is possible to make including the fault mode Stuck, namely a test comparing the actuator position sensor with the command. Therefore Stuck will not be isolable from relay faults.

Table 25: Non-isolable stuck in load relays

| Fault | Can not be isolated from |
|---|---|
| ESH[1/2]7[0/1/2/3/4/5]:Stuck | EY[1/2]7[0/1/2/3/4/5]:StuckClosed/StuckOpen |
| ESH[183/284]:Stuck | EY[183/284]:StuckClosed/StuckOpen |

### 5.4.4 Non-isolable stuck in load relays without loads

There are also load relays without any loads directly connected to them. Thus, a fault mode in the relay will not affect any current or voltage sensors and only one test quantity including the faults in the actuator position sensor and the relay can be designed, namely a test comparing the actuator with the command. Therefore these faults can not be isolated from each other (see Table 26).

Table 26: Non-isolable stuck in load relays without loads

| Fault | Can not be isolated from |
|---|---|
| ESH[184/283]:Stuck | EY[184/283]:StuckClosed/StuckOpen |
| EY[184/283]:StuckClosed/StuckOpen | ESH[184/283]:Stuck |

### 5.4.5 Non-isolable StuckClosed in relays except load relays

For all other relays (except the load relays) the fault mode StuckClosed can not be isolated from Stuck in the actuator position sensor (see Table 27). When the relay is in mode NominalOpen, there is no model for the current or voltage associated with that relay (see Chapter 3.4). Therefore only one test comparing the command with the actuator can be made including the fault mode StuckClosed when the relay is open. Thus this fault can not be isolated from Stuck in the actuator.

Table 27: Non-isolable StuckClosed in relays except load relays

| Fault | Can not be isolated from |
|---|---|
| EY[1/2/3][41/44]:StuckClosed | ESH[1/2][41/44]A:Stuck |
| EY[1/2]60:StuckClosed | ESH[1/2]60A:Stuck |

### 5.4.6 Non-isolable FailedOff in inverter

The inverter test quantities use the inverters related voltage and current sensors to compute the efficiency of the inverter. If it is higher or lower than expected, the battery is assumed to be FailedOff (see Chapter 4.1.2)). However, another explanation could be that the sensors are not reliable (see Table 28). According to our model, a failed off inverter will not affect any other quantities. Therefore, no other test can be designed including a failed off inverter. Thus it will not be isolable from sensor faults.

Table 28: Non-isolable FailedOff in inverter

| Fault | Can not be isolated from |
|---|---|
| INV[1/2]:FailedOff | E[1/2]6[1/5]:Offset/Stuck, ISH[1/2]62:Stuck |

### 5.4.7 Non-isolable faults in DC loads

According to the Table 29 the fault mode FailedOff for a DC load can not be isolated from the fault modes Offset or Stuck in the corresponding current sensor. Unlike the AC loads where a test using the phase sensors can be designed, only one test can be made for the DC loads using the current sensor. Thus, the mode FailedOff in a DC load can not be isolated from fault in the corresponding current sensor.

Table 29: Non-isolable faults in DC loads

| Fault | Can not be isolated from |
|---|---|
| DC48[2/5]:FailedOff | IT[1/2]81:Offset/Stuck |

### 5.4.8 Non-isolable FailedOff in loads

The tests comparing the admittances of the loads with an admittance change (see 4.1.3), make use of the fact that different loads have different characteristic admittance. However, if two loads have very similar or even the same admittances, an isolation problem will occur. If some of these loads in addition do not have a sensor measuring some other output (like temperature, light, speed etc.) it will not be isolable from other loads with similar/same admittance. That is the reason why the loads in Table 30 are not isolable.

Table 30: Non-isolable FailedOff in loads

| Fault | Can not be isolated from |
|---|---|
| LGT481:FailedOff | LGT411:FailedOff |
| LGT484:FailedOff | LGT410:FailedOff |

### 5.4.9 Non-isolable FlowBlocked in Water Pump

The power characteristic test quantity for the water pump compares the measured water flow with an expected value, and if this value is too small, the water pump is assumed to be in fault mode FlowBlocked (see Chapter 4.1.4). However, another explanation could be that the flow transmitter sensor is not reliable (see Table 31). According to our model, a blocked flow in a water pump not will affect any other quantities (unlike the fault modes Over-/Underspeed in the large fan, where its admittance changes!). Therefore no other test can be designed including the the fault mode FlowBlocked. Thus it will not be isolable from sensor faults.

Table 31: Non-isolable FlowBlocked in Water Pump

| Fault | Can not be isolated from |
|---|---|
| PMP42[0/5]:FlowBlocked | FT52[0/5]:Offset/Stuck |

## 5.5 Robustness

This section contains an analysis of the robustness of the diagnosis program, i.e. how sensitive the different test quantities are to modeling errors and signal noise. After an introduction the analysis is divided into sections discussing one component type at a time.

Since the diagnosis algorithm is never reset during a scenario, all sub-diagnosis statements are in a way remembered. This design of the diagnosis program makes it crucial that the test quantities have a very low false alarm rate, i.e. it is important that they do not send sub-diagnosis statements when there is no fault present. This means that during the design of the test quantities a low false alarm rate was one of the most important demands on each test quantity, and this made the test quantities more robust to model errors and signal noise.

### 5.5.1 Battery

The battery is a dynamic system that depends on many variables, for example temperatures, charge level and output current. With the sample data available and the time at disposal for modeling, it is inevitable that the battery model will be less precise than what is wanted. This undermodeling is evident if you look at Figure 4 where each point is a time-averaged internal resistance calculation, and still the variance is fairly large, especially compared to the small changes in internal resistance when the battery becomes degraded. In the same figure is also the threshold for the internal resistance, which as you can see is some distance away from the modeled internal resistance, implying that with a better model this gap could be reduced. The point however is that the test quantity is designed in such a way that this gap is allowed to be large, in order to make sure that the false alarm rate is low.

### 5.5.2 Inverter

There are two types of inverter test quantities:

The first one checks whether or not the inverter outputs 120 V when it is being fed with 22 V or more. Problems with robustness arises if the inverter actually requires a little bit more than 22 V to output 120 V, or the input voltage is just below the 22 V threshold

but sensor noise makes it seem as thought the input voltage is higher than the threshold. This problem is however easily solved by increasing the threshold, which gives a somewhat lower detection rate but more importantly decreases the chances for false alarms due to signal noise. Hence the uncertainty in the model parameter is taken care of by making sure the parameter is high enough.

The second one looks at the efficiency of the inverter, i.e. compares power input with power output. Figure 5 shows efficiency measurements and model (in the shape of upper and lower thresholds). The measurements in this figure are not averaged, as the efficiency in the test quantity, but they are instead min and max efficiencies taken from the sample data, so they include the effect of sensor noise. With that said it is clear that the thresholds are at a relatively safe distance from the measurements, and an ever safer distance to the averaged efficiency used by the test quantity. It is possible though that the efficiency of the inverter degrades as it gets older, and this is not taken into account in the model used, but this is of course not a problem unless the diagnosis program is used during long periods of time, e.g. years or tenths of years.

### 5.5.3 Load

The admittance tests detect changes at the load bank by comparing new measured values of the current and the phase with the predicted values from the previous iteration. If this difference is sufficiently large, a fault is assumed to have been detected. An algorithm parameter weighs how large this difference must be (based on the variances of signal noise) in order to alarm a detection. Thus, this parameter weighs robustness (to signal noise) against detectability.

Furthermore, all loads have been modelled to behave as a constant impedance. This assumption is more correct for some loads than for others. However, the modeling procedure does allow the user to specify variances for the model parameters, making it possible to reckon with these uncertainties. There is also an algorithm parameter deciding the confidence interval (number of standard deviations) to be used (see Figure 3). Thus, this parameter weighs robustness (to model errors) against isolability.

With these two parameters the test can be made arbitrary robust at expense of detect-/isolability performance.

### 5.5.4 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 temperature sensors for some light bulbs, speed transmitters for some fans, and flow transmitters for the pumps. In the diagnosis algorithm a kalman filter is used observing the power output from the load (see Chapter 4.1.4). A tuning parameter decides the variance of the process noise in these kalman filters. In kalman filters, tuning is a compromise between speed and noise reduction. In the diagnosis algorithm, the mentioned parameter will therefore weigh robustness (to signal noise) against detection/isolation time.

However, for this test no algorithm parameter exists weighting robustness (to model errors) against other performance properties. Therefore the potential robustness properties to model errors is limited.

### 5.5.5 Relay

There are two types of relay test quantities:

The first one compares the relay command with the relay position sensor reading, so the input signals to the test quantity are all boolean, and there are no model parameters. Therefore this test quantity has no robustness problems at all.

The second test quantity looks at the voltage difference across the relay at times when the relay position sensor says that the relay is closed. The voltage difference across a closed relay is so small that it is impossible to separate from the noise on the voltage sensors, and therefore the threshold for determining when the voltage difference is large is limited by the sensor noise. This noise is also rather small, so with the relatively large margin for the threshold that the test quantity uses, good robustness properties are achieved and at the same time the detection rate is kept high, since the threshold is much smaller than the voltage differences for when the relay is faulty.

### 5.5.6 Circuit breaker

There are two types of circuit breaker test quantities:

The first one uses boolean signals, one of which is constructed by comparing the greatest current so far through the circuit breaker to a current rating (the rating states at what current the circuit breaker trips, i.e. opens itself). The only robustness problem with this test quantity is the case in which the current reaches just above the current rating, and the circuit breaker doesn't trip, either because its rating is inaccurate, or because of current sensor noise. Since this case is rather specific, and it is the only case in which the test quantity in question may have bad robustness properties, this test quantity may well be regarded as robust to sensor noise and model errors.

The second circuit breaker test quantity looks at the voltage difference over the circuit breaker. The model used is that the voltage difference lies within a certain range (when the circuit breaker is closed), and although this may be regarded as an unsatisfactory model it works well for detecting when the circuit breaker or any of the used sensors are faulty. This is a result of the fact that the voltage difference over the circuit breaker is small. So even though the model (the upper and lower bounds of the voltage difference range) is rather uncertain it is possible to get good robustness (without loosing much in detection rate) by adding a pretty large margin to the upper and lower bounds.

### 5.5.7 Sensors

The sensor tests detect and isolate Stuck and Offset in sensors. By counting how many samples that are equal in a row Stuck can be detected, and by comparing the sensor value with its expected working area (some) offsets can be detected (for further information, see Chapter 4.1.7).

Both of these tests are sensitive to model errors (with false alarm as a result) and there are no algorithm parameters weighing robustness to such errors against some other performance properties. Therefore, this test in some extent lacks robustness to model errors.

However, robustness to signal noise is a minor problem for this test. Algorithm parameters demanding sensor values to be outside the working area for some samples before detecting offset, guarantee the robustness to signal noise. For the stuck-test, signal noise is even of benefit making the false alarm rate lower. (For noisy signals the risk is lower that many samples are equal even though the sensor is not stucked.)

# 6 Software

In the following section the diagnosis algorithm is described in terms of how the software is implemented. This includes information in terms of structure of the algorithm, files, classes and communication with the DxC framework provided by NASA.

## 6.1 Integration with the DxC Framework

This section covers how the algorithm is integrated into the DxC framework, and how communication between the DxC framework and the diagnosis algorithm.

### 6.1.1 Background

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

It is possible to develop the diagnosis algorithm in any programming language but there are two languages recommended by NASA. These languages are C++ and Java. These languages provides a basic class structure and objects to communicate with the DxC framework provided by NASA. Other languages have to communicate with the DxC framework by using lower level TCP-IP communication.

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

### 6.1.2 Communication with the DxC Framework

The DxC framework takes care of both input to and output from the diagnosis algorithm. Figure 12 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 algorithm with data. The data comes from previously recorded data sets that is available for download from the DCC homepage[dxp09]. Several scenarios with different injected faults are available for testing with the implemented 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 algorithm (through the Scenario Recorder). Briefly described it records the output from the diagnosis algorithm, for example the current error state if there is one. It also evaluates the results. This is done through the last two classes in Figure 12. The results from the scenario is stored in Scenario Results and evaluated and calculated into points that can be used to compare the diagnosis algorithm towards other algorithms in a competition using the Evaluator. How this evaluation is made in a competition is listed in section four of the Diagnostic Challenge Competition Announcement [KNP+09].

All communication made between the modules in Figure 12 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, that is handling all kinds of messages to and from the diagnosis algorithm. The diagnosis algorithm sends two types of messages: a ScenarioStatusData message signaling that the algorithm is ready to process data, and a DiagnosisData containing the faults detected. It is also possible for the diagnosis algorithm to send error messages through the DxC, informing the DxC that something has gone wrong. The diagnosis algorithm also takes three kinds of data types

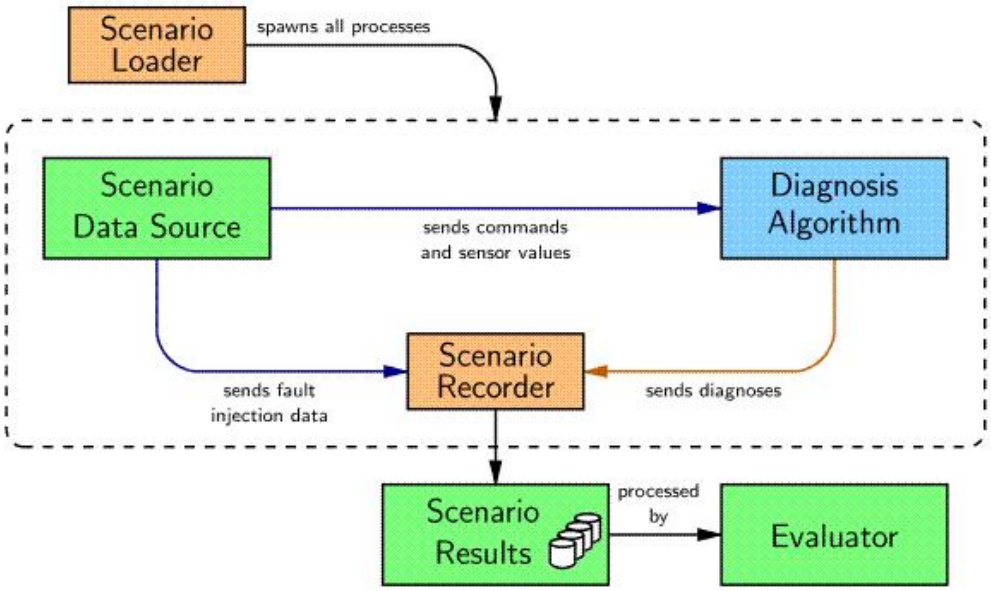



Figure 12: An overview of the ADAPT framework[sys09].

as inputs: SensorData, CommandData, and ScenarioStatusData. DxC will begin sending data once the initial ScenarioStatusData is sent by the diagnosis algorithm.

The different messages are inherited from a parent class called `DxC::DxcData`. The structure can be seen in Figure 13.

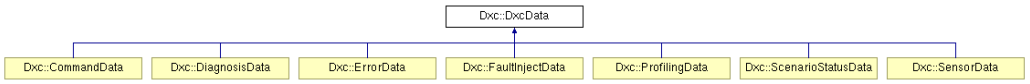



Figure 13: The messages to and from the DxC are inherited by the parent class DxcData as follows.

The data types sent from the DxC are inherited from a class called DxC::Value. Details about internal data storage are described in section 6.2.5. In the appendix B a description of messages sent to and from the DxC are included.

## 6.2 Implementation

This section covers how the diagnosis algorithm looks like in terms of structure and classes.

There is a demand that the developed diagnosis algorithm is generic and flexible to changes in parameters and in sensor configuration. It should be possible to change a components parameters without changing the whole diagnosis algorithm. An overview of how the diagnosis algorithm works can be seen in Figure 14.

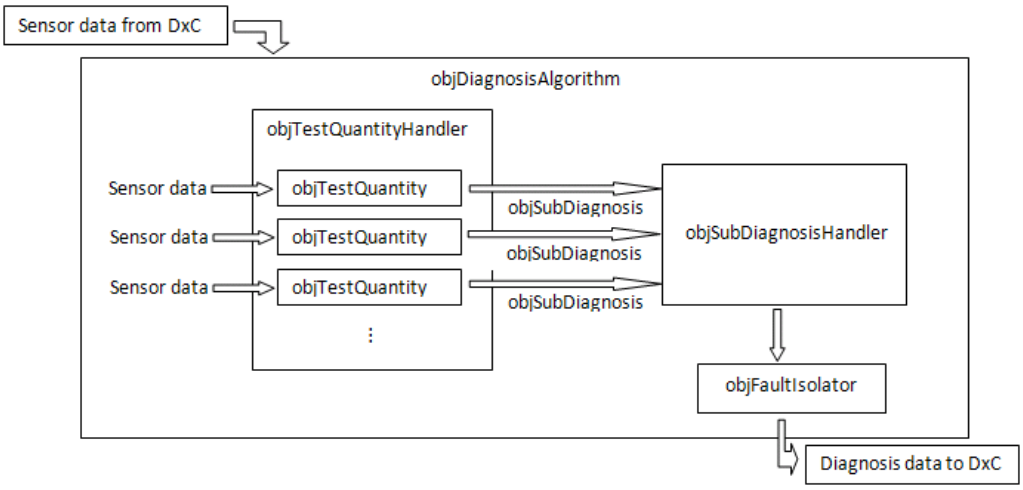



Figure 14: An overview of how the diagnosis algorithm will be implemented.

The diagnosis algorithm can be divided into a few major parts, namely test quantities,a sub-diagnosis handler and a fault isolator. Among with a few 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 sensor and command data and delivers a sub-diagnosis. The sub-diagnosis contains information about what parts that might have a fault. In between the start and finish each test quantity can be formed arbitrary, as long as it delivers the standardized output as a sub-diagnosis. By using a test quantity parent class and subclasses for each type of test quantity one may design proficient and suitable test quantities. Using this object oriented structure of test quantities also increases the flexibility of the diagnosis algorithm as more test quantities can be added without changing the old ones.

Test quantities uses the sensor data they need from the global sensor map and filter the data if needed. What data is needed and what filtering that will be made is based on which test is to be performed. Several tests only uses data from a few sensors, thus making it unwise to load data from all sensors into every test quantity.

The information gathered from all the test quantities is placed in a sub-diagnosis container class, called a sub diagnosis handler. This container stores information from all test quantities. When all test quantities has delivered their results to the container, the container is passed as input to the the fault isolator.

The fault isolator is a decision maker that is taking the sub-diagnoses from all the available test quantities and finds the correct diagnoses as described in section 4. This diagnosis is presented back to the DxC through a message of type DxC::DiagnosisData as mentioned in section 6.1.2. Once a diagnosis has been found (that is not an empty diagnosis), the fault isolator presents this diagnosis to the DxC every sample. If the diagnosis is altered in the fault isolator, the new diagnosis is presented instead. This could be a subject

for optimization as it's not perfectly clear how the DxC prefers it's diagnoses to get the highest competition score in the DCC competition. Once a fault have been detected and is presented to the DxC, this fault is flagged as detected. If there is only one candidate at the lowest dimension it also sends that the fault is isolated.

The fault isolator class does not require a named test quantity to be able to produce a diagnosis, thus increasing flexibility even more. Of course, it might be hard to isolate a diagnosis without certain test quantities, but that is always the case independent of the structure of the diagnosis algorithm. This also means that the diagnosis algorithm can be altered if a sensor is removed from the system. To do this all the test quantities using this sensor is removed from the diagnosis algorithm.

### 6.2.1 Model and algorithm parameters

In order to easily be able to change parameters for a component and levels for weighing contradicting parameters, two separate files for these parameters have been added to the algorithm. Those files are `modelparameters.h` and `algorithmparameters.h`. This section describes briefly the contents of each of these files and how to modify parameters. However, all parameters will not be listed here.

**Model parameters**
The `modelparameters.h` file contains model parameters that have been determined during the modelling phase. These include things as parameters for the internal resistance and parameters for specifying each load's magnitude and phase as described in section 3.3 etc. As these parameters have been decided by modeling each component using MATLAB ™, there should be little need to change those parameters while optimizing a test score. If a component is changed, replaced by another mark for example, some of the parameters have to be recalculated and modified.

**Algorithm parameters**
The `algorithmparameters.h` file contains various parameters used by the FFFDA to weigh contradicting parameters and other common parameters used within the algorithm. These parameters may be things such as thresholds for when to trigger a test quantity or when a load is thought to be in a stationary state. Changing these parameters will change the outcome from running scenarios in different ways. If a threshold is reduced one will get more detections of faults while running a scenario. But one may also experience an increased amount of false negative rates and less detection accuracy as more test quantities will detect faults due to their altered thresholds. If one is to try to increase a competition score without adding any more test quantities, the algorithm parameters may be trimmed for increased performance. This is partly due to the FFFDA have been created to try to minimize false positive rates, as this is only one of the metrics used to determine the test score.

The amount of algorithm parameters is quite large, and it may be confusing to find the right algorithm parameter to tune if one would try to optimize the competition score. This is basically due to each test quantity having it's own section in the `algorithmparameters.h` file, and there is no method to tune them all together. If one would prefer to have more detected faults in expense of a higher false negative rate, one would have to look through all the test quantities to modify each to this new condition. This would be a strong subject for continued development of the diagnosis algorithm, either by scripting the parameters in an effort to lower the algorithm parameters to just a few parameters, or by simply structuring the algorithm parameters better.

### 6.2.2 Initiation of test quantities

As shown in figure 14 there is a container for storing and handling the test quantities, called a test quantity handler. This container also initiates all test quantities that is to be run. Even if the initiation is made inside the test quantity handler a separate file have been made to make it easier to initiate a new test quantity. This is done because the amount of test quantities used is quite high, and creating all of those in a separate file will ease the initiation of new test quantities. The separate file is called `initTestQuantities.cc`. If one is interested in adding new test quantities, the new test quantity would have to be included and instantiated here.

### 6.2.3 Time limits

As described in the DCC specifications there was two time limits that would have to be taken into consideration while creating a diagnosis algorithm. Those were start up time and execution time for the diagnosis algorithm.

According to the DCC specifications the start up time has a maximum limit of 30 seconds. To check that this criteria is met a combination of occular survey and the command `time ./fffda` (in Linux) is used. The FFFDA has a start up time of approximately one second. It is therefore quite far from the limit of 30 seconds, and even if the method for checking the time limit is not the most scientific, this requirement is obviously met.

There was also a requirement on the execution time of each iteration of the diagnosis algorithm on 0.5 seconds. The time for executing each iteration of the diagnosis algorithm depends on how many diagnoses the fault isolator have to process each time. Using the algorithm of a high number of diagnoses clearly takes more time than if the algorithm handles just a few diagnoses. The time limit for execution time is checked by starting and stopping a stop watch before and after each iteration of the diagnosis algorithm. This stop watch is done by the function `clock_gettime` provided by <`time.h`> C library, and gives an accuracy of nanoseconds on the measurement. By doing this and measuring the highest execution time for one iteration in the scenario, the average usually is somewhere around 0.003 seconds for one iteration. This is noticeably smaller than the limit, and the requirement is therefore said to be met. The subject of threading the diagnosis algorithm so that the diagnosis is done separate from the data gathering have therefore not be performed.

### 6.2.4 Class and file structure

In order to use the DxC Framework there is a strong need that the diagnosis algorithm is implemented in a way that meets the criterions of the DxC. This includes placement of the algorithm as well as a special xml file that indicates what tracks the diagnosis algorithm is able to handle. The directory structure of the diagnosis is described in Figure 15. The home directory of the diagnosis algorithm holds `makefile` for compiling the algorithm and the required xml file. The `Bin` directory holds the executable diagnosis algorithm file. The `Build` directory holds the linked objected files (`*.o`). The `Src` directory holds the source code for the algorithm as well as the source code test quantities.

The implementation of the diagnosis algorithm is strongly based on the object-oriented class structure. In the Appendix (sec A) a UML specification of all the essential classes is presented. Table 32 presents a table of all provided classes and files, and a brief description of their usage.

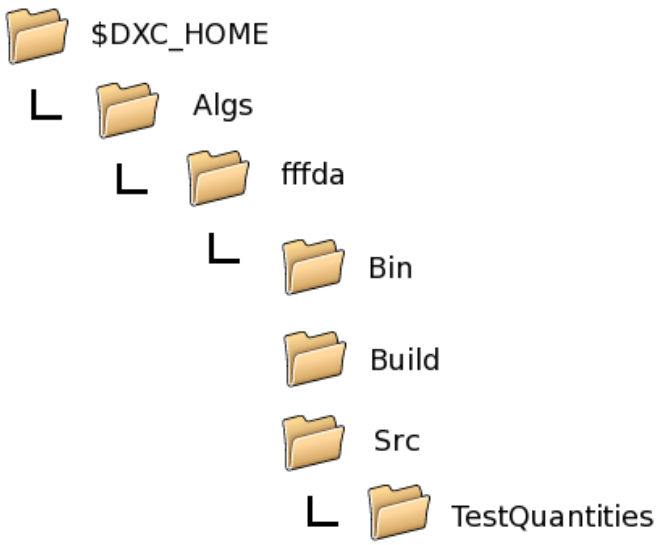



Figure 15: The directory structure of the diagnosis algorithm.

Table 32: Short description of all used classes in the DxC. The file suffix .* states that both header file (.h) and source file (.cc) is present for the class. The files of type `Src` files is placed in `fffda/Src` and files of type `TQ` in `fffda/Src/TestQuantities`.

| **Type** | **Name** | **Description** |
|---|---|---|
| Src | algorithmParameters.h | File that holds all algorithm parameters that can be tuned for altering performance. |
| Src | diagnosisError.* | Error class for throwing exceptions in the algorithm. |
| Src | initSensorProperties.cc | Initiation of the sensor properties. |
| Src | initTestQuantities.cc | Initation of TQ's in the test quantity handler. |
| Src | main.cc | File that includes main function and holds diagnosis algorithm. |
| Src | makefile | File that handles the compiling of the algorithm. |
| Src | modelParameters.h | File that holds all the modelled parameters. |
| Src | objAdmittance.* | Class for calculating admittances. |
| Src | objCallbackHandler.* | Class that handles the callbacks with the DxC. |
| Src | objData.* | Class that handles storage of commands and sensordata. |
| Src | objDiagnosisAlgorithm.* | The main algorithm. Holds containers etc. |
| Src | objFaultIsolator.* | Class that isolates faults. |
| Src | objSensorHandler.* | Class handling sensor levels and attributes. |
| Src | objSubDiagnosis.* | A sub-diagnosis. Output from each TQ. |
| Src | objSubDiagnosisHandler.* | Container class for the sub-diagnoses. |
| Src | objTestQuantity.* | Parent class of TQ's. |
| Src | objTestQuantityHandler.* | Container class for holding TQ's. |
| Src | structs.* | File holding minor structs used everywhere. |
| TQ | objTqAdmittance.* | One of the test quantities for the loads. |
| TQ | objTqBattery.* | The test quantities for the battery. |
| TQ | objTqCircuitBreaker.* | One of the test quantities for the circuit breakers. |
| TQ | objTqCircuitBreakerVoltage.* | One of the test quantities for the circuit breakers. |
| TQ | objTqInverterOnePower.* | One of the test quantities for the inverter. |
| TQ | objTqInverterOneVoltage.* | One of the test quantities for the inverter. |
| TQ | objTqInverterTwoPower.* | One of the test quantities for the inverter. |
| TQ | objTqInverterTwoVoltage.* | One of the test quantities for the inverter. |
| TQ | objTqInverterVoltage.* | One of the test quantities for the inverter. |
| TQ | objTqPower.* | One of the test quantities for the loads. |
| TQ | objTqRelay.* | One of the test quantities for the relays . |
| TQ | objTqRelayCurrent.* | One of the test quantities for the relays . |
| TQ | objTqRelayStuck.* | One of the test quantities for the relays. |
| TQ | objTqSensors.* | One of the test quantities for sensors offset/stucked. |

### 6.2.5 Data storage

As mentioned in section 6.1.2 the software handles three kinds of input messages from the DxC framework. These three are scenario status messages, command messages and sensor data messages.

The diagnosis algorithm will get a callback signal whenever a command data, sensor data or a scenario status is received from the DxC framework. A scenario status signals that the scenario has ended. When sensor- or command data arrives and the corresponding values is stored at the right place in a global sensor- or command map. When a command data is arriving, a series of parameters is set. These can then be used by the test quantities.

The sensordata is received from the DxC as a `typeid(SensorData)` message. This SensorData object contains a SensorValueMap which 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. However, there is no use for the string value, and the integer value only occurs when a sensor is delivering exactly zero. Therefore the use of string values is neglected and the integer values is converted to real values and stored as such.

The command and sensor data is stored in a class called objData. It contains maps for each type of value described above. For example, there objData container holds maps such as

`typedef map<std::string, double> doubleMap typedef map<std::string, bool> boolMap`

for storing real and boolean values, where the string connects the value with the correct sensor. The objData stores a chosen number of values in its map. The number of stored values can be changed by a parameter set in the `algorithmparameters.h` file.

The storage of the input signals received in a `typeid(CommandData)` message is done in the same way as the sensor data, by the use of a objData container and the use of maps.

The handling of the `typeid(ScenarioStatusData)` message is 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.

### 6.2.6 Error handling in the software

Even if this project does not focus on errors and exception handling, some mechanism for throwing and catching errors have been added. Error handling is done by exceptions and the use of throw and catch sentences within the code. There is an outer throw/catch statement around the different callbacks that is handling the running of the test quantities and the fault isolator. If an error is thrown somewhere within these functions, this error is caught and sent to the DxC as a `DxC::Error` data (see 6.1.2). There is also an inner throw/catch statement around the execution of each test quantity. This prevents one malfunctioning test quantity to stop the diagnosis algorithm from executing, and instead displays an error message while running the diagnosis algorithm.

By the addition of a simple exception class to the diagnostic algorithm one gets a simple error handling that still allows an extra degree of debugging possibilities. This error can be thrown wherever needed in the diagnostic algorithm, and then be used for debugging purposes. The exception class in the diagnostic algorithm is very simple. It is inherited

by the runtime_error class as follows:

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

### 6.2.7 Software manual

In order to get a basic understanding of the diagnosis algorithm and an overview of how to change or implement new test quantities in the software, a User Manual[Alm09] is written. It contains a short description of how fo find and change the different component parameters, as well as instructions for installing and integrating the algorithm into the DxC.

# A Class structure

Here follows the definitions of the most central classes in the diagnosis algorithm in UML form.

| **class objData** | |
|---|---|
| **Description:**<br>Object that handles and stores data of different datatypes. This class is used to handle sensor and command data. | |
| **Public Functions:** | |
| | objData()<br>*Constructor* |
| | objData(int newLength)<br>*Overloaded constructor which takes maximum stored data length.* |
| | ~objData()<br>*Destructor* |
| void | insertBool(std::string name, bool value)<br>*Inserts data value into 'name' in std::map structure.* |
| void | insertBool(std::string name, std::string value)<br>*Inserts data value into 'name' in std::map structure.* |
| void | insertBool(std::string name, int value)<br>*Inserts data value into 'name' in std::map structure.* |
| void | insertBool(std::string name, double value)<br>*Inserts data value into 'name' in std::map structure.* |
| std::vector<bool> | getBool(std::string name)<br>*Return the stored values from the specific key 'name'.* |
| std::vector<bool> | getString(std::string name)<br>*Return the stored values from the specific key 'name'.* |
| std::vector<bool> | getInt(std::string name)<br>*Return the stored values from the specific key 'name'.* |
| std::vector<bool> | getDouble(std::string name)<br>*Return the stored values from the specific key 'name'.* |
| void | clearAll(void)<br>*Erases the object of stored data.* |
| **Variables:** | |
| int | dataLength |

| class objDiagnosisAlgorithm | |
|---|---|
| **Description:**<br>Main class that holds the whole algorithm. This class communicates with the DxC Framework | |
| **Public Functions:** | |
| | objDiagnosisAlgorithm()<br>*Constructor* |
| | ~objDiagnosisAlgorithm()<br>*Destructor* |
| void | insertSensorValue(std::string name,bool value)<br>*Overloaded function that stores sensor values depending the data type.* |
| void | insertSensorValue(std::string name,std::string value)<br>*Overloaded function that stores sensor values depending the data type.* |
| void | insertSensorValue(std::string name,int value)<br>*Overloaded function that stores sensor values depending the data type.* |
| void | insertSensorValue(std::string name,double value)<br>*Overloaded function that stores sensor values depending the data type.* |
| void | insertCommandValue(std::string name,bool value)<br>*Overloaded function that stores command values depending the data type.* |
| void | insertCommandValue(std::string name,std::string value)<br>*Overloaded function that stores command values depending the data type.* |
| void | insertCommandValue(std::string name,int value)<br>*Overloaded function that stores command values depending the data type.* |
| void | insertCommandValue(std::string name,double value)<br>*Overloaded function that stores command values depending the data type.* |
| void | runTestQuantities(void)<br>*The functions runs through the test quantity objects.* |
| void | runFaultIsolator(void)<br>*Applies the fault isolator algorithm on the sub-diagnoses collected from the test quantities.* |
| **Variables:** | |
| objTestQuantityHandler*<br>objSubDiagnosisHandler*<br>objFaultIsolator* | testQuantityHandler<br>subDiagnosisHandler<br>faultIsolator |

| **class objTestQuantity** | |
|---|---|
| **Description:**<br>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()<br>*Constructor* |
| | objTestQuantity(std::string name)<br>*Overloaded constructor* |
| | ˜objTestQuantity()<br>*Destructor* |
| virtual vector<objSubDiagnosis*>* | run(objData* sensorData, objData* commandData)<br>*(Virtual) Runs the test quantity object and returns a objSubdiagnosis object* |
| std::string | getTestQuantityName(void)<br>*Return the name of the test quantity* |
| **Variables:** | |
| std::string | testQuantityName |

| class objTestQuantityHandler | |
|---|---|
| **Description:** Container class that holds all objTestQuantity objects. | |
| **Public Functions:** | |
| | objTestQualityHandler()<br>*Constructor* |
| | ˜objTestQualityHandler()<br>*Destructor* |
| void | run(objSubDiagnosisHandler * sdh)<br>*Runs through all test quantities and stores the subdiagnoses in the variable sdh of type objSubdiagnosisHandler*.* |
| void | addTestQuantity(objTestQuantity * newTestQuantity)<br>*Add a objTestQuantity object to the handler.* |
| void | insertSensorValue(std::string name,bool value)<br>*Overloaded function that stores sensor value depending on data type.* |
| void | insertSensorValue(std::string name,std::string value)<br>*Overloaded function that stores sensor value depending on data type.* |
| void | insertSensorValue(std::string name,int value)<br>*Overloaded function that stores sensor value depending on data type.* |
| void | insertSensorValue(std::string name,double value)<br>*Overloaded function that stores sensor value depending on data type.* |
| void | insertCommandValue(std::string name,bool value)<br>*Overloaded function that stores command value depending on data type.* |
| void | insertCommandValue(std::string name,std::string value)<br>*Overloaded function that stores command value depending on data type.* |
| void | insertCommandValue(std::string name,int value)<br>*Overloaded function that stores command value depending on data type.* |
| void | insertCommandValue(std::string name,double value)<br>*Overloaded function that stores command value depending on data type.* |
| objData* | getCommandMap(void)<br>*Returns all stored commands values.* |
| objData* | getSensorMap(void)<br>*Returns all stored sensor values.* |
| **Variables:** | |
| objData | commandMap |
| objData | sensorMap |
| std::vector<objTestQuantity*> | testQuantities |

| class objSubdiagnosis | |
|---|---|
| **Description:** Holds the sub-diagnosis from a test quantity. | |
| **Public Functions:** | |
| | objSubdiagnosis()<br>*Constructor* |
| | ~objSubdiagnosis()<br>*Destructor* |
| void | setTestQuantity(std::string newTestQuantityName)<br>*Set the name of the test quantity.* |
| std::string | getTestQuantityName()<br>*Returns the test quantity name.* |
| void | addFault(Fault* newFault)<br>*Add a fault candidate to the sub-diagnosis.* |
| void | addNonFault(Fault* newNonFault)<br>*Add a non fault candidate to the sub-diagnosis.* |
| vector<Fault*> | getFaults()<br>*Returns the faults as a vector<Fault*>* |
| **Variables:** | |
| string<br>vector<Fault*> | testQuantityName<br>faults |

| class objSubdiagnosisHandler | |
|---|---|
| **Description:** Stores the sub-diagnosis received from the objTestQuantity objects to be used later in the objDiagnosis object. | |
| **Public Functions:** | |
| | objSubdiagnosisHandler()<br>*Constructor* |
| | ~objSubdiagnosisHandler<br>*Destructor* |
| void | addSubdiagnosis(objSubDiagnosis* newSubDiagnosis)<br>*Add a subdiagnosis to the objSubdiagnosisHandler.* |
| std::vector<objSubdiagnosis*> | getSubdiagnoses()<br>*Returns all the objSubdiagnosis objects in a vector.* |
| **Variables:** | |
| vector<objSubdiagnosis*> | subDiagnoses |

| **class objFaultIsolator** | |
|---|---|
| **Description:**<br>Takes the faulty and non faulty components in the subDiagnosisHandler and returns a diagnosis candidate. | |
| **Public Functions:** | |
| | objFaultIsolation()<br><br>~objFaultIsolation()<br><br>run(objSubdiagnosisHandler* subDiagnoses)<br>*Takes all the sub-diagnoses and calculates candidate diagnoses and send a possible candidate to the DxC framework.* |
| void | |
| **Variables:** | |
| - | - |

| **struct Fault** | |
|---|---|
| **Description:**<br>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. |

# B Communication with the DxC

Here follows a description of classes used for communicating with the DxC framework.

| **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 is being sent when the diagnosis algorithm is ready to receive data, and when the diagnosis algorithm signals that it is 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 6.2.6). | |
| **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:**<br>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<br>⟨ Candidate,ltCandidate ⟩ | CandidateSet<br>*Candidate set typedef.* |
| **Public Functions:** | |
| | DiagnosisData (Timestamp timestamp, bool detectionSignal=false, bool isolationSignal=false, const CandidateSet &isolation=CandidateSet(), const std::string &notes="")<br>*Constructor to initialize the DiagnosisData with timestamp.*<br>DiagnosisData (bool detectionSignal=false, bool isolationSignal=false, const CandidateSet &isolation=CandidateSet(), const std::string &notes="")<br>*Constructor to initialize the DiagnosisData with current time as timestamp.* |
| bool | getDetectionSignal () const<br>*True if, according to the diagnosis, the system is believed to be in a faulty state.* |
| bool | getIsolationSignal () const<br>*True if faults have been isolated, i.e. candidates exist.* |
| virtual DiagnosisData * | clone () const<br>*Virtual copy constructor.* |
| virtual ostream & | put (ostream &) const<br>*Prints DxcData in standardized, parseable format.* |
| **Classes:** | |
| struct | Candidate<br>*Maps a set of component IDs to (hypothesized) faulty states.* |

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

# C ADAPT figures

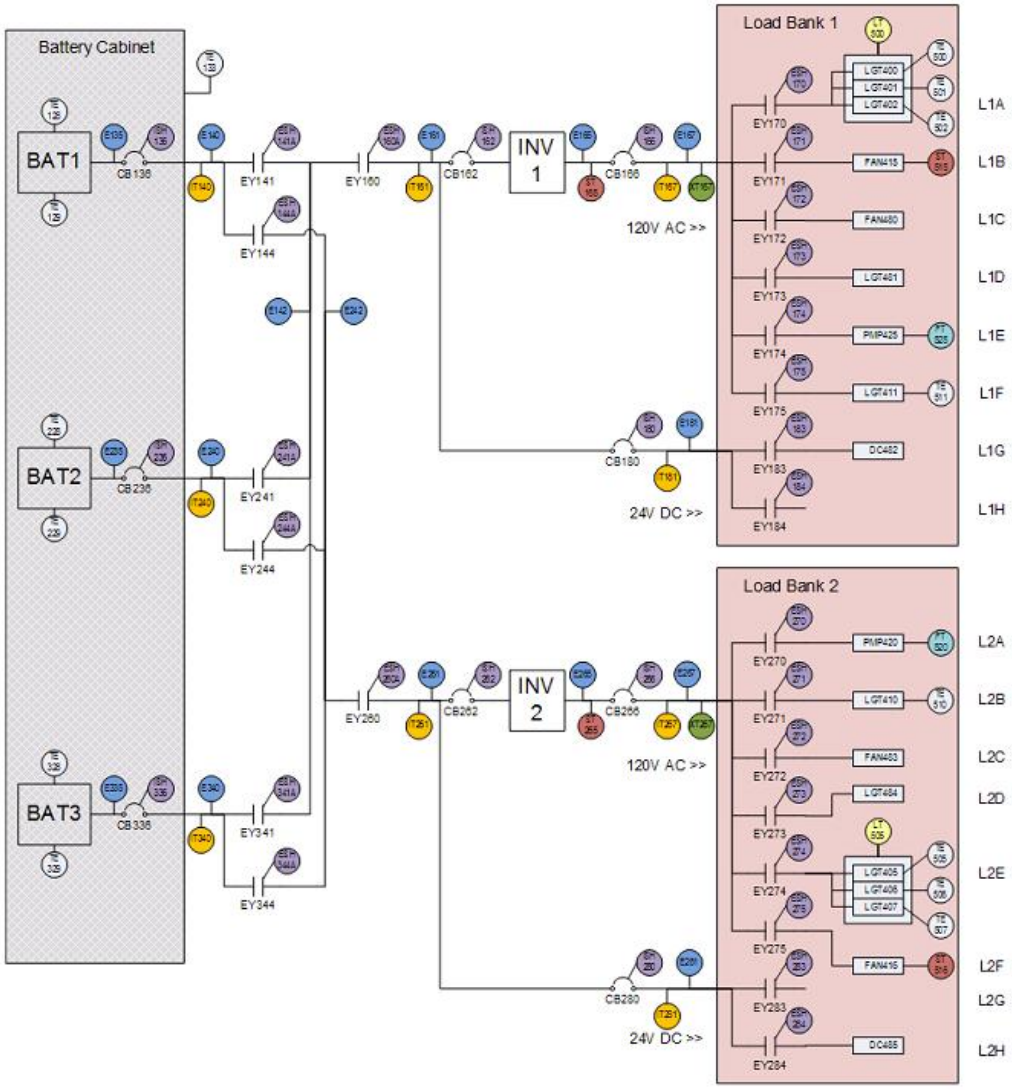



Figure 16: An overview of the ADAPT system and its components[sys09].

Figure 17: A picture of the ADAPT system[sys09].

# D Analysis program

Here is the MATLAB code of the program used during the analyses.

```
%% objAnalysis
classdef objAnalysis < handle
  properties
    TestQuantities = [];
  end % properties
  methods
  %% Constructor
  function obj = objAnalysis()
  end
  %% AddTQ
  % Add modes that test quantity reacts to.
  % 'component', 'mode', 'component', {'modes'}
  function obj = AddTQ(obj, tqName, varargin)

    if mod(nargin-2, 2) ~= 0
        error('Wrong number of arguments.');
    end

    %Save name
    obj.TestQuantities(end+1).Name = tqName;
    %Save modes
    for i = 1:2:nargin-2
      if  ~isfield(obj.TestQuantities(end), 'Faults')
          obj.TestQuantities(end).Faults = [];
      end
      obj.TestQuantities(end).Faults(end+1).Component = varargin(i);
      if iscell(varargin{i+1})
        temp = varargin{i+1};
        for num = 1:length(temp)
          if  ~isfield(obj.TestQuantities(end).Faults(end), 'Modes')
              obj.TestQuantities(end).Faults(end).Modes = {};
          end
          obj.TestQuantities(end).Faults(end).Modes{end+1} = temp{num};
        end
      else
        if  ~isfield(obj.TestQuantities(end).Faults(end), 'Modes')
            obj.TestQuantities(end).Faults(end).Modes = {};
        end
        obj.TestQuantities(end).Faults(end).Modes{end+1} = varargin{i+1};
      end
    end
  end
  %% Create Matrix
  function [A row col] = GetMatrix(obj)
    % Init
    A = [];
    row = {};
    col = {};

    component = [];
    mode = [];

    % Iterate through all test quantities
    for r = 1:length(obj.TestQuantities)
      row{r} = obj.TestQuantities(r).Name;
      % Iterate through all faults
      for c = 1:length(obj.TestQuantities(r).Faults)
        for m = 1:length(obj.TestQuantities(r).Faults(c).Modes)
          % Finns inte felet i listan
          if isempty(col)
            col{end+1} = [obj.TestQuantities(r).Faults(c).Component{1}, ...
```

```
            ':',obj.TestQuantities(r).Faults(c).Modes{m}];
          A(r, end+1) = 1;
        elseif  ¬any(strcmp([obj.TestQuantities(r).Faults(c).Component{1}, ...
            ':',obj.TestQuantities(r).Faults(c).Modes{m}], col))
          col{end+1} = [obj.TestQuantities(r).Faults(c).Component{1}, ...
            ':',obj.TestQuantities(r).Faults(c).Modes{m}];
          A(r, end+1) = 1;
        else
          pos = find(strcmp([obj.TestQuantities(r).Faults(c).Component{1}, ...
            ':',obj.TestQuantities(r).Faults(c).Modes{m}], col));
          A(r, pos) = 1;
        end
      end
    end
  end
  [col corr] = sort(col);
  A = A(:,corr);
end
%% Isolation
% Analyses which faults that can be isolated
function [Q col] = IsolabilityMatrix(obj, varargin)
  [A row col] = obj.GetMatrix();
  [Q col] = isolation(A, row, col);

  % Print the variables that varargin{i} can't be isolated from
  if ¬isempty(varargin)
    for i = 1:length(varargin)
      if isnumeric(varargin{i})
        r = varargin{i};
        fprintf('[%s] :: ', col{r});
      else
        r = find(strcmp(varargin{i}, col));
        fprintf('[%s] :: ', varargin{i});
      end

      if isempty(r)
        error('Fault not found.');
      end

      c = col(find(Q(r,:)));


      if ¬isempty(c)
        c(find(strcmp(col{r},c))) = [];
        for j = 1:length(c)
          fprintf('[%s] ', c{j});
        end
      end
      fprintf('\n');
    end
  end
end
%% Usefull
% Check how much extra information a specific TQ gives
function Usefull(obj, varargin)
  [A row col] = obj.GetMatrix();

  A2 = A;
  row2 = row;
  for tq = 1:length(varargin)
    r = find(strcmp(varargin{tq}, row2));
    A2(r,:) = [];
    row2(r) = [];
  end

  Q = isolation(A, row, col);
```

```
  Q2 = isolation(A2, row2, col);

  figure;
  spy(Q2, 'r');hold on;
  spy(Q, 'b');
end
%% Print info
function Info(obj, varargin)
  [A row col] = obj.GetMatrix();
  figure(1);
  spy(A);
  title('Detectability Matrix');
  if ¬isempty(varargin)
    for i = 1:length(varargin)
      pos = find(strcmp(varargin{i}, col));
      if isempty(pos)
        error('Incorrect fault entered.');
      end
      line([pos-0.5 pos-0.5], [0 length(col)+1], ...
        'Color', 'r', 'LineWidth', 1);
      line([pos+0.5 pos+0.5], [0 length(col)+1], ...
        'Color', 'r', 'LineWidth', 1);
    end
  end

  [Q col] = obj.IsolabilityMatrix();
  figure(2);
  spy(Q);
  title('Isolability Matrix');
  % Mark selected fault
  if ¬isempty(varargin)
    for i = 1:length(varargin)
      pos = find(strcmp(varargin{i}, col));
      if isempty(pos)
          error('Incorrect fault entered.');
      end
      line([0 length(col)+1], [pos-0.5 pos-0.5], ...
        'Color', 'r', 'LineWidth', 1);
      line([0 length(col)+1], [pos+0.5 pos+0.5], ...
        'Color', 'r', 'LineWidth', 1);
      line([pos-0.5 pos-0.5], [0 length(col)+1] , ...
        'Color', 'r', 'LineWidth', 1);
      line([pos+0.5 pos+0.5], [0 length(col)+1], ...
        'Color', 'r', 'LineWidth', 1);
    end
  end

  fprintf('# Test quantities: %d\n', length(row));
  fprintf('# Detectable faults: %d\n', length(col));

  if (length(varargin) == 1)
    fprintf('TQs that can detect "%s": ', varargin{1});

    col_pos = find(strcmp(varargin{1}, col));
    for tq_i = 1:length(obj.TestQuantities)
      tq = obj.TestQuantities(tq_i);
      for(f_i = 1:length(tq.Faults))
        f = tq.Faults(f_i);
        c = f.Component;
        for m_i = 1:length(f.Modes)
          if(strcmp([c{1}, ':', f.Modes{m_i}], varargin{1}) == 1)
            fprintf('%s, ', tq.Name);
          end
        end
      end
    end
```

```
      fprintf('\n');
    end
  end
  %% Print Faults
  function PrintFaults(obj)
      [A row col] = obj.GetMatrix();
      Q = isolation(A, row, col);
      fprintf('Icke isolerbara fel markeras med ''[fel]''.\n');
      % Sortera felen
      col = sort(col);
      component = '';
      for i = 1:length(col)
        [c m] = strtok(strrep(col{i}, ':', ' '));
        if strcmp(component, c) == 0
            component = c;
            fprintf('\n%s:', c);
        end

        if length(find(Q(strcmp(col, col{i}),:))) == 1
            fprintf(' %s;', m(2:end));
        else
            fprintf(' [%s];', m(2:end));
        end
      end
      fprintf('\n');
    end
    %% Print non-Isolable Faults
    % Denna funktion kan verka onödig, men den är snabb som vinden!
    function PrintNIFaults(obj)
      [A row col] = obj.GetMatrix();
      Q = isolation(A, row, col);
      % Sortera felen
      col = sort(col);
      for i = 1:length(col)
        [c m] = strtok(strrep(col{i}, ':', ' '));
        indices = find(Q(strcmp(col, col{i}),:));
        indices = indices(indices ~= i);

        if isempty(indices)
          continue
        end

        fprintf('\n%s:', c);

        m=strrep(m,' ','');
        fprintf('%s --- ', m);

        for k=indices
          fprintf('%s', col{k});
          if (find(indices == k) < length(indices))
            fprintf(', ');
          end
        end
      end
      fprintf('\n');
    end
  end % methods
end


% Does there exist different faults on one component
% then there is a contradiction.
function contr = contradicting(f, col)
  for i = 1:length(f)
    % Get component and mode
    [comp1 mode1] = strtok(strrep(col{f(i)},':', ' '));
```

```
    for j = 1:length(f)
      % Skip if i = j
      if i == j
        continue;
      end
      [comp2 mode2] = strtok(strrep(col{f(j)},':', ' '));
      if strcmp(comp1, comp2)
        if ¬strcmp(mode1, mode2)
          contr = 1;
          return;
        end
      end
    end
  end
  contr = 0;
end


% Extend a on b
function D_new = extend(a, b)
  if iscell(a) || iscell(b)
    error('a and b can''t be a cell');
  end
  D_new = {};
  for i = 1:length(b)
    D_new{end+1} = [a, b(i)];
  end
end


% Does a implies b?
function ok = implies(a, b)
  if iscell(a) || iscell(b)
    error('a and b can''t be a cell');
  end
  % If a is empty then return that a implies b
  if isempty(a)
    ok = 1;
    return;
  end
  % See if any in a is found in b
  for i = 1:length(a)
    if any(a(i) == b)
      ok = 1;
      return;
    end
  end
  ok = 0;
end


% Uses the test quantity matrix to localize which other fault
% a specific fault can not be isolated from.
function [Q col] = isolation(A, row, col)
  % isolability matrix
  Q = zeros(length(col), length(col));
  % Check for each column if the reacting fault is equal with
  % another fault

  sizeA = size(A);
  for f = 1:sizeA(2)
    % Compare with all other columns
    not_isolable_from = [];
    if ¬any(A(:,f))
      continue;
```

```
    end
    for c = 1:sizeA(2)
      if issubset(find(A(:,f)), find(A(:,c)))
        not_isolable_from(end+1) = c;
      end
    end
    if ¬isempty(not_isolable_from)
      Q(f, not_isolable_from) = 1;
    end
  end
end


% Is a a subset of b or a==b
function ok = issubset(a, b)
  if iscell(a) || iscell(b)
    error('a and b can''t be a cell');
  end

  for i = 1:length(a)
    if ¬any(a(i) == b);
        ok = 0;
        return;
    end
  end
  ok = 1;
  return;
end


clear
analys = objAnalysis();

%% Test quantities

analys.AddTQ('BAT1', 'BAT1', {'Degraded'}, 'E135', {'Offset', 'Stuck'}, ...
'IT140', {'Offset', 'Stuck'});
analys.AddTQ('BAT2', 'BAT2', {'Degraded'}, 'E235', {'Offset', 'Stuck'}, ...
'IT240', {'Offset', 'Stuck'});
analys.AddTQ('BAT3', 'BAT3', {'Degraded'}, 'E335', {'Offset', 'Stuck'}, ...
'IT340', {'Offset', 'Stuck'});

analys.AddTQ('ESH141A', 'E140', {'Offset', 'Stuck'}, 'E142', ...
{'Offset', 'Stuck'}, 'ESH141A', 'Stuck');
analys.AddTQ('ESH141A_SC', 'ESH141A', 'Stuck', 'EY141', 'StuckClosed');
analys.AddTQ('ESH141A_SO', 'ESH141A', 'Stuck', 'EY141', 'StuckOpen');

analys.AddTQ('ESH144A', 'E140', {'Offset', 'Stuck'}, 'E242', ...
{'Offset', 'Stuck'}, 'ESH144A', 'Stuck');
analys.AddTQ('ESH144A_SC', 'ESH144A', 'Stuck', 'EY144', 'StuckClosed');
analys.AddTQ('ESH144A_SO', 'ESH144A', 'Stuck', 'EY144', 'StuckOpen');

analys.AddTQ('ESH160A', 'E142', {'Offset', 'Stuck'}, 'E161', ...
{'Offset', 'Stuck'}, 'ESH160A', 'Stuck');
analys.AddTQ('ESH160A_SC', 'ESH160A', 'Stuck', 'EY160', 'StuckClosed');
analys.AddTQ('ESH160A_SO', 'ESH160A', 'Stuck', 'EY160', 'StuckOpen');

analys.AddTQ('ESH170_SC', 'ESH170', 'Stuck', 'EY170', {'StuckClosed','StuckOpen'});
analys.AddTQ('ESH171_SC', 'ESH171', 'Stuck', 'EY171', {'StuckClosed','StuckOpen'});
analys.AddTQ('ESH172_SC', 'ESH172', 'Stuck', 'EY172', {'StuckClosed','StuckOpen'});
analys.AddTQ('ESH173_SC', 'ESH173', 'Stuck', 'EY173', {'StuckClosed','StuckOpen'});
analys.AddTQ('ESH174_SC', 'ESH174', 'Stuck', 'EY174', {'StuckClosed','StuckOpen'});
analys.AddTQ('ESH175_SC', 'ESH175', 'Stuck', 'EY175', {'StuckClosed','StuckOpen'});
analys.AddTQ('ESH183_SC', 'ESH183', 'Stuck', 'EY183', {'StuckClosed','StuckOpen'});
analys.AddTQ('ESH184_SC', 'ESH184', 'Stuck', 'EY184', {'StuckClosed','StuckOpen'});
```

```
analys.AddTQ('ESH241A', 'E240', {'Offset', 'Stuck'}, 'E142', ...
{'Offset', 'Stuck'}, 'ESH241A', 'Stuck');
analys.AddTQ('ESH241A_SC', 'ESH241A', 'Stuck', 'EY241', 'StuckClosed');
analys.AddTQ('ESH241A_SO', 'ESH241A', 'Stuck', 'EY241', 'StuckOpen');

analys.AddTQ('ESH244A', 'E240', {'Offset', 'Stuck'}, 'E242', ...
{'Offset', 'Stuck'}, 'ESH244A', 'Stuck');
analys.AddTQ('ESH244A_SC', 'ESH244A', 'Stuck', 'EY244', 'StuckClosed');
analys.AddTQ('ESH244A_SO', 'ESH244A', 'Stuck', 'EY244', 'StuckOpen');

analys.AddTQ('ESH260A', 'E242', {'Offset', 'Stuck'}, 'E261', ...
{'Offset', 'Stuck'}, 'ESH260A', 'Stuck');
analys.AddTQ('ESH260A_SC', 'ESH260A', 'Stuck', 'EY260', 'StuckClosed');
analys.AddTQ('ESH260A_SO', 'ESH260A', 'Stuck', 'EY260', 'StuckOpen');

analys.AddTQ('ESH270_SC', 'ESH270', 'Stuck', 'EY270', {'StuckClosed','StuckOpen'});
analys.AddTQ('ESH271_SC', 'ESH271', 'Stuck', 'EY271', {'StuckClosed','StuckOpen'});
analys.AddTQ('ESH272_SC', 'ESH272', 'Stuck', 'EY272', {'StuckClosed','StuckOpen'});
analys.AddTQ('ESH273_SC', 'ESH273', 'Stuck', 'EY273', {'StuckClosed','StuckOpen'});
analys.AddTQ('ESH274_SC', 'ESH274', 'Stuck', 'EY274', {'StuckClosed','StuckOpen'});
analys.AddTQ('ESH275_SC', 'ESH275', 'Stuck', 'EY275', {'StuckClosed','StuckOpen'});
analys.AddTQ('ESH283_SC', 'ESH283', 'Stuck', 'EY283', {'StuckClosed','StuckOpen'});
analys.AddTQ('ESH284_SC', 'ESH284', 'Stuck', 'EY284', {'StuckClosed','StuckOpen'});

analys.AddTQ('ESH341A', 'E340', {'Offset', 'Stuck'}, 'E142', ...
{'Offset', 'Stuck'}, 'ESH341A', 'Stuck');
analys.AddTQ('ESH341A_SC', 'ESH341A', 'Stuck', 'EY341', 'StuckClosed');
analys.AddTQ('ESH341A_SO', 'ESH341A', 'Stuck', 'EY341', 'StuckOpen');

analys.AddTQ('ESH344A', 'E340', {'Offset', 'Stuck'}, 'E242', ...
{'Offset', 'Stuck'}, 'ESH344A', 'Stuck');
analys.AddTQ('ESH344A_SC', 'ESH344A', 'Stuck', 'EY344', 'StuckClosed');
analys.AddTQ('ESH344A_SO', 'ESH344A', 'Stuck', 'EY344', 'StuckOpen');

analys.AddTQ('EY141', 'E140', {'Offset', 'Stuck'}, 'E142', ...
{'Offset', 'Stuck'}, 'EY141', 'StuckOpen');
analys.AddTQ('EY144', 'E140', {'Offset', 'Stuck'}, 'E242', ...
{'Offset', 'Stuck'}, 'EY144', 'StuckOpen');
analys.AddTQ('EY160', 'E142', {'Offset', 'Stuck'}, 'E161', ...
{'Offset', 'Stuck'}, 'EY160', 'StuckOpen');
analys.AddTQ('EY241', 'E240', {'Offset', 'Stuck'}, 'E142', ...
{'Offset', 'Stuck'}, 'EY241', 'StuckOpen');
analys.AddTQ('EY244', 'E240', {'Offset', 'Stuck'}, 'E242', ...
{'Offset', 'Stuck'}, 'EY244', 'StuckOpen');
analys.AddTQ('EY260', 'E242', {'Offset', 'Stuck'}, 'E261', ...
{'Offset', 'Stuck'}, 'EY260', 'StuckOpen');
analys.AddTQ('EY341', 'E340', {'Offset', 'Stuck'}, 'E142', ...
{'Offset', 'Stuck'}, 'EY341', 'StuckOpen');
analys.AddTQ('EY344', 'E340', {'Offset', 'Stuck'}, 'E242', ...
{'Offset', 'Stuck'}, 'EY344', 'StuckOpen');

analys.AddTQ('CB136VoltageDrop', 'E135', {'Offset', 'Stuck'}, ...
'E140', {'Offset', 'Stuck'}, 'ISH136', 'Stuck');
analys.AddTQ('CB236VoltageDrop', 'E235', {'Offset', 'Stuck'}, ...
'E240', {'Offset', 'Stuck'}, 'ISH236', 'Stuck');
analys.AddTQ('CB336VoltageDrop', 'E335', {'Offset', 'Stuck'}, ...
'E340', {'Offset', 'Stuck'}, 'ISH336', 'Stuck');
analys.AddTQ('CB166VoltageDrop', 'E165', {'Offset', 'Stuck'}, ...
'E167', {'Offset', 'Stuck'}, 'ISH166', 'Stuck');
analys.AddTQ('CB266VoltageDrop', 'E265', {'Offset', 'Stuck'}, ...
'E267', {'Offset', 'Stuck'}, 'ISH266', 'Stuck');
analys.AddTQ('CB180VoltageDrop', 'E161', {'Offset', 'Stuck'}, ...
'E181', {'Offset', 'Stuck'}, 'ISH180', 'Stuck');
analys.AddTQ('CB280VoltageDrop', 'E261', {'Offset', 'Stuck'}, ...
'E281', {'Offset', 'Stuck'}, 'ISH280', 'Stuck');
```

```
analys.AddTQ('CB136_FO', 'CB136', 'FailedOpen', ...
'ISH136', 'Stuck', 'IT140', {'Offset', 'Stuck'});
analys.AddTQ('CB136_SC', 'CB136', 'StuckClosed', ...
'ISH136', 'Stuck', 'IT140', {'Offset', 'Stuck'});
analys.AddTQ('CB136', 'ISH136', 'Stuck', 'IT140', ...
{'Offset', 'Stuck'});
analys.AddTQ('CB236_FO', 'CB236', 'FailedOpen', ...
'ISH236', 'Stuck', 'IT240', {'Offset', 'Stuck'});
analys.AddTQ('CB236_SC', 'CB236', 'StuckClosed', ...
'ISH236', 'Stuck', 'IT240', {'Offset', 'Stuck'});
analys.AddTQ('CB236', 'ISH236', 'Stuck', 'IT240', ...
{'Offset', 'Stuck'});
analys.AddTQ('CB336_FO', 'CB336', 'FailedOpen', ...
'ISH336', 'Stuck', 'IT340', {'Offset', 'Stuck'});
analys.AddTQ('CB336_SC', 'CB336', 'StuckClosed', ...
'ISH336', 'Stuck', 'IT340', {'Offset', 'Stuck'});
analys.AddTQ('CB336', 'ISH336', 'Stuck', 'IT340', ...
{'Offset', 'Stuck'});
analys.AddTQ('CB162_FO', 'CB162', 'FailedOpen', ...
'ISH162', 'Stuck', 'IT161', {'Offset', 'Stuck'}, ...
'IT181', {'Offset', 'Stuck'});
analys.AddTQ('CB162', 'ISH162', 'Stuck', 'IT161', ...
{'Offset', 'Stuck'}, 'IT181', {'Offset', 'Stuck'});
analys.AddTQ('CB262_FO', 'CB262', 'FailedOpen', ...
'ISH262', 'Stuck', 'IT261', {'Offset', 'Stuck'}, ...
'IT281', {'Offset', 'Stuck'});
analys.AddTQ('CB262', 'ISH262', 'Stuck', 'IT261', ...
{'Offset', 'Stuck'}, 'IT281', {'Offset', 'Stuck'});
analys.AddTQ('CB166_FO', 'CB166', 'FailedOpen', ...
'ISH166', 'Stuck', 'IT167', {'Offset', 'Stuck'});
analys.AddTQ('CB166', 'ISH166', 'Stuck', 'IT167', ...
{'Offset', 'Stuck'});
analys.AddTQ('CB266_FO', 'CB266', 'FailedOpen', ...
'ISH266', 'Stuck', 'IT267', {'Offset', 'Stuck'});
analys.AddTQ('CB266', 'ISH266', 'Stuck', 'IT267', ...
{'Offset', 'Stuck'});
analys.AddTQ('CB180_FO', 'CB180', 'FailedOpen', ...
'ISH180', 'Stuck', 'IT181', {'Offset', 'Stuck'});
analys.AddTQ('CB180', 'ISH180', 'Stuck', 'IT181', ...
{'Offset', 'Stuck'});
analys.AddTQ('CB280_FO', 'CB280', 'FailedOpen', ...
'ISH280', 'Stuck', 'IT281', {'Offset', 'Stuck'});
analys.AddTQ('CB280', 'ISH280', 'Stuck', 'IT281', ...
{'Offset', 'Stuck'});

analys.AddTQ('INV1_V', 'INV1', 'FailedOff', 'E161', {'Offset', 'Stuck'}, ...
'E165', {'Offset', 'Stuck'}, 'ISH162', {'Stuck'});
analys.AddTQ('INV2_V', 'INV2', 'FailedOff', 'E261', {'Offset', 'Stuck'}, ...
'E265', {'Offset', 'Stuck'}, 'ISH262', {'Stuck'});
analys.AddTQ('INV1_P', 'E161', {'Offset', 'Stuck'}, 'E165', ...
{'Offset', 'Stuck'}, 'IT161', {'Offset', 'Stuck'}, 'IT181', ...
{'Offset', 'Stuck'}, 'IT167', {'Offset', 'Stuck'}, 'XT167', ...
{'Offset', 'Stuck'}, 'ISH162', {'Stuck'}, 'ISH166', {'Stuck'});
analys.AddTQ('INV2_P', 'E261', {'Offset', 'Stuck'}, 'E265', ...
{'Offset', 'Stuck'}, 'IT261', {'Offset', 'Stuck'}, 'IT281', ...
{'Offset', 'Stuck'}, 'IT267', {'Offset', 'Stuck'}, 'XT267', ...
{'Offset', 'Stuck'}, 'ISH262', {'Stuck'}, 'ISH266', {'Stuck'});

%Load
analys.AddTQ('LOAD1', 'LGT400', 'FailedOff', 'LGT401', 'FailedOff', ...
'LGT402', 'FailedOff');
analys.AddTQ('LOAD2', 'EY170', 'StuckOpen', 'EY173', 'StuckOpen', ...
'EY175', 'StuckOpen', 'LGT481', 'FailedOff', 'LGT411', 'FailedOff');
analys.AddTQ('LOAD3', 'EY171', 'StuckOpen','FAN415', 'FailedOff');
analys.AddTQ('LOAD4', 'EY172', 'StuckOpen','FAN480','FailedOff');
analys.AddTQ('LOAD5', 'EY174', 'StuckOpen','PMP425','FailedOff');
```

```
analys.AddTQ('LOAD6', 'FAN415','OverSpeed');
analys.AddTQ('LOAD7', 'FAN415','UnderSpeed');

analys.AddTQ('LOAD8', 'EY170', 'StuckClosed', 'EY173', ...
'StuckClosed', 'EY175', 'StuckClosed');
analys.AddTQ('LOAD9', 'EY171', 'StuckClosed');
analys.AddTQ('LOAD10', 'EY172', 'StuckClosed');
analys.AddTQ('LOAD11', 'EY174', 'StuckClosed');

analys.AddTQ('DCLOAD1', 'EY183', 'StuckOpen','DC482', ...
'FailedOff', 'IT181', {'Offset','Stuck'});
analys.AddTQ('DCLOAD2', 'EY183', 'StuckClosed');

analys.AddTQ('LOAD12', 'LGT405', 'FailedOff', 'LGT406', ...
'FailedOff', 'LGT407', 'FailedOff');
analys.AddTQ('LOAD13', 'EY274', 'StuckOpen', 'EY271', ...
'StuckOpen', 'EY273', 'StuckOpen', 'LGT410', 'FailedOff', ...
'LGT484', 'FailedOff');
analys.AddTQ('LOAD14', 'EY275', 'StuckOpen','FAN416', ...
'FailedOff');
analys.AddTQ('LOAD15', 'EY272', 'StuckOpen','FAN483', ...
'FailedOff');
analys.AddTQ('LOAD16', 'EY270', 'StuckOpen','PMP420', ...
'FailedOff');
analys.AddTQ('LOAD17', 'FAN416','OverSpeed');
analys.AddTQ('LOAD18', 'FAN416','UnderSpeed');

analys.AddTQ('LOAD19', 'EY271', 'StuckClosed', 'EY273', ...
'StuckClosed', 'EY274', 'StuckClosed');
analys.AddTQ('LOAD20', 'EY270', 'StuckClosed');
analys.AddTQ('LOAD21', 'EY272', 'StuckClosed');
analys.AddTQ('LOAD22', 'EY275', 'StuckClosed');

analys.AddTQ('DCLOAD3', 'EY284', 'StuckOpen','DC485', ...
'FailedOff', 'IT281', {'Offset','Stuck'});
analys.AddTQ('DCLOAD4', 'EY284', 'StuckClosed');

analys.AddTQ('LOAD2f','LGT481', 'FailedOff', 'LGT411', ...
'FailedOff');
analys.AddTQ('LOAD3f', 'FAN415', 'FailedOff');
analys.AddTQ('LOAD4f', 'FAN480','FailedOff');
analys.AddTQ('LOAD5f', 'PMP425','FailedOff');
analys.AddTQ('LOAD6f', 'FAN415','OverSpeed');
analys.AddTQ('LOAD7f', 'FAN415','UnderSpeed');

analys.AddTQ('DCLOAD1f', 'DC482', 'FailedOff', 'IT181', ...
{'Offset','Stuck'});

analys.AddTQ('LOAD13f', 'LGT410', 'FailedOff', 'LGT484', ...
'FailedOff');
analys.AddTQ('LOAD14f', 'FAN416', 'FailedOff');
analys.AddTQ('LOAD15f', 'FAN483','FailedOff');
analys.AddTQ('LOAD16f', 'PMP420','FailedOff');
analys.AddTQ('LOAD17f', 'FAN416','OverSpeed');
analys.AddTQ('LOAD18f', 'FAN416','UnderSpeed');

analys.AddTQ('DCLOAD3f', 'DC485', 'FailedOff', 'IT281', ...
{'Offset','Stuck'});

analys.AddTQ('LOAD_AC1_IT', 'IT167', {'Offset','Stuck'});
analys.AddTQ('LOAD_AC1_XT', 'XT167', {'Offset','Stuck'});

analys.AddTQ('LOAD_AC2_IT', 'IT267', {'Offset','Stuck'});
analys.AddTQ('LOAD_AC2_XT', 'XT267', {'Offset','Stuck'});

analys.AddTQ('LOAD_DC1_IT', 'IT181', {'Offset','Stuck'});
```

```
analys.AddTQ('LOAD_DC2_IT', 'IT281', {'Offset','Stuck'});


analys.AddTQ('LT500_PCM1','LT500',{'Offset','Stuck'}, ...
'EY170', 'StuckOpen');
analys.AddTQ('TE500_PCM1','TE500',{'Offset','Stuck'}, ...
'EY170', 'StuckOpen', 'LGT400', 'FailedOff');
analys.AddTQ('TE501_PCM1','TE501',{'Offset','Stuck'}, ...
'EY170', 'StuckOpen', 'LGT401', 'FailedOff');
analys.AddTQ('TE502_PCM1','TE502',{'Offset','Stuck'}, ...
'EY170', 'StuckOpen', 'LGT402', 'FailedOff');
analys.AddTQ('ST515_PCM1','ST515',{'Offset','Stuck'}, ...
'EY171', 'StuckOpen', 'FAN415', 'FailedOff');
analys.AddTQ('ST515_OS_PCM1','ST515',{'Offset','Stuck'}, ...
'FAN415', 'OverSpeed');
analys.AddTQ('ST515_US_PCM1','ST515',{'Offset','Stuck'}, ...
'FAN415', 'UnderSpeed');
analys.AddTQ('FT525_PCM1','FT525',{'Offset','Stuck'}, ...
 'EY174', 'StuckOpen', 'PMP425', 'FailedOff');
analys.AddTQ('FT525_PCM1','FT525',{'Offset','Stuck'}, ...
'PMP425', 'FlowBlocked');
analys.AddTQ('TE511_PCM1','TE511',{'Offset','Stuck'}, ...
'EY175', 'StuckOpen','LGT411', 'FailedOff');

analys.AddTQ('FT520_PCM1','FT520',{'Offset','Stuck'}, ...
'EY270', 'StuckOpen', 'PMP420', 'FailedOff');
analys.AddTQ('FT520_PCM1','FT520',{'Offset','Stuck'}, ...
'PMP420', 'FlowBlocked');
analys.AddTQ('TE510_PCM1','TE510',{'Offset','Stuck'}, ...
'EY271', 'StuckOpen','LGT410', 'FailedOff');

analys.AddTQ('LT505_PCM1','LT505',{'Offset','Stuck'}, ...
'EY274', 'StuckOpen');
analys.AddTQ('TE505_PCM1','TE505',{'Offset','Stuck'}, ...
'EY274', 'StuckOpen', 'LGT405', 'FailedOff');
analys.AddTQ('TE506_PCM1','TE506',{'Offset','Stuck'}, ...
'EY274', 'StuckOpen', 'LGT406', 'FailedOff');
analys.AddTQ('TE507_PCM1','TE507',{'Offset','Stuck'}, ...
'EY274', 'StuckOpen', 'LGT407', 'FailedOff');

analys.AddTQ('ST516_PCM1','ST516',{'Offset','Stuck'}, ...
'EY275', 'StuckOpen', 'FAN416', 'FailedOff');

analys.AddTQ('ST516_OS_PCM1','ST516',{'Offset','Stuck'}, ...
'FAN416', 'OverSpeed');
analys.AddTQ('ST516_US_PCM1','ST516',{'Offset','Stuck'}, ...
'FAN416', 'UnderSpeed');

analys.AddTQ('LT500_PCM0','LT500',{'Offset','Stuck'}, ...
'EY170', 'StuckClosed');
analys.AddTQ('TE500_PCM0','TE500',{'Offset','Stuck'}, ...
'EY170', 'StuckClosed');
analys.AddTQ('TE501_PCM0','TE501',{'Offset','Stuck'}, ...
'EY170', 'StuckClosed');
analys.AddTQ('TE502_PCM0','TE502',{'Offset','Stuck'}, ...
'EY170', 'StuckClosed');
analys.AddTQ('ST515_PCM0','ST515',{'Offset','Stuck'}, ...
'EY171', 'StuckClosed');
analys.AddTQ('FT525_PCM0','FT525',{'Offset','Stuck'}, ...
'EY174', 'StuckClosed');

analys.AddTQ('TE511_PCM0','TE511',{'Offset','Stuck'}, ...
'EY175', 'StuckClosed');
```

```
analys.AddTQ('FT520_PCM0','FT520',{'Offset','Stuck'}, ...
'EY270', 'StuckClosed');

analys.AddTQ('TE510_PCM0','TE510',{'Offset','Stuck'}, ...
'EY271', 'StuckClosed');

analys.AddTQ('LT505_PCM0','LT505',{'Offset','Stuck'}, ...
'EY274', 'StuckOpen');
analys.AddTQ('TE505_PCM0','TE505',{'Offset','Stuck'}, ...
  'EY274', 'StuckOpen', 'LGT405', 'FailedOff');
analys.AddTQ('TE506_PCM0','TE506',{'Offset','Stuck'}, ...
  'EY274', 'StuckOpen', 'LGT406', 'FailedOff');
analys.AddTQ('TE507_PCM0','TE507',{'Offset','Stuck'}, ...
  'EY274', 'StuckOpen', 'LGT407', 'FailedOff');

analys.AddTQ('ST516_PCM0','ST516',{'Offset','Stuck'}, ...
  'EY275', 'StuckOpen', 'FAN416', 'FailedOff');

% Sensor TQ'S
analys.AddTQ('E135_S','E135','Stuck');
analys.AddTQ('E140_S','E140','Stuck');
analys.AddTQ('E142_S','E142','Stuck');
analys.AddTQ('E161_S','E161','Stuck');
analys.AddTQ('E165_S','E165','Stuck');
analys.AddTQ('E167_S','E167','Stuck');
analys.AddTQ('E181_S','E181','Stuck');
analys.AddTQ('E235_S','E235','Stuck');
analys.AddTQ('E240_S','E240','Stuck');
analys.AddTQ('E242_S','E242','Stuck');
analys.AddTQ('E261_S','E261','Stuck');
analys.AddTQ('E265_S','E265','Stuck');
analys.AddTQ('E267_S','E267','Stuck');
analys.AddTQ('E281_S','E281','Stuck');
analys.AddTQ('E335_S','E335','Stuck');
analys.AddTQ('E340_S','E340','Stuck');
analys.AddTQ('FT520_S','FT520','Stuck');
analys.AddTQ('FT525_S','FT525','Stuck');
analys.AddTQ('IT140_S','IT140','Stuck');
analys.AddTQ('IT161_S','IT161','Stuck');
analys.AddTQ('IT167_S','IT167','Stuck');
analys.AddTQ('IT181_S','IT181','Stuck');
analys.AddTQ('IT240_S','IT240','Stuck');
analys.AddTQ('IT261_S','IT261','Stuck');
analys.AddTQ('IT267_S','IT267','Stuck');
analys.AddTQ('IT281_S','IT281','Stuck');
analys.AddTQ('IT340_S','IT340','Stuck');
analys.AddTQ('LT500_S','LT500','Stuck');
analys.AddTQ('LT505_S','LT505','Stuck');
analys.AddTQ('ST165_S','ST165','Stuck');
analys.AddTQ('ST265_S','ST265','Stuck');
analys.AddTQ('ST515_S','ST515','Stuck');
analys.AddTQ('ST516_S','ST516','Stuck');
analys.AddTQ('TE128_S','TE128','Stuck');
analys.AddTQ('TE129_S','TE129','Stuck');
analys.AddTQ('TE133_S','TE133','Stuck');
analys.AddTQ('TE228_S','TE228','Stuck');
analys.AddTQ('TE229_S','TE229','Stuck');
analys.AddTQ('TE328_S','TE328','Stuck');
analys.AddTQ('TE329_S','TE329','Stuck');
analys.AddTQ('TE500_S','TE500','Stuck');
analys.AddTQ('TE501_S','TE501','Stuck');
analys.AddTQ('TE502_S','TE502','Stuck');
analys.AddTQ('TE505_S','TE505','Stuck');
analys.AddTQ('TE506_S','TE506','Stuck');
analys.AddTQ('TE507_S','TE507','Stuck');
analys.AddTQ('TE510_S','TE510','Stuck');
```

```
analys.AddTQ('TE511_S','TE511','Stuck');
analys.AddTQ('XT167_S','XT167','Stuck');
analys.AddTQ('XT276_S','XT267','Stuck');


analys.AddTQ('E135_O','E135','Offset');
analys.AddTQ('E140_O','E140','Offset');
analys.AddTQ('E142_O','E142','Offset');
analys.AddTQ('E161_O','E161','Offset');
analys.AddTQ('E165_O','E165','Offset');
analys.AddTQ('E167_O','E167','Offset');
analys.AddTQ('E181_O','E181','Offset');
analys.AddTQ('E235_O','E235','Offset');
analys.AddTQ('E240_O','E240','Offset');
analys.AddTQ('E242_O','E242','Offset');
analys.AddTQ('E261_O','E261','Offset');
analys.AddTQ('E265_O','E265','Offset');
analys.AddTQ('E267_O','E267','Offset');
analys.AddTQ('E281_O','E281','Offset');
analys.AddTQ('E335_O','E335','Offset');
analys.AddTQ('E340_O','E340','Offset');
analys.AddTQ('FT520_O','FT520','Offset');
analys.AddTQ('FT525_O','FT525','Offset');
analys.AddTQ('IT140_O','IT140','Offset');
analys.AddTQ('IT161_O','IT161','Offset');
analys.AddTQ('IT167_O','IT167','Offset');
analys.AddTQ('IT181_O','IT181','Offset');
analys.AddTQ('IT240_O','IT240','Offset');
analys.AddTQ('IT261_O','IT261','Offset');
analys.AddTQ('IT267_O','IT267','Offset');
analys.AddTQ('IT281_O','IT281','Offset');
analys.AddTQ('IT340_O','IT340','Offset');
analys.AddTQ('LT500_O','LT500','Offset');
analys.AddTQ('LT505_O','LT505','Offset');
analys.AddTQ('ST165_O','ST165','Offset');
analys.AddTQ('ST265_O','ST265','Offset');
analys.AddTQ('ST515_O','ST515','Offset');
analys.AddTQ('ST516_O','ST516','Offset');
analys.AddTQ('TE128_O','TE128','Offset');
analys.AddTQ('TE129_O','TE129','Offset');
analys.AddTQ('TE133_O','TE133','Offset');
analys.AddTQ('TE228_O','TE228','Offset');
analys.AddTQ('TE229_O','TE229','Offset');
analys.AddTQ('TE328_O','TE328','Offset');
analys.AddTQ('TE329_O','TE329','Offset');
analys.AddTQ('TE500_O','TE500','Offset');
analys.AddTQ('TE501_O','TE501','Offset');
analys.AddTQ('TE502_O','TE502','Offset');
analys.AddTQ('TE505_O','TE505','Offset');
analys.AddTQ('TE506_O','TE506','Offset');
analys.AddTQ('TE507_O','TE507','Offset');
analys.AddTQ('TE510_O','TE510','Offset');
analys.AddTQ('TE511_O','TE511','Offset');
analys.AddTQ('XT167_O','XT167','Offset');
analys.AddTQ('XT276_O','XT267','Offset');
analys.AddTQ('E135_SO','E135',{'Stuck','Offset'});
analys.AddTQ('E140_SO','E140',{'Stuck','Offset'});
analys.AddTQ('E142_SO','E142',{'Stuck','Offset'});
analys.AddTQ('E161_SO','E161',{'Stuck','Offset'});
analys.AddTQ('E165_SO','E165',{'Stuck','Offset'});
analys.AddTQ('E167_SO','E167',{'Stuck','Offset'});
analys.AddTQ('E181_SO','E181',{'Stuck','Offset'});
analys.AddTQ('E235_SO','E235',{'Stuck','Offset'});
analys.AddTQ('E240_SO','E240',{'Stuck','Offset'});
analys.AddTQ('E242_SO','E242',{'Stuck','Offset'});
analys.AddTQ('E261_SO','E261',{'Stuck','Offset'});
```

```
analys.AddTQ('E265_SO','E265',{'Stuck','Offset'});
analys.AddTQ('E267_SO','E267',{'Stuck','Offset'});
analys.AddTQ('E281_SO','E281',{'Stuck','Offset'});
analys.AddTQ('E335_SO','E335',{'Stuck','Offset'});
analys.AddTQ('E340_SO','E340',{'Stuck','Offset'});
analys.AddTQ('FT525_SO','FT525',{'Stuck','Offset'});
analys.AddTQ('FT525_SO','FT525',{'Stuck','Offset'});
analys.AddTQ('IT140_SO','IT140',{'Stuck','Offset'});
analys.AddTQ('IT161_SO','IT161',{'Stuck','Offset'});
analys.AddTQ('IT167_SO','IT167',{'Stuck','Offset'});
analys.AddTQ('IT181_SO','IT181',{'Stuck','Offset'});
analys.AddTQ('IT240_SO','IT240',{'Stuck','Offset'});
analys.AddTQ('IT261_SO','IT261',{'Stuck','Offset'});
analys.AddTQ('IT267_SO','IT267',{'Stuck','Offset'});
analys.AddTQ('IT281_SO','IT281',{'Stuck','Offset'});
analys.AddTQ('IT340_SO','IT340',{'Stuck','Offset'});
analys.AddTQ('IT281_SO','IT281',{'Stuck','Offset'});
analys.AddTQ('IT340_SO','IT340',{'Stuck','Offset'});
analys.AddTQ('LT500_SO','LT500',{'Stuck','Offset'});
analys.AddTQ('LT505_SO','LT505',{'Stuck','Offset'});
analys.AddTQ('ST165_SO','ST165',{'Stuck','Offset'});
analys.AddTQ('ST265_SO','ST265',{'Stuck','Offset'});
analys.AddTQ('ST515_SO','ST515',{'Stuck','Offset'});
analys.AddTQ('ST516_SO','ST516',{'Stuck','Offset'});
analys.AddTQ('TE128_SO','TE128',{'Stuck','Offset'});
analys.AddTQ('TE129_SO','TE129',{'Stuck','Offset'});
analys.AddTQ('TE133_SO','TE133',{'Stuck','Offset'});
analys.AddTQ('TE228_SO','TE228',{'Stuck','Offset'});
analys.AddTQ('TE229_SO','TE229',{'Stuck','Offset'});
analys.AddTQ('TE328_SO','TE328',{'Stuck','Offset'});
analys.AddTQ('TE329_SO','TE329',{'Stuck','Offset'});
analys.AddTQ('TE500_SO','TE500',{'Stuck','Offset'});
analys.AddTQ('TE501_SO','TE501',{'Stuck','Offset'});
analys.AddTQ('TE502_SO','TE502',{'Stuck','Offset'});
analys.AddTQ('TE505_SO','TE505',{'Stuck','Offset'});
analys.AddTQ('TE506_SO','TE506',{'Stuck','Offset'});
analys.AddTQ('TE507_SO','TE507',{'Stuck','Offset'});
analys.AddTQ('TE510_SO','TE510',{'Stuck','Offset'});
analys.AddTQ('TE511_SO','TE511',{'Stuck','Offset'});
analys.AddTQ('XT167_SO','XT167',{'Stuck','Offset'});
analys.AddTQ('XT276_SO','XT267',{'Stuck','Offset'});

analys.Info();
```

# References

[Alm09] Erik Almqvist. User manual, 2009.

[dxp09] http://www.dx-competition.org/, september 2009.

[KNP+09] Tolga Kurtoglu, Sriram Narasimhan, Scott Poll, David Garcia, Lucas Kuhn, Johan de Kleer, and Alexander Feldman. https://dashlink.arc.nasa.gov/static/dashlink/media/topic/dxc09_announcement.pdf, september 2009.

[NF09] Mattias Nyberg and Erik Frisk. Model based diagnosis of technical processes, October 2009.

[Nyb06] Mattias Nyberg. A fault isolation algorithm for the case of mutiple faults and multiple fault types. In *Proceedings of IFAC Safeprocess'06*, 2006.

[sys09] https://dashlink.arc.nasa.gov/static/dashlink/media/topic/adaptdxc_2008-12-09.zip, september 2009.

[tes09] https://dashlink.arc.nasa.gov/static/dashlink/media/topic/adapttier2data_2009-02-14.zip, september 2009.

[xan09] http://www.megahz.com/specimages/statpower/prosinepdf, october 2009.