Institutionen för systemteknik
Department of Electrical Engineering

Examensarbete

Modelling for diagnosis in Modelica - implementation and analysis

Examensarbete utfört i Vehicular Systems
vid Tekniska högskolan i Linköping
av

Olof Bäck

LITH-ISY-EX--08/4118--SE

Linköping 2008

Linköpings universitet
TEKNISKA HÖGSKOLAN

Department of Electrical Engineering
Linköpings universitet
SE-581 83 Linköping, Sweden

Linköpings tekniska högskola
Linköpings universitet
581 83 Linköping

Modelling for diagnosis in Modelica - implementation and analysis

Examensarbete utfört i Vehicular Systems
vid Tekniska högskolan i Linköping
av

Olof Bäck

LITH-ISY-EX--08/4118--SE

Handledare: **Erik Frisk**
ISY, Linköpings universitet

Examinator: **Erik Frisk**
ISY, Linköpings universitet

Linköping, 28 May, 2008

Avdelning, Institution Division, Department Division of Vehicular Systems Department of Electrical Engineering Linköpings universitet SE-581 83 Linköping, Sweden	**Datum** Date 2008-05-28

Språk Language ☐ Svenska/Swedish ☒ Engelska/English ☐ ______	**Rapporttyp** Report category ☐ Licentiatavhandling ☒ Examensarbete ☐ C-uppsats ☐ D-uppsats ☐ Övrig rapport ☐ ______	**ISBN** — **ISRN** LITH-ISY-EX--08/4118--SE **Serietitel och serienummer** Title of series, numbering **ISSN** —

URL för elektronisk version

http://www.vehicular.isy.liu.se

http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-12169

Titel
Title

Modellering för diagnos i Modelica - implementation och analys

Modelling for diagnosis in Modelica - implementation and analysis

Författare
Author

Olof Bäck

Sammanfattning
Abstract

Technical systems of today are often complex and integrated. To maintain operational functionality and security it is sometimes necessary to have a surveillance system which can detect a fault in an early stage. The device that detects and locates the faulty component is called a diagnosis system. There are several different approaches to fault diagnosis, this study focus on a part of technical diagnosis that uses a model of the system for assistance to make the diagnosis. In the creation of the model of the system computer tools can be used, in this thesis it is investigated if one such software tool is practical to use in the building of the model of the system.

One aspect of technical diagnosis based on models is the creation of a model of the process in question with support for faults, a fault model. In this thesis, Modelica is used to create fault models. The models produced from Modelica are then analyzed by some existing diagnosis algorithms.

The approach of designing fault models in Modelica and then exporting the corresponding system and perform diagnosis analysis on it in MATLAB is considered feasible. But the process falls a bit short of the aim of a automatic model building and and diagnosis analysis procedure.

If the results from this thesis are to be used in the future will depend on if the freedom that Modelica gives in designing fault models are deemed worthwhile to accommodate so that the results from the modeling suits the diagnosis analysis. One way to do ease this transformation of data is to restrict the freedom of model designs in Modelica but then some of the benefit that Modelica brings is negated. It is shown here however that it it possible to design fault models in Modelica and then use the results to analyze the models regarding sensor placement and test design.

Nyckelord
Keywords Diagnos, Fault modelling

Abstract

Technical systems of today are often complex and integrated. To maintain operational functionality and security it is sometimes necessary to have a surveillance system which can detect a fault in an early stage. The device that detects and locates the faulty component is called a diagnosis system. There are several different approaches to fault diagnosis, this study focus on a part of technical diagnosis that uses a model of the system for assistance to make the diagnosis. In the creation of the model of the system computer tools can be used, in this thesis it is investigated if one such software tool is practical to use in the building of the model of the system.

One aspect of technical diagnosis based on models is the creation of a model of the process in question with support for faults, a fault model. In this thesis, Modelica is used to create fault models. The models produced from Modelica are then analyzed by some existing diagnosis algorithms.

The approach of designing fault models in Modelica and then exporting the corresponding system and perform diagnosis analysis on it in Matlab is considered feasible. But the process falls a bit short of the aim of a automatic model building and and diagnosis analysis procedure.

If the results from this thesis are to be used in the future will depend on if the freedom that Modelica gives in designing fault models are deemed worthwhile to accommodate so that the results from the modeling suits the diagnosis analysis. One way to do ease this transformation of data is to restrict the freedom of model designs in Modelica but then some of the benefit that Modelica brings is negated. It is shown here however that it it possible to design fault models in Modelica and then use the results to analyze the models regarding sensor placement and test design.

Acknowledgments

I would like to thank my supervisor **Erik Frisk** for his support and understanding during the work with this thesis. Thanks also to **Erik Frisk**, **Mattias Krysander** and **Jan Åslund** at Vehicular Systems for the use and help with the different diagnosis analysis algorithms used in this thesis. A big thank you to **Adrian Pop** at the Institution of Computer Science for all the help regarding big and small issues in OpenModelica that he has helped me to sort out even though he were in his final stages of his Ph.D. Thank you also **Erik** and **Pär**, my colleagues in the project room for all the fun discussions and anecdotes about everything from rings to fire.

Contents

Chapter 1

Introduction

Technical systems of today are often complex and integrated. If a fault occurs, the consequences can be severe both for the system itself and its surroundings. To maintain operational functionality and security it is sometimes necessary to have a surveillance system which can detect a fault at an early stage. The device that detects and locates the faulty component is called a diagnosis system. There are several different approaches to fault diagnosis, this study focus on a part of technical diagnosis that uses a model of the system for assistance to make the diagnosis. The model is tested against observations from the physical process to determine if the behaviour of the system is consistent with the expected behaviour. In the creation of the model of the system computer tools can be used, in this thesis it is investigated if one such software tool is practical to use in the building of the model of the system.

One aspect of technical diagnosis based on models is the creation of a model of the process in question with support for faults, a fault model. In this thesis, Modelica is used to create fault models. The models produced from Modelica are then analyzed by some existing diagnosis algorithms. The viability of the approach to use Modelica to model fault models and the different benefits and problems that arises are discussed.

OpenModelica is a programming environment with compiler, for the programming language Modelica, that is developed at PELAB, Programming Environments Laboratory, at, IDA, the Institution of Computer Sience, at LIU, Linköpings University. OpenModelica is opensorce.

1.1 Objective

The objective of this study is twofold. The first objective is to design a general way to model faulty components. Using these components, a model of a system can be assembled that will describe the total behavioral of the object, including when it is broken (within limits). These new models for faulty components should be standardised and compatible with the normal component library so that it is easy to implement new diagnosis-functionality in existing models. The next part is to use the data, i.e. the equations, from these models to be able to do diagnosis analysis and synthesis. Analysis should include test design and sensor placement. The whole process from model to analysis result should be as automatized as possible, with little need for the user to become involved. The feasibility of the whole approach with fault modeling in Modelica and conversion of the data to a format that MATLAB can handle should also be evaluated.

1.2 Thesis Outline

In Chapter 2 an introduction to diagnosis theory, including modeled based diagnosis is given. In Chapter 3, fault modeling in Modelica is discussed. The issue of how the data that describe the fault model is handled and represented is explained in Chapter 4. The diagnos analysis of the fault models is done in Chapter 5 and Chapter 6. In Chapter 7 an example of an hybrid drive line is put through the different steps of modeling to some diagnosis analysis. The last Chapter 8 summarize the conclusions of the thesis and discusses the overall benefits and problems of using OpenModeica for fault modeling as it is done in this study.

1.3 Contributions

The main contributions of this thesis are: Chapter 3, where a small fault modeling library is introduced and the problems and benefits with fault modeling in Modelica is analysed. Chapter 4, where transformation of the representation of data, including behavioural modes is performed so that the data is on a form that suits the analysis algorithms. Chapter 5 and Chapter 6, where it is shown how the diagnosis algorithms in MATLAB are used for sensor placement analysis respective test design analysis. In Chapter 5 is also presented a way to reduce the system to decrease the computing time of the sensor placement analysis.

Chapter 2

Diagnosis theory

In this chapter some theory of fault diagnosis is outlined and model based diagnosis to achieve fault detection and isolation is described. The word diagnosis is derived through Latin from Greek word for to discern or distinguish. In the technical domain, diagnosis is the process of detecting and locating the source of deviating behaviour of a system, faults. The aim of the diagnosis process is to detect and isolate faults. Detection of faults is simply to discover that a fault has occurred. Isolation of fault is to determine where the fault has occurred. See Figure 2.1 for an outline of how the diagnosis system functions in interaction with the physical system.

2.1 Model Based Diagnosis

In model based diagnosis, a model of the system that should be diagnosed is created. Then the observed behaviour of the system is compared to the behaviour of the model. If the observations from reality and those from the modeled behaviour do not agree, a fault has occurred. See Figure 2.2 for an outline of a comparison between a model of the system and the actual system.

In the comparison between the observations from the model and the observation from the actual process it is possible to detect faults. The different observations form a sort of redundancy, i.e. the information about the system is available from several sources, both the system itself and its model. To be able to also isolate the faults, i.e determine in which part of the system a fault originated, analytical redundancy is used to construct tests. Analytical redundancy is exists if it is possible to determine the value of a variable in several ways. If analytical redundancy exists in a system it is possible to compare some of the systems parts with each others to discover inconsistency's in the system, i.e. faults.

Consider as an example the system described by (2.1).

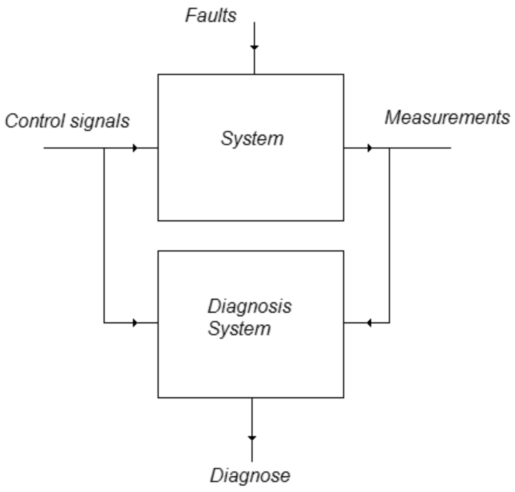

Figure 2.1: Outline of a system and corresponding diagnosis system with signals.

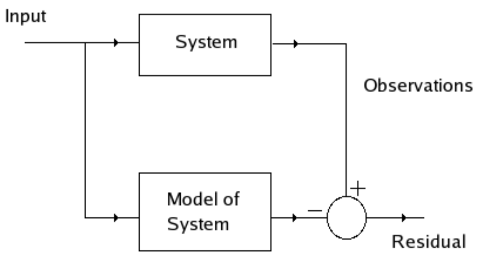

Figure 2.2: Comparisons between system and model.

Table 2.1: Overview of which variable that influence which residual.

	u	y_1	y_2
r_1	X	X	
r_2	X		X
r_3		X	X

$$x_1 = u \tag{2.1a}$$
$$x_2 = -x_1 \tag{2.1b}$$
$$y_1 = x_1 \tag{2.1c}$$
$$y_2 = x_2 \tag{2.1d}$$

The varible u is a input variable to the system and its value can be measured, y_1 and y_2 are output variables from the system and are also observable, the variables x_1 and x_2 are internal variables to describe the system. From equations (2.1), a number of tests can be constructed to test if the observations from the system are consistent with its model. The test are in the form of residuals that acquire a nonzero value if the respective variables do not behave as expected.

From (2.1a) and (2.1c) the residual (2.2) can be created.

$$r_1 = y_1 - u \tag{2.2}$$

From (2.1a), (2.1b) and (2.1d) can the residual (2.3) be derived.

$$r_2 = y_2 + u \tag{2.3}$$

The equations (2.1b), (2.1c) and (2.1d) can produce the residual (2.4)

$$r_3 = y_2 + y_1 \tag{2.4}$$

Now the three residuals (2.2)– (2.4) can be used to determine which variables that is the cause if some inconsistence should arise in the observed variables. If for instance the sensor that measure the input signal u should malfunction and give the wrong value this will affect residuals (2.2) and (2.3) but not residual (2.4). Faults in the sensors that measure variables y_1 and y_2 will affect different residuals in a similare way, see Table 2.1. Thus it is possible to determine which fault that has occurred from the information of which residuals that are nonzero.

If a system contains analytical redundancy, then it is equivalent to that there are analytical overdetermined parts in the system.

When a residual reaches some threshold, it will be considered to have reacted, see Figure 2.3 for a plot of a residual with a threshold level indicated by the crosshatched line. The threshold should be a value that is considered significantly large so that the diagnosis system do not report faults because of disturbances, i.e. false alarms.

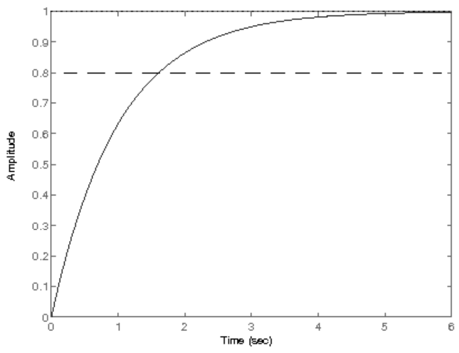

Figure 2.3: Plot over a residual registering a fault.

As seen, the information from tests can be used to determine what, if any, fault that affect the system,. The results from all tests are gathered together in a decision process to decide if a fault has occurred and which fault in that case, i.e. detection and isolation of faults, see Figure 2.4. To make sure that all interesting faults are possible to detect accordingly to the design specification, fault modeling is used.

A fault model is a model with support for faults, naturally only faults that are anticipated are possible to model. When the model with implemented faults is analyzed, there may be missing analytical redundancy in some part of the system, so that some interesting fault is not possible to isolate. In that case it is possible to form the foundation for the extra tests that are needed to obtain the requested isolatebility of faults. Extra redundancy is introduced by createing new overdetermined parts. These are created by introducing a measurable variable, a sensor, into the part of system where more redundancy is desired. Thus the fault model is used to verify that the system can be monitored by the diagnosis system and which test that are interesting to construct and use.

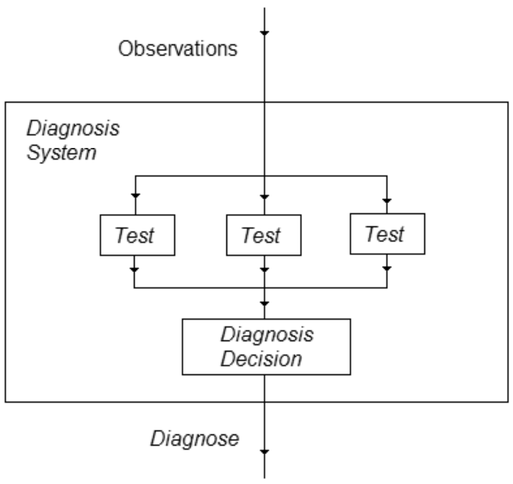

Figure 2.4: Diagnosis system.

Chapter 3

Fault Modeling for Diagnosis in Modelica

Modelica is an object-oriented programming language developed especially for modeling. It is easy to use and allows the designer much freedom in modeling different behaviour of the model. The description of a model is done by listing its equations, these are defined by the laws of nature that the object follows.

For example: A falling object is affected by the Earths gravitation. (The resistance from air is neglected). The Equations 3.1 describes the behaviour of the object.

$$\begin{aligned} \dot{h} &= v \\ \dot{v} &= -g \end{aligned} \tag{3.1}$$

Where v is the objects velocity, h the height of the object above some reference level and g the acceleration that the earth gravitational field affect the object with. The OpenModelica compiler merges the different equations, variables and parameters in the model to one system of equations. An example with two falling objects, Ball and Apple are modeled with the equations from 3.1. The result from the OpenModelica compiler would look like this:

```
parameter Real Apple.g = 9.82;
Real Apple.h;
Real Apple.v;
parameter Real Ball.g = 9.82;
Real Ball.h;
Real Ball.v;
equation
  der(Apple.h) = Apple.v;
  der(Apple.v) = -Apple.g;
  der(Ball.h) = Ball.v;
  der(Ball.v) = -Ball.g;
```

This model can be simulated with start values for the h variables.

3.1 Fault Modeling

To model faults can be somewhat challenging, since only faults that are known so to speak is possible to actively model, unforeseen faults are by definition hard to predict and model. But the predictable faults can be incorporated into the models of the respective component. When designing the models that should describe faulty behaviour as well as the normal behaviour, Modelicas standard component library were used as a platform to expand upon. Both to make the new models compatible with old existing designs and to minimize the effort in implementing fault models. The actual fault modeling is done in two ways, faults that affect the value of a quantity, like friction or voltage, is modeled as a variable that is inserted into the model. The fault variable is not instantiated with a value and it serves as an identifier of the fault. Some faults may not be suitable to model in this way. Faults that changes the behaviour of the model in some manner that a variable is unable to describe. These kinds of faults are modeled by using conditional equations. A conditional equation is an equation that only holds if its condition is fulfilled, see (3.2) for an example of a conditional equation. To identify the variables that represents faults, name convention is used. Some names are reserved for special variables and the specifications in Section 4.3 should be adhered to so that the models with fault support are compatible with the analysis functions developed in this study.

$$x = \begin{cases} y/2 & \text{if } a = 1 \\ y & \text{if } a = 2 \end{cases} \tag{3.2}$$

3.1.1 Fault Modeling with Inheritage

Modelica uses inheritage, this was used so that the design could be as modular as possible. That way many of the fault designs are uniform for components that inherit the same basic classes.

For example, in the Modelica standard library there exist a class called OnePort, it consists of just two pins and is the basis for all electrical components that has two connection pins. See Figure 3.1. Faults that may affect all electrical two pin components can be can be modeled in the OnePort class, which is inherited by all classes that has two pins, e.g resistor, inductor, etc.

Another advantage of inheritage is that models higher up in the hierarchy may need virtually no modifications to incorporate the fault design as long as no new faults may occur in the new model that inherited other classes. But say a electrical circuit, were some fault may cause a short circuit between components may need some additional modifications to support the short circuit-fault.

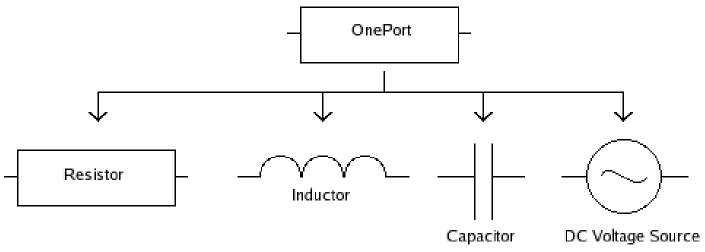

Figure 3.1: Inheritage of OnePort class

3.2 Behavioural Modes

The conditional equations discussed in Section 3.1 are used to model different behaviour under different circumstances. These different circumstances are called behaviour modes. One behaviour mode is always the fault free case, abbreviated as “NF”, no fault. Other modes, describing different faults have corresponding labels, e.g. “FaultyResistor” is the label of the mode that describes a resistor that have a fault in its resistance quantity. Under every behaviour mode the models behaviour is described with equations, these can be different or in part the same for different modes. To exemplify; a electrical component, a resistor for example, behaves correct when in mode NF, behaves as a wire if in mode ShortCircuit and as isolation if in OpenCircuit mode, see (3.5)–(3.7).

The aim is that all models should work as normal if every behaviour mode set to NF, fault free. The model should then be possible to simulate. There are some obstacles to get the simulation to work however, see Section 3.5 for more information about this. To implement the behavioural modes as conditional equations, two ways have been investigated, if-equations and if-expressions.

3.2.1 If-Equations

The most intuitive way to model behavioural modes would be if-equations, which is equations that holds if some condition is true. An if-equation is an if-expression with equations in the conclusion part, i.e. after the word then. For example, if A then (equation a), else then (equation b)

OpenModelica supports if-equations, but the if-equations get transformed to if-expressions inside equations by the OpenModelica compiler. So the conditional equations is in the end in if-expression format.

3.2.2 If-Expressions

An if-expression is an expression containing an if-clause. For example: $a = b \cdot$ (if B then b else then c).

The usage of if-expression to describe a behaviour may result in quite large equations, because to realize a if-equation may demand several if-expressions. Here is OpenModelica code from the model of an ideal linear electrical capacitor that have fault support.

```
equation
      0 = if (bm == "NF") then i - C*der(v)
      elseif (bm == "FaultyCapacitor") then i - (C+C_fault)*der(v)
      else 0;
```

The expression after the word `then` on the first line holds if the condition variable bm, abbreviation for behavioural mode, is equal to NF. The resulting equation will then be (3.3), i.e. the equation describing the current from a capacitor. If the condition variable is equal to FaultyCapacitor, then the second line will result in (3.4), which is the same equation as (3.3) with an additional variable C_{fault} that models a fault in the capacitors physical quantity, the capacitance.

$$i = C\dot{v} \tag{3.3}$$

$$i = (C + C_{\text{fault}}) \cdot \dot{v}. \tag{3.4}$$

3.3 Examples of Models with support for Fault Modeling

As mentioned in Section 3.1, the existing Modelica library were used as base for the fault supporting components that were designed. The new components with fault support were placed in a new library called ModelicaFault. Basic electrical and mechanical faults are implemented into the basic models in this library. For example, the base class for the electrical two pin component which is mentioned in Section 3.1.1, OnePort, is taken. In its fault free implementation all two pin components have some basic equations that describe their behaviour, see (3.5).

$$v = p.v - n.v \tag{3.5a}$$

$$0 = p.i + n.i \tag{3.5b}$$

$$i = p.i \tag{3.5c}$$

The equations (3.5), where p and n stands for the positive respective negative pin, describe how the current i flows through the component and how the voltage v is the difference between the two pins. Two possible faults that may affect any two pin component where devised; the component may be short circuited or disrupted. In the case of a short circuit, (3.5) is still true, but (3.6) is added to model that the two pin component behaves as a wire.

$$v = 0 \tag{3.6}$$

If the component is disrupted and thus behaves as a isolation (3.5a) is replaced by (3.7), which along with (3.5b) and (3.5c) describes how no current flows through the component but leaves the voltage at the two pins free to acquire any value.

$$i = 0 \tag{3.7}$$

From this base class with support for short circuit and disrupted behaviour, named FaultyOnePort, the more advanced two pin components are built, the capacitor is described in Section 3.2.2. The component that models a voltage source will also be brought up as an example, the rest of the two pin components with support for faults are designed in a similar way. Here below is the Modelica code of the component that model a voltage source that delivers a constant voltage:

```
model FaultyConstantVoltage
   //"Source for constant voltage with support for faults"
   parameter SI.Voltage V=1 "Value of constant voltage";
   SI.Voltage V_fault;
   String bm;
   extends Interfaces.FaultyOnePort;
equation
   0 = if (bm == "NF") then v - V
   elseif (bm == "FaultyConstantVoltage") then v - V + V_fault
   else 0;
end FaultyConstantVoltage;
```

In the code can be seen both how the model extends the FaultyOnePort and how an additional behavioural mode and the equation that holds in that are added. The OpenModelica compiler will thus produce this system of equations from a FaultyConstantVoltage component, named U:

```
  0.0 = if U.bm == "NF" then U.v - U.V else if U.bm ==
"FaultyConstantVoltage" then U.v + -U.V + U.V_fault else 0.0;
  0.0 = if U.bm == "NF" then U.i - U.p.i else if U.bm == "ShortCircuit"
then U.i - U.p.i else if U.bm == "OpenCircuit" then U.i - U.p.i else 0.0;
  0.0 = if U.bm == "NF" then U.v + -U.p.v + U.n.v else if U.bm ==
"ShortCircuit" then U.v + -U.p.v + U.n.v else if U.bm == "OpenCircuit"
then U.i else 0.0;
  0.0 = if U.bm == "NF" then U.p.i + U.n.i else if U.bm == "ShortCircuit"
then U.p.i + U.n.i else if U.bm == "OpenCircuit" then U.p.i + U.n.i
else 0.0;
  0.0 = if U.bm == "ShortCircuit" then U.v else 0.0;
```

It can be seen that all the behaviour of the FaultyOnePort class is inherited and an additional variable that represents a fault in the voltage quantity is introduced on the second line.

Another way to introduce faults in a model is to add basic components whose sole purpose is to model faults, in the example above with the voltage source this approach would result in a extra voltage source connected in series with the first.

The new voltage should then have the parameter that represents its voltage amplitude exchanged to a variable that models the fault. This approach is sometime required, e.g. when a potential short circuit between components in a circuit is to be modeled, then an extra switch can be introduced between the concerned components. The switch control variable is then connected to a fault variable.

3.4 Extracting Flattened Equations

To be able to use the information the OpenModelica compiler produces from a model the functionality to dump equation systems to file were implemented in the OpenModelica compiler. The compiler file DAEQuery.mo in OpenModelicas file structure contains all features that this study contributed to the OpenModelica compiler.

The system is described by differential algebraic equations in OpenModelica. Among those are the if-expression discussed earlier in Section 3.2.2. Since only structural representation is requested, only the information that a variable is present in a equation is needed, not *how* it occurs in the equation. See Chapter 4 for information about structural representation.

The information from OpenModelica is accesed through the command:

```
exportDAEtoMatlab("Modelname")
```

to the OpenModelica compiler, which writes the data on a format that MATLAB can handle to a file named “modelname”_imatrix.m. See Section 4.2 for information about the format in this file.

3.5 Simulation

It was stated in Section 3.2 that the aim is to be able so simulate models that has behavioural modes. There are however some problems regarding the simulation of these models in OpenModelica. To begin with, string equality is not yet implemented into the OpenModelica compiler and the conditional variables, abbreviated bm, are strings. To be able to simulate the models with behavioural modes, some modifications for the models are in order until string equality works in OpenModelica. A workaround is to change the conditional variable to some other type than a string, for example an integer or a boolean. Another problem that arises when trying to simulate a model with behavioural modes is the empty equations that may be introduced via the behavioural mode design. For example in Section 3.3, the equation (3.6) only holds when the behavioural of the component is in the short circuit-mode. Thus, to implement this equation an else-part is needed in the if-expression, this else part is simply the equation: $0 = 0$, which is just a dummy equation. However, when OpenModelica tries to simulate the model this dummy equation is counted when the compiler determines if the model is under- or overdetermined and results in an overdetermined system according to the compiler. One quick fix to get around this problem is to introduce corresponding dummy variables for each dummy equation that makes a difference in the count towards a well posed problem with equal amount of variables and equations. A better solution would be if the compiler could identify and disregard dummy equations.

When analysis algorithms that do not have support for behavioural modes are to be applied to the model some care has to be taken when introducing behavioural modes that add extra equations to the model. See Section 4.2.1 for more detailed information regarding this.

The fault variables also affect the variable/equation count and makes the model underdetermined, but this is simply an effect of the fault modeling design and it is in the nature of the problem that a model with a fault of unknown quantity is not possible to simulate. Thus the fault variables must be removed or instantiated with values if simulation should be possible.

3.6 Conclusions

The fault modeling design described in this chapter is by no way a definitive way to implement fault modeling. But it may function as a groundwork to expand or develop from. The analysis that is done on the material that the OpenModelica compiler produces can handle a variation of designs and the library structure presented in Section 3.3 is just a suggestion, if only a few specific faults are interesting then it is perhaps easier to implement these higher up in the hierarchy as extensions to a fault free model. Some issues arise when trying to simulate models with fault support as discussed in Section 3.5, solutions to these has not been thoroughly researched as the ability to simulate the models are not the chief aim of this study.

Chapter 4

Representation of Data

As seen in Chapter 3, the models are built and compiled with OpenModelica, then, as will be explained in Chapters 5–7, some diagnosis analysis is done on the data produced from OpenModelica. The analysis uses structural methods and is done in MATLAB. As will be described in Chapter 6, the analytical information from the OpenModelica model is also of interest. Thus the system of equations produced by OpenModelica should be converted to a format that MATLAB can handle, both on structural and analytical form. In this chapter the structural representation in MATLAB is solved, but the representation of the analytical information encounter obstacles.

In this chapter the representation of the data is explained. First in Section 4.1, is an overview of structural representation, then in Section 4.2 it is explained how the model data is represented during the different phases from modeling to analysis. In Section 4.3 naming conventions are discussed and in Section 4.4 is a discussion about derivatives.

4.1 Structural Representation

In a structural representation of a model only the presence of a variable in an equation is presented. The analytical and numerical values in equations are ignored. Thus a structural equation is just a list of the variables that the equation contain. As an example consider the system of equations (4.1), which is on analytical form. A structural representation of the system of equations (4.1) is the matrix (4.2), where the variable x is represented by column one, variable y by column two and variable z by column three. Equation (4.1a) is representated by row one in the matrix (4.2), (4.1b) by row two and (4.1c) by row number three.

$$2x = \frac{y}{z} \tag{4.1a}$$

$$\dot{x} = -3z \tag{4.1b}$$

$$z^3 = 2 \tag{4.1c}$$

$$\begin{bmatrix} X & X & X \\ X & & X \\ & & X \end{bmatrix} \tag{4.2}$$

All information about operations and relations between variables disappear when the system is converted from analytic form to structural form. However, as will be explained in Section 6.5, for detecting strong and weak faults, information regarding the derivatives fom each variable must be acquired. See Section 4.4 of how the derivatives are represented in the data.

4.1.1 Structural Matrix

A structural matrix is an matrix where there exists no values and signs of the elements, just a placeholder that signifies that an element is present. The structural matrix is in a sense digital, it only contain binary information about the elements. In this study ones and zeros are used to indicate if a place in the matrix is filled or not.

Structural matrixes are used here to represent systems of structural equations. Each row represents an equation and each column represents a variable. Hence a one at place (i, j) in a structural matrix denotes that the equation i contain the variable that stands for column j.

4.2 Representation of System

The models originate from Modelica, in Modelica the variable names are on dot notation form, i.e. a dot separates the levels of hierarchy from each other, as seen in the code from OpenModelica in the begining of Chapter 3, where both the object apple and ball has variables for height and velocity h and v. When the model has been compiled by the OpenModelica compiler it is representated by a system of differentiated algebraic equations. This system is exported to a file named "modelname_imatrix.m" via the command `exportDAEtoMatlab` to the OpenModelica compiler. In this file the system of equations from the model is represented as an array containing lists, one list for each equation. Listed in each equation list is the variable indexes of the variables that the equation contain. The equations containing conditional expressions as the one in Section 3.2.2 are represented with some additional information, see Section 4.2.1. In the file is also a list of all the variables in the model, they are simply recorded as a list of all the variable names. The first variable in the list has variable index one, the second variable name has variable index two and so on. Last in the file is the whole model including all declarations of parameters, variables and all equations i analytical form. This information is used when recreating analytical equations after the test design analysis, see Section 6.2.

The information in the imatrix.m file is in a format that MATLAB can handle, it is read by the function imatrix2sm. This MATLAB-function converts the information in the file to two matrices and two lists. The first matrix is a structural

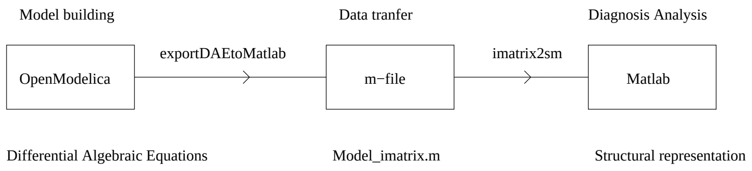

Figure 4.1: Overview of the different data representations

matrix, see Section 4.1.1 for information about structural matrixes. It is denoted by "m" and contains a row for each equation and a column for each variable in the system. The second matrix is denoted by "bm", and is referred to as the behavioural mode-matrix. In the behavioural mode matrix additional information regarding the equations are given, namely if they are conditional equations and if so their properties. For condition equations, the condition, i.e. the condition variable or variables, the relation and the value to compare against is listed. The behavioural matrix also contain information about which equation in the "_imatrix.m" file that the structural equation originated from. All equations in the structural matrix have information about their original equation in the "_imatrix.m" file. See Section 4.2.1 for more information about the representation of behavioural modes.

The third item of the system representation is a list, of all the variables that the model contains. It is referred to as variable list and is the same as the variable list in the m-file. It is denoted by "v".

The final part of the system in Matlab is the list of the analytical equations, this list contain the same equations as the list of the analytical system in the m-file does, but the declaration of parameters and variables are removed.

So to conclude, the system is represented by a structure with four sub-parts in Matlab: a structural matrix, a behavioural mode matrix, a variable list and a list of analytical equations. In Matlab notation with the data set named sm:

```
sm.m, sm.bm, sm.v sm.EqStr
```

See Figure 4.1 for an overview of the representation of model data in this study.

The whole process of how the data is transfered from OpenModelica to the Matlab format is exemplified in Section 4.2.2, were the format of the conditional equations in the file and the behavioural mode-matrix is also shown.

4.2.1 Representation of Behavioural Modes

Behavioural modes and conditional equations, such as (3.2), are represented in OpenModelica as equations with if-expressions as seen in Section 3.2. When exporting these equations to the m-file, they end up in the array with equations as lists as described in Section 4.2. The lists representing equations with if-expressions contain more information than just the indexes of the variables that are contained in the equation. For each if-statement there is a new list with information about the conditions concerning this if-statement. First in this list is the word "if" identifying the list as representing an if-expression. The second item is the value that the condition variable should be compared against to determine if the condition is true or not, next comes the relation that are used for the comparison. As item number four is the index of the condition variable. Items number five and six are the conclusions of the if-expressions, the information at the fifth place represents the result if the condition is true, the sixth represents the result if the condition is validated as false. Both item five and six can be new conditional equations represented by new lists or just some variable indexes representing variables that are part of the equation in respective case. Thus several nested if-expressions can be handled.

When the equations are represented as matrixes in the MATLAB format the respective equations from OpenModelica containing if-expressions are transformed to if-equations in structural form. Thus these equations are split into one structural equation stored in the structural matrix for each conditional case. The information about which equations that hold under which circumstances is stored in the behavioural mode-matrix.

The behavioural mode matrix has four rows and the same number of columns as there are equations in the system, i.e. the number of rows in the structural matrix. The equations are ordered in the same way as the equations in the structural-matrix: the first column in the behavioural mode-matrix is connected to the first row in the structural-matrix, the second column to the second row and so on. The behavioural mode-matrix is a table for all equations in the structural matrix. All equations, including the ones that are not conditional equations are represented in the behavioural mode matrix with a column.

The first row in the bm-matrix identifies which Modelica-equation the present equation originated from, the second row shows if the equation is an if-equation. If so the element on the second row is non-zero, it is the index of the variable/variables in the condition in the if-equation. This information is structural and shows just which variables that are part of the condition. The third row contain the relation in the condition, e.g. $==$ or $<$. The fourth and final row is the value that the variable in the if-condition should be compared to via the relation, e.g. true, 5 or $P_{\max}$.

If the condition equation structure is deep, i.e. several if-expressions are nested inside each other, the information in the behavioural mode matrix describes just the last condition for each resulting equation. The total condition when respective equation is valid can however be traced back via the information regarding which equation in the m-file the specific structural equation was derived from. From the number of the original equation in the m-file the analytical equation stored in the list described in Section 4.2 can be obtained. From the analytical expression it can be concluded under which circumstances the structural equation holds. This process is not automated as of now and must be done manually, in part because there are some problems with the ordering of equations in the analytical list of equations see 6.2 and in part because no more advanced handling of logic expressions, which is needed, for MATLAB have been constructed or imported in this study.

Merging of If-Equations

In some cases all conditional equations are unwanted or unnecessary, then there exists the possibility to merge all equations back to the original equations they were derived from. Then all the variables in the different branches of this equation, minus the conditional variables, will be present in the new merged equation. The MATLAB-function that merges conditional equations branches is called mergeif.m.

The `mergeif`-function looks up all equations in the structural representation that originated from the same equation in the "_imatrix." file. All the variables in all the equations that "belong" to the same original equations are put together in a structural equation that replaces the equations that originated from the same analytical equation.

If the mergeif-function is to be used, care should be taken so that the model after the merging of condition equations still describes the requested behaviour of the model. Behavioural modes are used in the fault model building to add equations that only holds under special circumstances, if these equations are merged into the general behaviour of the model with no consideration taken to under which condition the equations hold, the result may be incorrect. Consider the OnePort component in Section 3.3 as example.

When the OnePort component is in behavioural mode short circuit, (3.6) is added to the system of equations (3.5). But if the information about the behavioural modes is lost as it is when the mergeif-function is applied, the system will automatically assume the mode short cicuit if (3.6) is present and the other faults of the OnePort component are impossible to detect. A solution to this problem is to remove (3.6) from the model, if analysis that do not support behavioural modes are used, such as the sensor placement analysis in Chapter 5.

4.2.2 An Example for the Representation of a Model

An equation with an if-expression in OpenModelica code:

```
x = if a then y else z;
```

The model is dumped to a file with the ending "_imatrix.m" through the command exportDAEtoMatlab("modelname") to the OpenModelica compiler. In this file the equation with the if-expression will become a list:

```
[1,{'if', 'true',' == ' {4},{2},3}]
```

Where the numbers are the indexes for the variables, i.e. the variabel list looks like this:

```
sm.v = {'x','y','z','a'}
```

Thus the 1 representing the x variable starts the equation, then comes the if-expression inside brackets, {}, the if-expression is identified by its first item, the word "if". Then comes the value that the condition variable is compared against, then the relation and after that the index of the condition variable, 4 in this case, representing the variable a. As item number five in the if-expression list comes the conclusion if the condition relation is true, which is 2 here, representing variable y. Last comes the conclusion if the condition relation is false, 3, representing variable z. The list can thus be read like this: An equation containing the variable with index 1 and if the variable with index 4 is equal to, ("=="), the value "true", then the equation also contain the variable with index 2. If variable with index 4 is not equal to the value "true", then the variable with index 3 is part of the equation.

In the next step, each branch of the if-expression is disconnected from the rest of the equation and thus becomes a stand alone equation that is valid if the condition is fulfilled. Hence in the example above two structural equations would result from the equation with the if-expression. The first equation would contain variables with indexes 1 and 2, the second equation would contain variables with indexes 1 and 3. In matrix (4.3) the two equations are shown in structural matrix form, the format of the matrix is that explained in Section 4.1.1. The last column shows which equations variable with index 4 is part of, this column will be empty because the condition variable in this case is not part of any equation, it just decides which equation that holds.

$$\text{sm.m} = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix} \tag{4.3}$$

In matrix (4.4) the behavioural mode-matrix is shown. The first column holds information regarding the first equation and the second column for the second equation of the structural matrix (4.3). The first row shows that both the equations in (4.3) originated from the first equation in the file exported from OpenModelica. The second row indicate that both the equations in (4.3) are conditional equations, both with variable with index 4 as conditional variable. The third row shows the relation that should be fulfilled for the respective equation in (4.3) to be valid. In the fourth and last row the value that the condition variable in the second row should be compared against via the relation in the third row is shown. Thus the first equation holds if the variable with index 4 is equal to "true". The second equation in (4.3) holds if the variable with index 4 is not equal to "true". The tilde, $\sim$, character represents logical negation in MATLAB.

$$\text{sm.bm} = \begin{array}{cc} 1 & 1 \\ 4 & 4 \\ '==' & '\sim\ ==' \\ \text{'true'} & \text{'true'} \end{array} \tag{4.4}$$

4.3 Name Convention of Variables

In Section 3.1 it was mentioned that the fault variables are identified through a naming convention. The solution to use name convention is perhaps not an optimal one. A special type that designates interesting variables would maybe be better. However the problems in implementing this solution so that the type of variable gets communicated in a practical way together with the rest of the data have not been solved in this study. This would probably involve adding or modifying the representation in the OpenModelica compiler and is left for future work.

The name conventions are the following: Variable names ending with "_fault" are fault variables that denote a fault. Variable names with "_sensor" as ending are reserved for variables that signify possible sensor locations. Input variables that are considered known and should be handled that way in the analysis end their name with "_known". As seen in Section 4.4 the prefix "der_" to variable names is used to identify variables derivatives and should not be used in the model building. The variable name "bm" is used in the models with behavioural modes support, described in Section 3.2, as the name of condition variables. There is no real problem in using "bm" in other circumstances other than it may cause some confusion.

The naming convention is used to facilitate and automatize the diagnosis analysis. It is possible to disregard the naming conventions when building models with fault support, but one result is that more attention have to be spent in the analysis phase to correctly identify the different special variables.

4.4 Representation of derivative

Information about derivatives is needed to analyze weak/strong detectability, see Section 6.5 for more information about weak/strong detectability. If a variable is differentiated, this is denoted by a minus sign (-) in the file that the OpenModelica compiler produces. The denotion with the minus sign is somewhat obscure: If the variable is differentiated and occur undifferentiated somewhere else in the system then all instances where it is *not* differentiated, it occurs with a minus.

As a result of the way OpenModelica represents the differentiated variables, the whole system must be searched to determine if a variable somewhere occurs with a minus sign. If the variable is differentiated somewhere where the minus sign is not present. If a variables derivative is found, a new variable is introduced to the system. This new variable is called

```
der_'variablename'
```

The part after "der_", 'variablename' is the original name of the variable whose derivative has been found. The structural matrix and the variable list is expanded with the new variable.

Functionality exists to merge the variables and their derivative to a single variable in the structural representation. This merging must be done when searching for overdetermined parts in the system. When merging the differentiated variables all information about derivatives is lost, but this information can be accessed by going back to the unmerged structural model. The Matlab function that merges variables is called mergeder.m.

Chapter 5

Sensor Placement Analysis

In this Chapter it is shown how sensor placement analysis can be performed on fault models designed along the lines given in Chapter 3. A reduction algorithm that reduces the structural models is presented in Section 5.3 and in Section 5.4 is sensor placement exemplified on a small circuit.

Sensor placement analysis for diagnosis is the analysis of possible placements of sensors that makes detection and isolation of faults possible. Given a design-specification the analysis results in the different possible sets (if any) of sensors that identifies the specified faults. A fault can be just detectable, i.e. it is possible to determine that a fault has occurred, but not which fault. If it is possible to separate the faults from each other, specifically to conclude which fault that has occurred, the fault is isolable from the rest of the faults.

As a basis for the sensor placement analysis lies redundancy. If it is possible to compute a variable in several ways and these do not all agree, then something has changed the behaviour of the process, thus a fault has occurred.

5.1 SensPlaceTool

SensPlaceTool is a tool developed by Erik Frisk and Mattias Krysander at Vehicular Systems at LiTH. SensPlaceTool is implemented in Matlab. Given a model on structural form (see Section 4.1.1 for an explanation of structural matrices), a set of faults and a set of possible sensors, SensPlaceTool will determine which faults that can be detected and isolated and which set of sensors that will achieve the design specifications.

The model format for input is as follows: a structural matrix, a list of variable names, a list of which variables that are faults and should be monitored and a list of which variables that are possible sensors. Furthermore it is possible to state if new sensors can become faulty.

SensPlaceTool can not handle behavioural modes. Thus the model will have to be modified by merging all if-equations that are derived from a equation containing if-expressions back to a single equation. These merged equations are in structural form and the information about different modes is lost, see Section 4.2.1.

5.2 Identification of Sensor, Fault and Input-Signals

From the system it is necessary to determine which variables that are sensors, faults and inputs. The SensPlaceTool needs to have these specified to work correctly. These signals can either be manually specified or they can be automatically sorted out from the system if the model is designed according to the specifications, e.g. the naming convention, discussed in Section 4.3, is used. If the design specification is automatically constructed, all accurate named possible faults, sensors and input-signals will be included. If, e.g. not all faults that the model supports should be included in the design specification, this needs to be manually adjusted.

5.3 Reductions of the Structural Model

SensPlaceTool is sensitive to the amount of equations and variables, i.e. the size of the structural matrix, regarding its executing time. As seen earlier, in Chapter 3, OpenModelica produces an abundant quantity of equations for even small models. In order to reduce the computing time, the structural matrix is reduced to a smaller size. The resulting system is equivalent to the original regarding the matter of over- and underdetermined parts and detectability and isolability of faults.

It is important that the analysis result is invariant, meaning that they do not change the behaviour of the model with regards to the analysis result. To ensure that no vital information is lost during the simplification, all specified fault- and sensor-signals are identified before, see Section 5.2. The reduction function for the structural matrix will preserve these signals when reducing the system.

The actual reduction algorithm is divided into three parts:

The first reduction step eliminates variables that can be replaced by another variable in the system. Structural equations in the system containing exactly two variables are identified, i.e. rows with two elements (ones) in the structural matrix. This equation is called eq_{elim}. The variables in eq_{elim} are checked to see if they are reserved, i.e. if they are sensor or fault signals. If at least one of the variables is not reserved, it is booked as var_{elim} and is the variable that will be removed. Then the other variable of eq_{elim} replaces var_{elim} everywhere var_{elim} exists. Lastly eq_{elim} and var_{elim} is removed from the system, which thus contain one equation and one variable less than before.

In (5.1) a small structural matrix representing a system is reduced by eliminating the variable with index two. The third row is identified as an equation containing exactly two variables, variable with index two is not a fault or sensor variable, then the other variable in row three replaces variable two in rows one and two. Equation two does already contain variable two and is thus unchanged. Lastly equation, (row), three and variable, (column), two are removed. In (5.1) eq_{elim} is the equation with index three and variable var_{elim} has index two. The system of equation (5.2) could be the system of analytical equations corresponding to the structural matrix (5.1). The corresponding eliminations are done in (5.2) to illuminate the process.

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 1 \end{bmatrix} \rightarrow \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix} \rightarrow \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \tag{5.1}$$

$$y = 1 \qquad z = 1 \qquad z = 1 \tag{5.2a}$$
$$x + y = 2z \quad \rightarrow \quad x + z = 2z \quad \rightarrow \quad x = z \tag{5.2b}$$
$$y = z \qquad z = z \tag{5.2c}$$

The second reduction step is similar to the first, with the difference that columns, instead of rows, with two elements are identified. Variables that are not reserved and is part of exactly two equations are identified, such a variable is called var_{elim}. One of the equations that contain var_{elim} is called eq_{elim}. The variables in eq_{elim} except var_{elim} is called var_{res} and are added to the other equation that contain var_{elim}. The variable var_{elim} and equation eq_{elim} are then removed from the structural matrix which is reduced by one equation and one variable.

In (5.3) a small structural matrix representing a system is reduced by the method described above. The third column is identified as representing a variable that is part of exactly two equations and this variable is not a fault or sensor variable, variable with index three is thus var_{elim} in this example. Equation two contain var_{elim} and is chosen as eq_{elim}, and thus the rest of the variables in equation three other than var_{elim} is var_{res}, just variable with index two in this case. Then var_{res} is added to var_{elim}. Finally var_{elim} and eq_{elim} are removed.

$$\begin{bmatrix} 1 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 0 & 1 & 1 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \tag{5.3}$$

In (5.4) the same procedure is done as in (5.3) but on a system of analytical equations. It can be seen that the second equation in the second step in (5.4) do not corresponds exactly to the second row in the second matrix in (5.3), this is because the elimination of the variable z is done in two steps in the structural representation. First the substitute, $2y$ in analytical form this case, is added, then the variable that is to be eliminated is removed. But the end results are the same.

$$x = y \qquad x = y \qquad x = y \tag{5.4a}$$
$$z = 2 \quad \rightarrow \quad 2y = 2 \quad \rightarrow \quad 2y = 2 \tag{5.4b}$$
$$2y = z \qquad 2y = z \tag{5.4c}$$

The third reduction step identifies equations that only contain one variable. If this variable is not reserved it is removed and the equation containing just the one variable is also removed. An equation with just one variable can be viewed as a assignment of value to this variable. The variable is hence no longer unknown and since representation of numerical values is not part of the structural representation, see Section 4.1, the variable can be removed from the system in all equations where it exists.

In (5.5) a small structural matrix is reduced by identifying variable with index three as being present in an equation with just one variable, equation two. Equation two and variable three are removed.

$$\begin{bmatrix} 1 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \tag{5.5}$$

In (5.6) can be noted the corresponding process as in (5.5) but on a system of analytical equations. In the analytical case another step is added to transfer the nummerical value that the variable z is assigned to the variabel y, this step is ignored in the structural case since nummerical values are ignored.

$$\begin{aligned} x &= y & &\quad & x &= y & &\quad & x &= y & &(5.6a) \\ z &= 2 & &\rightarrow & z &= 2 & &\rightarrow & 2y &= 2 & &(5.6b) \\ 2y &= z & & & 2y &= 2 & & & & & &(5.6c) \end{aligned}$$

Because OpenModelicas produces a significant amount of simple equation to describe the system, the reduction methods presented in this Section reduces the system by a substantial degree in the typical case of a model system constructed via OpenModelica. The reduction function is accessed by the Matlab function reducesm.m

Conditional equations with corresponding information in the behavioural matrix, see Section 4.2.1, are reserved from elimination in the reduction function since the aim of the process is to make a model more manageable by the sensor placement analysis and SensPlaceTool lacks support for behavioural modes. See Section 4.2.1 about eliminating behavioural modes.

5.4 Circuit Example

In this section sensor placement analysis will be performed on a small circuit. For a more detailed and indepth example on how to design a model with support for faults and how to perform test design analysis as well as sensor placement analysis on it, see Chapter 7.

The circuit in this section consist of three resistors, a capacitance, an ideal operational amplifier and a voltage source, connected as shown in Figure 5.1. In figure 5.1 some electrical potentials and currents are also indicated.

The equations that describes the model are referred to as Equations 5.7. The symbols v_i and i_j represents electrical potentials and currents. The upper-case letters R_n and C are the physical quantities of the components resistance and capacitance and u is the electrical voltage output of the voltage source, in this

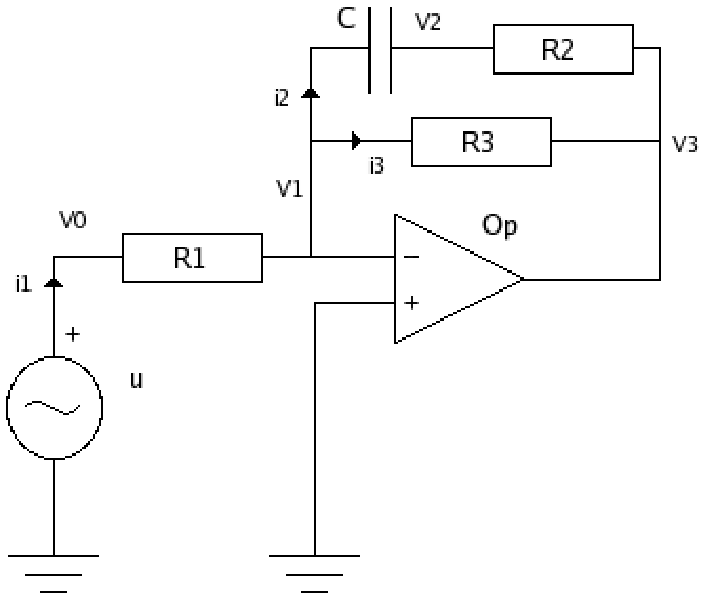

Figure 5.1: Circuit schematic

example u is considered to be known. When instantiating the model of the circuit in Figure 5.1, OpenModelica produces a equivalent system but with far more equations. The Modelica produced equations contains support for diffrent behavioural modes, which is not used in this example, see Apendix A for all equations produced by the OpenModelica compiler from the model.

$$v_0 - v_1 = R_1 \cdot i_1 \tag{5.7a}$$
$$v_2 - v_3 = R_2 \cdot i_2 \tag{5.7b}$$
$$v_1 - v_3 = R_3 \cdot i_3 \tag{5.7c}$$
$$C\frac{d}{dt}(v_1 - v_2) = i_2 \tag{5.7d}$$
$$v_1 = 0 \tag{5.7e}$$
$$v_0 = u \tag{5.7f}$$
$$i_1 = i_2 + i_3 \tag{5.7g}$$

The model consists of 7 equations and 7 unknown variables and are a well-posed problem with a unique solution.

When building the model in OpenModelica, new components with built in possible faults, as discussed in Section 3.1, are used so that each component incorporates a fault. In equation form this can be considered as that (5.7a)– (5.7f), that represents the different components influence on the behaviour of the circuit, are each corrupted by a fault and does not hold.

As explained in Section 3.1 the corruption of the equations are achieved by adding an extra unknown variable. This variable represent the fault that the sensors should detect. The equations describing the new model with faults are thus Equations 5.8.

$$v_0 - v_1 = (R_1 + R_{1\text{fault}})i_1 \tag{5.8a}$$
$$v_2 - v_3 = (R_2 + R_{2\text{fault}})i_2 \tag{5.8b}$$
$$v_1 - v_3 = (R_3 + R_{3\text{fault}})i_3 \tag{5.8c}$$
$$(C + C_{\text{fault}})\frac{d}{dt}(v_1 - v_2) = i_2 \tag{5.8d}$$
$$v_1 = Op_{\text{fault}} \tag{5.8e}$$
$$v_0 = u + u_{\text{fault}} \tag{5.8f}$$
$$i_1 = i_2 + i_3 \tag{5.8g}$$

The equation system is now underdetermined with 7 equations and 13 unknown variables. The system of the circuit represents no longer a problem with an unique solution, as can be expected when faults with unknown quantities are added. Possible sensor locations are added to the circuit, by adding sensor-variables. All possible sensor locations should be added in this phase, since the sensor placement analysis will later result in possible sensor-sets that achieve the desired isolability and detectability specification. When implementing both faults and sensors in Modelica the name convention discussed in Section 4.3 must be followed. These new sensor-variables are connected to corresponding physical quantities by equations, see Equations 5.9.

$$v_0 = v_{0\text{sensor}} \tag{5.9a}$$
$$v_1 = v_{1\text{sensor}} \tag{5.9b}$$
$$v_2 = v_{2\text{sensor}} \tag{5.9c}$$
$$v_3 = v_{3\text{sensor}} \tag{5.9d}$$
$$i_1 = i_{1\text{sensor}} \tag{5.9e}$$
$$i_2 = i_{2\text{sensor}} \tag{5.9f}$$
$$i_3 = i_{3\text{sensor}} \tag{5.9g}$$

The modified system with fault-signals and possible sensor locations consisting of Equations 5.8 and 5.9 is made of 14 equations and 13 unknown variables and is thus overdetermined. But since the sensor Equations 5.9 represent *possible* sensor locations the sensor placement analysis may not result in a overdetermined system in the end. Here is the Modelica code of the model, called Simpcirc:

```
model Simpcirc
```

```
  FaultyCapacitor C;
  FaultyResistor R1;
  FaultyResistor R2;
  FaultyResistor R3;
  Modelica.Electrical.Analog.Basic.Ground G2;
  Modelica.Electrical.Analog.Basic.Ground G1;
  FaultyIdealOpAmp3Pin Op;
  FaultyPulseVoltage u(period=period_known,V=v_known);

  Real    period_known,v_known;
  Real    v2_sensor,v3_sensor,v0_sensor,v1_sensor;
  Real    i1_sensor,i2_sensor,i3_sensor;
equation
  connect(u.p,R1.p);
  connect(R1.n,R3.p);
  connect(R1.n,C.p);
  connect(R1.n,Op.in_n);
  connect(C.n,R2.p);
  connect(R2.n,R3.n);
  connect(R3.n,Op.out);
  connect(u.n,G1.p);
  connect(G2.p, Op.in_p);

  v0_sensor  = u.p.v;
  v1_sensor  = R1.n.v;
  i2_sensor  = C.p.i;
  i3_sensor  = R3.p.i;
  v2_sensor  = C.n.v;
  v3_sensor  = Op.out.v;
  i1_sensor  = u.p.i;
end Simpcirc;
```

As can be seen, the sensor variables and the variables that are considered known, (the two variables defining the voltage source), follows the naming standard disscused in Section 4.3. SensPlaceTool is aplied to this system, with the design specification that all faults should be detectable and isolable. The result: To achieve detectability for all 6 faults, one of these sensors must exist:

$$v_{2\text{sensor}},\ v_{3\text{sensor}},\ i_{2\text{sensor}} \text{ or } i_{3\text{sensor}}.$$

There exists two sets of sensors that makes it possible to isolate all faults from each other:

$$\{v_{0\text{sensor}}, v_{2\text{sensor}}, i_{3\text{sensor}}, v_{3\text{sensor}}\} \text{ and } \{v_{0\text{sensor}}, v_{2\text{sensor}}, i_{1\text{sensor}}, v_{3\text{sensor}}\}.$$

If the sensors of one of these sets are added to the circuit and these new sensors contain possible faults. The result is still the same, i.e. no new sensors needs to be added for the detection and isolation of any new faults that may arise from the added sensors.

If not full isolation is required, perhaps only $R_{2\text{fault}}$ and u_{fault} needs to be isolated and the only possible sensors are: $i_{3\text{sensor}}, i_{1\text{sensor}}$ and $v_{2\text{sensor}}$ and no faults may occur in the sensors. With this specification, then one of these sensor sets is enough to isolate the two faults:

$$\{i_{1\text{sensor}}, i_{3\text{sensor}}\},\ \{i_{3\text{sensor}}, v_{2\text{sensor}}\} \text{ or } \{i_{1\text{sensor}}, v_{2\text{sensor}}\}.$$

If the sensors can introduce new faults then the resulting sensor set is:

$$\{i_{3\text{sensor}}, v_{2\text{sensor}}\},\ \{i_{1\text{sensor}}, i_{3\text{sensor}}, i_{3\text{sensor}}\} \text{ or } \{i_{1\text{sensor}}, v_{2\text{sensor}}, v_{2\text{sensor}}\}.$$

From this result it can be seen that if the sensors introduce new faults, some extra sensors may be required.

5.5 Conclusions

In the example in Section 5.4, it was possible to reach complete isolability of all faults with just 4 sensors. So the other 3 of the possible sensors can thus be left out without any affect to the diagnosis behaviour. If isolation of all faults is not necessary then the desired behaviour can be achieved with an even smaller number of sensors. If just detection of faults is asked for, then it is enough with one sensor in the example. If several sensor sets obtain the same diagnosis performance the user is free to chose between the sensor-sets along other optimization lines; cost, reliability, accessibility or some other criteria. So to conclude, from sensor placement analysis all sensor-sets that achieve the requested diagnosis performance are obtained. The sensor placement analysis is possible to do on models built along the lines described in Chapter 3, even large systems are not a problem if the reduction algorithm in Section 5.3 is used to reduce the model.

Chapter 6

Test Design

In this chapter it is shown how to identify sets of equations that may constitute a test. The actual construction of tests requires analysis of the analytical equations that are identified through the methods presented here. In Section 6.3 it is shown how a test can be constructed, illustrated by an example. In Section 6.2 it is explained how to obtain the analytical equations. In Section 6.4 it is shown how support for models with behavioural modes are handled in the test design analysis. Lastly in Section 6.5 the subject of weak and strong detectability is discussed.

To check if the process is functioning properly, i.e. according to its model, diagnosis analysis tests observations against the expected behaviour of the model to se if the two are consistent. Only parts with redundancy can be used for test construction, as it is only in redundant parts it is possible to measure or compute a variable in several ways and compare the results against each other. Parts with redundancy are equivalent to overdetermined systems of equations. After designing a model that support faults and subjecting it to sensor placement analysis as in Chapters 3 and 5, a test design analysis is used to obtain the actual residuals that can be constructed.

If a set of equations is overdetermined and ceases to be overdetermined if any of the equations are removed it is called a minimal overdetermined set. If the model is structural, as in this study, the set is called a minimal structural overdetermined set. For more information about structural models see Section 4.1. That is if a set of equations is overdetermined, diagnosis is possible on the part that the set represents and a minimal overdetermined set is the simpelest possible set to construct a test from.

Here is a small example of how to construct tests from a OpenModelica model, a larger example is presented in Section 6.3. Equations (6.1) is a model of a process. The system is overdetermined with two unknown variables and four equations.

$$\dot{x}_1 = -u \tag{6.1a}$$

$$\dot{x}_2 = -x_1 \tag{6.1b}$$

$$y_1 = x_2 \tag{6.1c}$$

$$y_2 = x_1 \tag{6.1d}$$

The set (6.1) contains three minimal overdetermined sets: {(6.1a), (6.1d)}, {(6.1a), (6.1b), (6.1c)} and {(6.1b), (6.1c), (6.1d)}

The residual that can be constructed from the first minimal overdetermined set is obtained by differentiating (6.1d), resulting in (6.2). The variable $\dot{x}_1$ is then eliminated by merging (6.1a) and (6.2) to (6.3). The residual r is obtained through (6.4), r is close to zero in the fault free case.

$$\dot{y}_2 = \dot{x}_1 \tag{6.2}$$

$$\dot{y}_2 = -u \tag{6.3}$$

$$r = \dot{y}_2 + u \tag{6.4}$$

When obtaining the overdetermined sets of equations that is desired to construct tests, the fault variables in the model is ignored. Since these are defined as faults and the purpose of the test design analysis is to find overdetermined sets just to be able to detect faults via residuals.

6.1 Minimal Structural Overdetermined Sets

To find overdetermined systems of structural equations, an algorithm that finds all minimal structural overdetermined sets is used. The algorithm is developed by Mattias Krysander, uses graph-theoretical tools and is efficient compared to previous algorithms [4].

It is recommended that the algorithm to find the minimal structural overdetermined sets are applied to the unreduced system. Thus the algorithm in Section 5.3 should not be used to reduce the system. Since, if the reduction algorithm is used, then the equations that should constitute the test may be the results of variable elimination. Thus the analytical equations that the structural system originated from may be harder to find when constructing the test since no corresponding elimination and reduction is done to the analytical system of equations. The reduction of the structural system in Chapter 5 is done to reduce the processing time of the sensor placement analysis algorithm. The algorithm that finds minimal structural overdetermined sets of equations does not analyse the same problem which is done much faster by comparison and there is no need to reduce the system to optimize processing time.

6.2 Identifying Analytical Equations

As described in Section 4.2, the flat OpenModelica code, consisting of the analytical equations, that describes the system is dumped along with the rest of the model representation from OpenModelica. The algorithm in Section 6.1 uses structural format and the result is thus in the form of the indexes of the interesting equations. But analytical equations are needed to reduce the system of equations to a residual. To transform from index format to analytical equations the idea is that it should simply be to take the analytic equations corresponding to the equation indexes from the minimal structural overdetermined set finding algorithm. But there are some problem with the ordering of the equations in the different formats. The OpenModelica compiler rearange the equations and variables at some point after that the analytical system of equations are dumped. So that the structural and analytical systems do not always agree in terms of the order of the equations. For now it is possible to identify the analytical equations by which variables they contain and thus manually sort out the minimal overdetermined set of analytical equations.

6.3 Test Design on a Circuit Example

The circuit used in Chapter 5 is utilized again here to exemplify how the test design results may look. Sensors are added to the first of the possible sensor positions sets that were found in Section 5.4, which obtained isolation of all faults. This set is referred to as 6.5.

$$\{v_{0\text{sensor}}, v_{2\text{sensor}}, i_{3\text{sensor}}, v_{3\text{sensor}}\} \tag{6.5}$$

The algorithm refered to in Section 6.1 is then applied to the system and finds a total of 41 minimal overdetermined sets of equations. One of these contain 13 equations. Of the 13 equations in this set, 11 are from OpenModelica, with the variables in OpenModelica notation, here in analytical form:

$$0.0 = R3.i - R3.p.i \tag{6.6a}$$
$$i3\_\text{sensor} = R3.p.i \tag{6.6b}$$
$$0.0 = R3.v - R3.p.v + R3.n.v \tag{6.6c}$$
$$R3.n.v = Op.out.v \tag{6.6d}$$
$$v3\_\text{sensor} = Op.out.v \tag{6.6e}$$
$$R3.p.v = C.p.v \tag{6.6f}$$
$$C.p.v = Op.in\_n.v \tag{6.6g}$$
$$0.0 = Op.in\_p.v - Op.in\_n.v + Op.Op\_\text{fault} \tag{6.6h}$$
$$G2.p.v = Op.in\_p.v \tag{6.6i}$$
$$G2.p.v = 0.0 \tag{6.6j}$$
$$0.0 = R3.v - (R3.R + R3.R\_\text{fault})R3.i \tag{6.6k}$$

In addition to these 11 equations, the two equations that represent the adding of sensors that measure the variables at the possible sensor locations $i_{3\text{sensor}}$ and $v_{3\text{sensor}}$ is part of the minimal overdetermined set:

$$i3\_\text{sensor} = y\text{-}i3\_\text{sensor} \tag{6.7a}$$
$$v3\_\text{sensor} = y\text{-}v3\_\text{sensor} \tag{6.7b}$$

The variables representing possible faults are ignored as discussed earlier, consequently $Op.Op_$fault in (6.6h) and $R3.R_$fault in (6.6k) are removed from the system. The quantity $R3.R$ is a known model parameter. The system consisting of (6.6) and (6.7) should thus be overdetermined and it should be feasible to construct a test from it.

By combining (6.6i) and (6.6j), (6.8a) is obtained. (6.8a) and (6.6h) together results in (6.8b). The value of $Op.in\_n.v$ in (6.8b) is inserted into (6.6g) and the value of $C.p.v$ is inserted into (6.6f) and accordingly (6.9a) and (6.8c) are obtained. By rearranging (6.6k) to (6.8d) and substituting $R3.v$ in (6.6c) with the right-hand side of (6.8d), (6.9b) becomes the result. The variable $R3.p.i$ is eliminated by combining (6.6a) and (6.6b) to (6.9c). The variable $Op.out.v$ is eliminated similary by combining (6.6d) and (6.6e) to (6.9d).

$$Op.in\_p.v = 0 \tag{6.8a}$$
$$0.0 = Op.in\_n.v \tag{6.8b}$$
$$0.0 = C.p.v \tag{6.8c}$$
$$R3.v = R3.R * R3.i \tag{6.8d}$$

$$0 = R3.p.v \tag{6.9a}$$
$$R3.p.v - R3.n.v = R3.R \cdot R3.i \tag{6.9b}$$
$$i3\_\text{sensor} = R3.i \tag{6.9c}$$
$$v3\_\text{sensor} = R3.n.v \tag{6.9d}$$

The system of equations (6.9) corresponds to (5.9d), (5.7c), (5.9g) and (5.7e) in Section 5.4. How the variables corresponds can be seen in Table 6.3.

Table 6.1: Corresponding name of variables representing the same quantity in Section 5.4 and Section 6.3.

Symbol in Section 5.4	Symbol in Section 6.3
i_3	R3.i
v_1	R3.p.v
v_3	R3.n.v
R_3	R3.R

To construct the residual that constitute the test, the variables that have represented possible sensor locations, $i3$_sensor and $v3$_sensor are replaced with the actual sensor variables y-$i3$_sensor and y-$v3$_sensor via (6.7a), (6.7b), (6.9c) and (6.9d) resulting in (6.10a) and (6.10b).

$$y\text{-}i3\_\text{sensor} = R3.i \tag{6.10a}$$

$$y\text{-}v3\_\text{sensor} = R3.n.v \tag{6.10b}$$

The left-hand side of (6.9a), (6.10a) and (6.10b) are inserted into (6.9b) thus becoming the residual in (6.11) after some rearranging. The residual in (6.10) should have a value close to zero when no faults have occurred in the circuit.

$$r = R3.R \cdot y\text{-}i3\_\text{sensor} + y\text{-}v3\_\text{sensor} \tag{6.11}$$

Thus a test is constructed that via the residual r check if the part of the circuit that is described by (6.6) is consistent with the model. If the fault variables $Op.Op$_fault or $R3.R$_fault is non-zero, they will affect the value of the residual r in accordance with (6.12) and the faults can be detected.

$$r = R3.R\_\text{fault} \cdot y\text{-}i3\_\text{sensor} - Op.Op\_\text{fault} \tag{6.12}$$

6.4 Behavioral Modes

Mattias Krysander and Jan Åslund at the Vehicular Systems group at the Institution of Electrical Engineering, LiTH, has developed an algorithm that handles behavioural modes. The representation of the modes is done via an assistance matrix that determines which equations are incompatible. The assistance matrix is an $n \times n$ matrix, were n is the number of equations in the system. A one, 1, at place (i,j) in the matrix denotes that equation i and equation j contradicts each other, i.e. they can not hold at the same time. This relation is symmetric, so a one is found at place (j,i) in the matrix as well. Consequentially the assistance matrix is always diagonally symmetric. The behavioural representation of the system in the behavioural mode matrix, see Section 4.2.1, is transformed to the assistance matrix form trough a MATLAB function named CreateAssistantMatrix.m. This function seeks trough the behavioural mode matrix and finds equations with the same condition variable, if they have different values the function concludes that they are incompatible. Thus the function can only handle simple conditions and only equivalent relations, i.e. only when the condition is that the condition variable should have exactly a value. If more advanced conditions are present in the model some manual manipulations are required to transform behavioural mode matrix-form to assistance matrix-form as of now.

The algorithm take the contradictions into account when determining the minimal structural overdetermined sets of equations. Here is a small example of a model with behavioural modes were the algorithm is used. OpenModelica code:

```
model MSObmex
  Real x1,x2,x3,y1,y2,u;
  String bm;
equation
  y1 = x2;        //e1
  y2 = x3;        //e2
  der(x1) = -x1 + x2 + u;     //e3
  der(x2) = if(bm == "A") then x1 - 2*x2
  else if (bm == "B") then -x2 + x3
  else 0;                     //e4
  der(x3) = x2 - 3*x3;        //e5
end MSObmex;
```

The model consists of five equations of which one is a conditional equation, three unknown variables, x1, x2 and x3, one input variable, u, and two output signals, y1 and y2. There is also a condition variable, bm.

The conditional equation is split into two equations on structural form so that the model has a total of six equations. The resulting equations gets numbered as number four and number five. The assistance matrix that keep track of the conflicting equations thus look like Matrix 6.13, with ones at places $(4, 5)$ and $(5, 4)$ in the matrix.

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{6.13}$$

As mentioned earlier in Section 6.2, the ordering of equations and variables is not quite intuitive after that the data has been dumped to structural form. In this particular example the unknown variables $x1 - -x3$ are stored in reversed order in the structural representation, compared to the order of declaration in the Modelica model. Matrix 6.14 is the structural representation of the six equations and three unknown variables. The equations are ordered as in the Modelica model, with the conditional equation split into two equations, numbered four and five. The equations (rows in the Matrix 6.14) are named as $\{e1, e2, e3, e4a, e4b, e5\}$, equation $e4a$ and $e4b$ originated from conditional equation $e4$ in the Modelica model. The variables (columns in the Matrix 6.14) are the unknown variables ordered like this: $\{x3, x2, x1\}$.

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 0 \end{bmatrix} \tag{6.14}$$

The result of the test design analysis on this model is six minimal structural overdetermined sets of equations, referred to as (6.15). It is possible to construct a test from each one of the sets in (6.15).

$$\begin{gathered}\{e2, e4b, e5\},\ \{e1, e4b, e5\},\ \{e1, e2, e5\}, \\ \{e1, e2, e4b\},\ \{e3, e4a, e1\} \text{ and } \{e2, e5, e3, e4a\}.\end{gathered} \tag{6.15}$$

6.5 Weak/Strong Detection Analysis

The detectability differs between different faults, one distinction that can be made is if the fault is weakly or strongly detectable. If the detectability of a fault is weak, it is possible to detect the fault, the fault will affect a residual. The affect may not be lasting. I.e. if one of the faults derivatives can be measured it is weakly detectable. A fault is strongly detectable if it affects a residual for a fault signal that has bounded final value that is non-zero. The definitions are taken from [2]. To determine if a fault is detectable in a strong or weak sense, how the different variables are differentiated must be booked in the representation of the system, as shown in Section 4.4, support for keeping track of the variables derivatives exist. In Figure 6.1 an example of a residual that is affected by a strongly detectable fault is shown, in Figure 6.2 an example of a residual that detects a weakly detectable fault is shown.

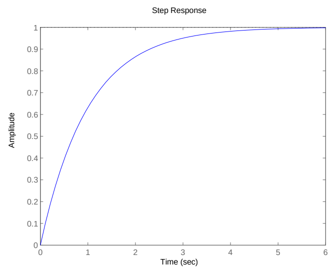

Figure 6.1: Response of a residual that detect a strongly detectable fault.

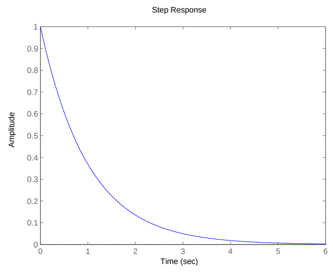

Figure 6.2: Response of a residual that detect a weakly detectable fault.

6.5.1 Example with Weak/Strong Fault Detection

Here is a small example to illustrate when a fault is weakly respectively strongly detectable. In (6.16) a small model described. The model consists of two unknown variables, one input variable, one output variable and one fault variable in three equations.

$$\begin{aligned}\dot{x}_1 &= -u\\ \dot{x}_2 &= -x_1 + f_1\\ y_1 &= x_2\end{aligned}\tag{6.16}$$

The system of equations (6.16) is overdetermined and the residual that it is possible to construct would be (6.17). In this example it is only possible to detect changes in the faults derivative via the observable variables. The fault f_1 is thus weakly detectable.

$$\ddot{y}_1 - u = \dot{f}_1\tag{6.17}$$

If the system would look like Example (6.18) instead, the residual would be (6.19) and the fault f_2 would be strongly detectable.

$$\begin{aligned}\dot{x}_1 &= -u + f_2\\ \dot{x}_2 &= -x_1\\ y_1 &= x_2\end{aligned}\tag{6.18}$$

$$u - \ddot{y}_1 = f_2\tag{6.19}$$

6.5.2 Automatically determine Weak/Strong fault detection

The information on how the fault variable is part of the residual equation can be automatically acquired via a algorithm that handles the derivatives of the variables in the model. The algorithm is written by Mattias Krysander. The structural representation of the system that is obtained from OpenModelica is automatically transformed into the format that the algorithm uses. The model representation should include differentiated variables, i.e. the derivative merge function described in Section 4.4 should not be applied to the system. The transformation of the structural representation to the new format is done by a MATLAB function called mssformat.m. The new format is not described in detail here, it is fairly straightforward to understand by looking at the MATLAB function. The information obtained about how the fault variables are included into the residual can be used to determine which tests are more attractive from a strong/weak-detectability view. It can also be deduced from this information if it is impossible to strongly detect a particular fault via the possible tests that can be performed.

6.6 Conclusions

Through the use of the algorithm in Section 6.1 the foundations, i.e. possible sets of equations, for constructing test can be obtained. In Section 6.3 an residual is constructed from the set of equation that the algorithm produced. If the model has support for behavioural modes, the algorithm in Section 6.4 can be used to sort out which tests that belong to which behavioural mode. Information about how the fault variables are included into the test can be obtained from the algorithm mentioned in Section 6.5.2, from that information it can be deduced if the fault is strongly or weakly detectable.

Some manual work is still necessary to construct the residuals, mainly identifying the analytical equations from the structural ones. Then to deduce the residual-equation from the set of equations obtained it is necessary to analytically manipulate the equations, something that lacks support in this study. There is also need for manually translate advanced behavioural mode-conditions to the assistance matrix form used for identifying minimal structural overdetermined sets when behavioural modes are used.

Chapter 7

Fault Modeling of and Sensor Placement Analysis on a Drive Line

In this chapter, the sequence of steps will be described of how to model a system with faults and then apply the sensor placement- and test design-analysis described earlier in Chapter 5 and 6 on that system. This chapter can hopefully serve as a form of user guide on how to utilize the contributions of this study.

7.1 General Notes on How to Design a Model with Support for Faults

First of all a model of the process or object is needed, if a model does not already exist then it is necessary to construct one. The model should describe the anticipated structural behaviour of the object, for more information about structural models see Chapter 4. After designing a model, it should be implemented in OpenModelica, using existing components and classes or by implementing new classes. All behaviour of the model should be described by using equations, conditional equations are allowed since they can be implemented by equations that holds under special circumstances, see Section 3.2, but algorithms should be avoided since they are not supported by the fault handling processes in this study.

The model should constitute a well posed problem with a single solution given all input and it should thus be possible to simulate the model so far. It is usually a good idea to simulate the model before it is extended with fault supporting features, it serves as an extra verification that the model behaves correctly and that all implementations into OpenModelica has been accepted by the compiler. Error searching of the model is much easier with help of OpenModelicas inbuilt error messages compared to trying to find them later. It gets harder to finding errors further into the process, since this may require in depth knowledge of all steps and formats. Some error handling is incorporated into the different functions in Matlab, but because two different programs and several representations of the data is used, it may be hard and demand much time to find errors when there is need to trace them back several steps.

After the model has been successfully implemented into Modelica it is time to extend it with possible faults and possible sensor locations. At this stage a conception about which faults that may affect the process and where they arise should be formed. This understanding is probably best fashioned together with an expert on the device or process in question. After an idea about which faults that are interesting and significant are formed, support for these faults should be incorporated into the model. One simple way to do this is to replace components where faults can occur with the corresponding components that have support for fault modeling if they exists. If no suitable component or model for the desired fault exist, the fault should be implemented by the user. See Chapter 3 for examples of components with support for faults that have been constructed in this study and general instructions regarding fault modeling.

7.2 Drive Line System

As an example to describe the process of fault modeling and some analysis, a drive line is used in this chapter. A drive line is the parts of a motor vehicle that generate power and delivers it to the road surface. This includes the engine, transmission, drive shafts, differentials, and the final drive. The drive line modeled in this chapter is for a hybrid car, that is a car that can be propelled by both a internal combustion engine that uses fuel (such as petroleum) and an electrical motor that gets its energy from electricity stored in a battery. There are fundamentally two kinds of hybrid drive lines; parallel and serial. In a parallel hybrid drive line, either the internal combustion engine or the electrical motor drives the vehicle. In a serial hybrid drive line both the internal combustion engine and the electrical motor can drive the vehicle at the same time. A serial one is used in this example.

Some possible faults that can affect a drive line were devised. One fault in the internal combustion engine, one in the battery, another in the electrical motor and a last one in a bearing of the shaft before the transmission. The faults were spread out over the drive line design to exemplify the diversity of faults that are possible to model and analyse.

7.3 Modeling a Drive Line

Now, when the kind of process to model and what kinds of faults that shall affect it has been decided, an actual model is required. In this case several models already exist of drive lines, so since model building is not central here, an existing model is used.

The drive line model used is initially from Vehicular Systems at LiTH. It is written in Modelica for use in Dymola, another Modelica tool. The model had to be modified and reduced compared to its original form. This because problems arise regarding some features that are not supported in OpenModelica, more about this in Section 8.1.

A basic model of the hybrid serial drive line is constructed by arranging a battery, an internal combustion engine, an electrical motor, a clutch, a gearbox and an inert load, as in Figure 7.1. The inert load is in place of a final drive i.e. wheels, because the modeling of wheels as well as the internal combustion engine in the original model introduces some of the problems discussed in Section 8.1.

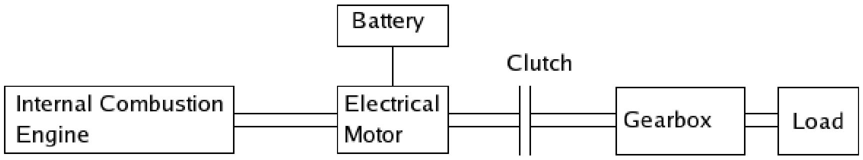

Figure 7.1: Schematic of the serial hybrid driveline used in this chapter.

If the models of the different components are supposed to exist for the moment, the basic Modelica code for connecting the components according to the schematic shown in Figure 7.1 looks like this:

```
connect(battery.PosPin, dcmotor.PosPin);
connect(battery.NegPin, dcmotor.NegPin);
connect(dcmotor.Torque, clutch.flange_a);
connect(engine.Torque, clutch.flange_a);
connect(clutch.flange_b, gearbox.flange_a);
connect(gearbox.flange_b, load.flange_a);
```

As mentioned earlier in this section, the internal combustion engine model had some problems regarding OpenModelica. Thus the internal combustion engine is described by a new simple model that is solely for diagnostic purposes using structural methods. Since the model is just used for diagnosis analysis, the equations describing the behaviour needs only to be structurally correct. That is why some equations regarding the internal combustion engine may not look correct from a engine modeling point of view. But the equations give the relevant structural information. For more on this issue see Chapter 4.

Figure 7.2 shows an outline of the internal combustion engine and Table 7.1 shows explanations of the symbols used in the modeling of the engine. A servo with a time constant steers the throttle α from reference signal α_{ref}, see Equation 7.1a. The throttle angle, α, together with the inlet air pressure, p, determines the air mass flow past the throttle, w_{in}, see Equation 7.1b. The air mass flow into the cylinders, w_{cyl}, is determined by the engine speed, N, and the pressure p, see

Equation 7.1c. The difference in air mass flow past the throttle, w_{in} and air mass flow into the cylinders, w_{cyl} constitutes the change of the air pressure, p, i.e. $\dot{p}$, see Equation 7.1d. The air mass flow into the cylinders, w_{cyl} relates directly to the engines output torque τ_{eng}, see Equation 7.1e. The difference between the engines output torque, τ_{eng}, and the load of the engine, τ_{load}, results in the change of engine speed, N, see Equation 7.1f. All equations for the engine model are derived from [5].

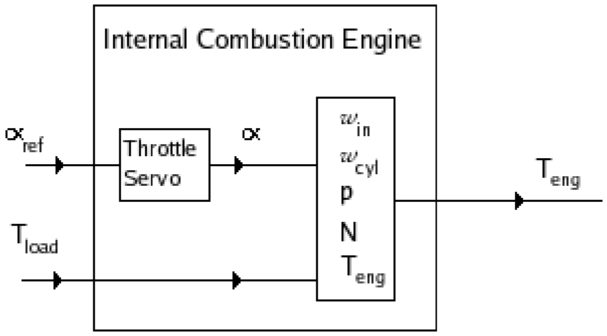

Figure 7.2: Schematic of internal combustion engine.

Quantity	Symbol
Engine speed	N
Inlet air pressure	p
Air mass flow past throttle	w_{in}
Air mass into the cylinders	w_{cyl}
Throttle angle	α
Throttle reference angle	α_{ref}
Engine output torque	τ_{eng}
Engine load	τ_{load}

Table 7.1: Notations and variables in the internal combustion engine model.

The equations describing the structure of the simplified internal combustion engine.

$$0.1\dot{\alpha} = -\alpha + \alpha_{ref} \tag{7.1a}$$
$$w_{in} = p \cdot cos\alpha \tag{7.1b}$$
$$w_{cyl} = N \cdot p \tag{7.1c}$$
$$\dot{p} = w_{in} - w_{cyl} \tag{7.1d}$$
$$\tau_{eng} = w_{cyl} \tag{7.1e}$$
$$\dot{N} = \tau_{eng} - \tau_{load} \tag{7.1f}$$

Here is an implementation of the internal combustion engine in Modelica code:

```
class SimpleEngine
  "Simple model of a combustion engine"
  Real p,w_in,w_cyl,N,alpha,tau_eng,tau_load;
  Modelica.Mechanics.Rotational.Interfaces.Flange_b Torque;
  Real alpha_ref;
  Real alphatime;
equation
    0.1*der(alpha) = - alpha + alpha_ref;
    w_in = p*cos(alpha);
    w_cyl = N*p;
    der(p) = w_in - w_cyl;
    tau_eng = w_cyl;
    der(N) = tau_eng - tau_load;
    der(Torque.phi) = N;
    Torque.tau + tau_load = 0;
end SimpleEngine;
```

The last two equations are just for connecting the engine with its flange interface, they are not represented in the Equation system 7.1.

The battery is modeled as a simple electrical circuit with a constant voltage source to represent the battery's chemical charging, a capacitance and a resistance. See Figure 7.3. The battery circuit is described by the basic electrical Equations 7.2.

$$C\frac{d}{dt}(v_0 - v_1) = i_1 \tag{7.2a}$$
$$v_1 - 0 = R \cdot i_1 \tag{7.2b}$$
$$v_0 = u \tag{7.2c}$$
$$i_0 = i_1 + i_2 \tag{7.2d}$$
$$i_0 + i_3 = i_1 + i_2 \tag{7.2e}$$

The Modelica code for the model of the battery is basically connections of the premade existing components from the standard Modelica library to create a battery as shown in Figure 7.3. As well as creating connecting interfaces for the

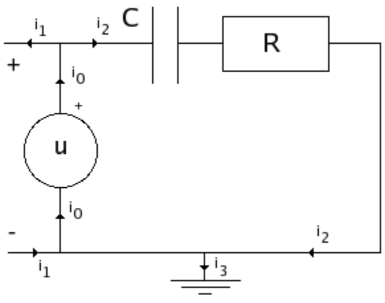

Figure 7.3: Schematic of battery model.

battery to connect to the electrical motor. Here is the Modelica implementation of the schematic seen in figure 7.3, built with basic electrical components from the Modelica standard library.:

```
class SimpleBattery
  Modelica.Electrical.Analog.Basic.Capacitor C(C=1);
  Modelica.Electrical.Analog.Basic.Resistor R(R=1);
  Modelica.Electrical.Analog.Sources.ConstantVoltage U;
  Modelica.Electrical.Analog.Interfaces.PositivePin PosPin;
  Modelica.Electrical.Analog.Interfaces.NegativePin NegPin;
  Modelica.Electrical.Analog.Basic.Ground G;
  equation
    connect(U.p,C.n);
    connect(C.p,R.p);
    connect(U.n,R.n);
    connect(U.p,PosPin);
    connect(U.n,NegPin);
    connect(U.n,G.p);
end SimpleBattery;
```

The electrical motor consists of an inductance, a resistance, an electromotive force (Emf) that transform electrical energy to mechanical energy, and a flange for connecting the motor to a mechanical interface. See Figure 7.4. The Equations 7.3 describes the behaviour of the electrical motor. The k in Equations 7.3c and 7.3d is a constant describing the efficiency in the conversion from electrical energy to mechanical work in the Emf. k is set to 1 by default. Once again, the actual value, if it is known, is of no consequence as long as only structural methods are used.

$$L\frac{d}{dt}i = v_0 - v_1 \tag{7.3a}$$

$$v_1 - v_2 = R \cdot i \tag{7.3b}$$

$$\dot{\varphi} \cdot k = v_2 - v_3 \tag{7.3c}$$

$$\tau = -k \cdot i; \tag{7.3d}$$

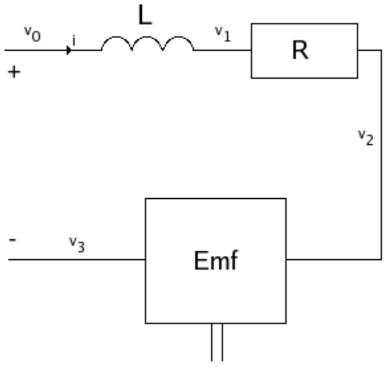

Figure 7.4: Schematic of direct current electrical motor.

Similarly to the Modelica code for the battery model, the code for the electrical motor is basically just connections between the components that constitutes the motor. The actual code is not shown here. Refer to the code for the battery and the schematic shown in Figure 7.4.

The standard Modelica library contain a clutch model, which is used. The clutch model exists of two flanges with friction between them. The two flanges are pressed together with a normal force. The normal force is determined by a normalized input signal, with value between 0 and 1. The model include some events, i.e. conditional equations, to handle the different modes of the component; still, moving forward and moving backward. The torque transfered from one flange to the other can be somewhat simplified described by Equation 7.4. The symbols in Equation 7.4 are explained in Table 7.2.

$$\tau_\mu = c_{\text{geo}} \cdot \mu_{\text{w}} \cdot F_n \tag{7.4}$$

The actual Modelica code is not included here, but can be accessed from the Modelica standard library [1].

The gearbox is taken directly from VehProLib. It consists of five clutches and five gears as well as some additional components to model the internal inertia and losses of efficiency in the gearbox transmission. Equations 7.5 describes the basic behaviour of the gearbox model. The symbols in Equations 7.5 are explained in

Quantity	Symbol
Torque transfered between flanges	τ_μ
Constant depending on geometry	c_{geo}
Friction value	μ_{w}
Normal force	F_n

Table 7.2: Notations and variables in clutch model.

Table 7.3.

$$\varphi_a = c_{\text{ratio}} \cdot \varphi_b \tag{7.5a}$$

$$0 = c_{\text{ratio}} \cdot \tau_a + \tau_b \tag{7.5b}$$

Quantity	Symbol
Rotation angel of driving respective driven flange	φ_a, φ_b
Transmission ratio (depends on which gear is used)	c_{ratio}
Torque of driving respective driven flange	τ_a, τ_b

Table 7.3: Notations and variables in Gearbox model.

The Modelica code of the gearbox model is not included here since it is very large and would take up to much space.

7.3.1 Addition of possible faults

A number of possible faults are added to the model. One is a possible fault in the servo that steers the throttle angle. The fault variable $\alpha\text{time}_{\text{fault}}$ is introduced. It is added to the Equation 7.1a that describes the servo. Thus modeling a change in the time constant in the servo.

$$(0.1 + \alpha\text{time}_{\text{fault}}) \cdot \dot{\alpha} = -\alpha + \alpha_{\text{ref}} \tag{7.6}$$

The Modelica code modeling the throttle servo in the internal combustion engine in Section 7.3 is thus altered to this:

```
(0.1+alphatime)*der(alpha) = - alpha + alpha_ref;
```

And the new variable $\alpha\text{time}_{\text{fault}}$ is connected via an equation to a variable with a name following the naming convention dissused in Section 4.3.

```
alphatime_fault = engine.alphatime;
```

The model of the battery is modified by replacing the voltage source with a potentially faulty one: FaultyConstantVoltage. This modified component contain support to model a drop in the battery's charging capability. The model of the faulty voltage source contains several behavioural modes and is described in Section 3.3. One of these modes introduce a faulty voltage in the voltage source, in equation form this is described by introducing a new variable. Equation 7.2c is changed to 7.7.

$$v_0 = u + u_{\text{fault}} \tag{7.7}$$

Regarding the actual Modelica code, the ConstantVoltage component in Section 7.3 is just changed to FaultyConstantVoltage.

```
Modelica.Electrical.Analog.Sources.FaultyConstantVoltage U;
```

The electrical motor is changed in a similar way by replacing the Emf with a potentially faulty one. The potentially faulty Emf has support to model a lowered conversion efficiency in the Emf. A new variable k_{fault} is introduced to model the fault and Equation 7.3c is changed to Equation 7.8.

$$\dot{\varphi} \cdot (k + k_{\text{fault}}) = v_2 - v_3 \tag{7.8}$$

Again it is just to change the Modelica code from EMF to FaultyEMF in the model of the electrical motor.

```
Modelica.Electrical.Analog.Basic.FaultyEMF Emf;
```

To model some mechanical faults, a coupling is inserted between the clutch and gearbox. The coupling can potentially become stuck in its housing or become unhinged and separated. It is called FrictionCoupling and should have a variable that describes the way a unknown friction may affect the model. These faults are added in the way of a modified brake, of which a base model already exists in the standard Modelica library. The standard brake is modeled in a similar way as the clutch and the braking force, which models the unknown friction force that may arise due to a fault, is simply connected to a fault variable F_{nfault} by Equation 7.9b, for explanations of symbols in Equations 7.9 see Table 7.4, in Modelica code it looks like this, with the fault variable named following the naming convention described in Section 4.3:

```
FrictionCoupling.f_normalized = friction_fault;
```

$$\tau_{\mu} = c_{\text{geo}} \cdot \mu_{\text{w}} \cdot F_n \tag{7.9a}$$
$$F_n = F_{\text{nfault}} \tag{7.9b}$$

The modified brake model is inserted into the drive line model between the clutch and gearbox. In Modelica code the connection between the clutch and gearbox shown in Section 7.3:

```
connect(clutch.flange_b, gearbox.flange_a);
```

is changed to:

Quantity	Symbol
Braking torque	τ_μ
Constant depending on geometry	c_{geo}
Friction value	μ_{w}
Normal force	F_n
Introduced fault	F_{nfault}

Table 7.4: Notations and variables in the brake model.

```
connect(clutch.flange_b, FrictionCoupling.flange_a);
connect(FrictionCoupling.flange_b, gearbox.flange_a);
```

The different added potential faults represents the different approaches to fault modeling that has been discussed in Section 3.1. First the αtime$_{\text{fault}}$ which is simply added as a variable in the existing model. Secondly the electrical faults which are incorporated through modified models for voltage source and Emf which both use the same base class for inheritance, FaultyOnePort. The mechanical faults in their turn are added via a new component that is for fault modeling purposes only. In this instance, a modified brake, whose basic model already exist in the standard Modelica library. The brake force is used to model a unpredicted friction force. So to conclude, a total of four potential faults, shown in Table 7.5 is added to the model of the drive line. Figure 7.5 shows a schematic of the drive line with the four potential faults as well as the three inputs indicated.

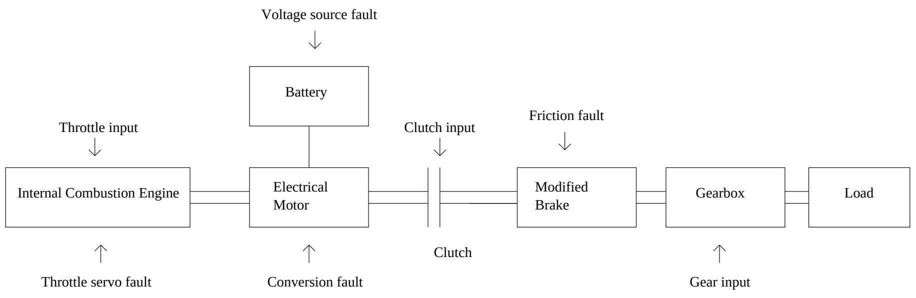

Figure 7.5: Schematic of drive line with potential faults.

Nothing hinders more faults to be added, all potential faults can later be removed or ignored depending on the analysis specifications. I.e. the model can contain support for a special fault that the user is not interested in at the moment. That fault can then be disregarded during the analysis. As long as the faults are implemented according to the standard described in Section 4.3.

Quantity	Symbol
Fault in the internal combustion engines servo	$\alpha\mathrm{time}_{\mathrm{fault}}$
Fault in the batterys charging effect	u_{fault}
Fault in the electrical motors conversion effeciency	k_{fault}
Fault in a bearing of the driveline	F_{nfault}

Table 7.5: Overview of potential faults in the drive line model.

7.3.2 Addition of possible sensor locations and handling of known input signals

Possible locations for sensors are added. The possible sensors that are added are listed in Table 7.6, with the corresponding variable that is measured.

Sensor name	Measured signal	Modelica equation
$\mathrm{N}_{\mathrm{sensor}}$	Engine speed of the internal combustion engine.	engine.N = N_sensor;
$\mathrm{p}_{\mathrm{sensor}}$	Inlet manifold pressure in the internal combustion engine.	engine.p = p_sensor;
$\mathrm{v}_{\mathrm{sensor}}$	Speed of the final drive.	der(load.flange_b.phi) = v_sensor
$\mathrm{V}_{\mathrm{sensor}}$	Electrical potential of the batterys positive pin.	battery.PosPin.v = V_sensor;
$\mathrm{i}_{1\mathrm{sensor}}$	Electrical current through the electrical motor.	dcmotor.Emf.i = i1_sensor;
$\mathrm{i}_{2\mathrm{sensor}}$	Electrical current through the resistance in the battery.	battery.R.i = i2_sensor;

Table 7.6: Possible sensors and the quantity that they measure along with Modelica code for connecting them to the corresponding variable.

Similar to adding faults, sensors can be added and then ignored in the analysis. Furthermore, the different input the driver is responsible for is considered known, and this fact is modeled by just giving the actual signals a name corresponding with the naming convention discussed in Section 4.3. In structural form this becomes a equation with just one variable, which means that its determined and considered known in the system. The input signals and their corresponding Modelica assignments are shown in Table 7.7. These input signals can be automatically handled by the sensor placement program in MATLAB as long as they are named along the lines discussed in Section 4.3.

Input Symbol	Input signal	Modelica equation
α_{ref}	Throttle position.	engine.alpha_ref = alpha_known;
clutch$_{input}$	Clutch input.	clutch_input = clutch_known;
gear$_{input}$	Gear position.	gear_input = gear_known;

Table 7.7: Known variables, their symbols and Modelica code for connecting input variables to the corresponding known variable.

7.4 Analysis

The system generated by Modelica consist of 328 equations and 338 variables and is thus underdetermined. It has 10 more variables than equations. The faults that are introduced in Section 7.3.1 account for four of the extra variables. The input variables listed in Table 7.7 account for another three variables.

The last three of the 10 extra variables are condition variables that are introduced by the component models with support for different behavioural modes, see Section 3.2 for more information about behavioural modes. As explained in Section 5.1, SensPlaceTool does not have support for behavioural modes and therefore all if-equations are merged and the belonging condition variables are removed. See Section 4.2.1 for more information regarding the handling of conditional equations.

If these special variables are removed, the system of the drive line is a well posed problem with 328 equations and 328 variables. The possible sensor variables and their equations are included in these numbers. The sensor placement function handles these special variables and the system contain the basic 328 equations and variables as well as the 4 variables that represents faults when the sensor placement analysis is applied on it.

7.4.1 Sensor placement

Because of the size of the example and SensPlaceTools problems with large models, some invariant reductions, regarding the result, of the system are in order to cut down the processing time for the algorithm, see Section 5.3 for more on the reduction of structural systems.

Result of the reduction of the system:

If SensPlaceTool is applied to the unreduced system of 328 equations and 332 variables, the computing time is well over three hours, measured with PROFILER in MATLAB.

After the reduction algorithm described in Section 5.3 is applied to the system, it is reduced to 22 equations and 26 variables. The system is still underdetermined with the 4 faults making the difference. The time for the sensor placement algorithm is now at six seconds with the reduction algorithm accounting for another second. The analysis is done on a computer running a 2.2 Ghz processor and 512 MB RAM memory.

The sensor placement result: To detect all faults one of these sensors is required:

$$p_{\text{sensor}},\ N_{\text{sensor}},\ v_{\text{sensor}},\ \text{or}\ i_{1\text{sensor}}$$

To isolate all faults from each other, one of these sensor sets is required:

$$\{V_{\text{sensor}},\ p_{\text{sensor}},\ v_{\text{sensor}}\},\ \{V_{\text{sensor}},\ i_{1\text{sensor}},\ p_{\text{sensor}}\},$$
$$\{i_{2\text{sensor}},\ p_{\text{sensor}},\ v_{\text{sensor}}\},\ \{i_{2\text{sensor}},\ i_{1\text{sensor}},\ p_{\text{sensor}}\}$$
$$\{V_{\text{sensor}},\ i_{1\text{sensor}},\ v_{\text{sensor}}\}\ \text{or}\ \{i_{2\text{sensor}},\ i_{1\text{sensor}},\ v_{\text{sensor}}\}$$

7.4.2 Test design

The algorithm described in Section 6.1 is used to obtain the minimal overdetermined sets of equations. The sensors from the first of the sensor sets in Section 7.4.1 that results in isolability of all faults is used:

$$\{V_{\text{sensor}},\ p_{\text{sensor}},\ v_{\text{sensor}}\}$$

If these sensors are added to the system it will yield a total of 86 minimal structural overdetermined sets of equations. The frequency of sets according to the number of equations in them can be seen as a histogram in Figure 7.6. The majority of equations in these sets are very simple equations and the sets of equations are usually easily reducible to a more modest amount of equations. So the relatively large number of equations per set is not a problem.

The set with the least amount of equations of these sets contain 10 equations. The equations in this set on analytical form from the Modelica model can be seen here:

```
0.0 = battery.U.v + -battery.U.V + battery.U.V_fault

0.0 = battery.U.v + -battery.U.p.v + battery.U.n.v

battery.G.p.v = 0.0;

battery.U.n.v = battery.R.n.v;

battery.R.n.v = battery.NegPin.v;

battery.NegPin.v = battery.G.p.v;

battery.U.p.v = battery.C.n.v;

battery.C.n.v = battery.PosPin.v;

V_sensor = battery.PosPin.v;

V_sensor = y-V_sensor;
```

The first two equations are modified, in the original Modelica model they are parts of larger if-expressions, here is shown the part of the if-expression that holds in these circumstances. (The two equations describes the behaviour of a voltage source component, that can be affected by a fault in the voltage input but otherwise functions as it should). In the first equation the battery.U.V variable value is known and the battery.U.V_fault variable represents the possible fault and is disregarded when designing possible sets of equations that may constitute a test set. The last equation is added in the sensor placement process to represent that a actual sensor is placed at the possible sensor location that V_{sensor} symbolize. In this particular case it can be seen that the 10 equations above define a overdetermined system of equations. The actual residual would look like this:

```
  r = battery.U.V - y-V_sensor;
```

Where the residual variable r should be close to zero if this part of the process is functioning in a way that is consistent with the model.

The equations from all minimal structurally overdetermined sets can at least in a structural sense always be used to design a test. Thus, from the 86 different sets of minimal structural overdetermined sets of equations, 86 possible tests can in theory be designed. For more information about test design and obtaining analytic equations, see Chapter 6.

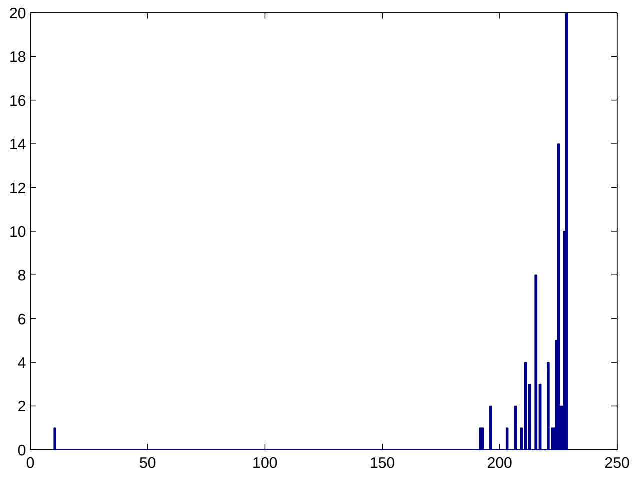

Figure 7.6: Histogram over the number of equations in the 86 minimal overdetermined sets generated by the sensor set in Section 7.4.2.

7.5 Conclusion

In this Chapter a fairly large Modelica model of a drive line were designed. A variety of possible faults and possible sensor locations were added. The system of equations that Modelica produced from this model underwent sensor placement analysis. The possibility to reduce the system of equations were shown to cut down the processing time of the sensor placement analysis by a significant amount. Then one of the sensor placement solutions were applied to the system and all the different minimal structural overdetermined sets were extracted.

Chapter 8

Discussion

Modelica was chosen as the environment to develop fault models in because, as stated in Chapter 3 it allows the user much freedom in designing models and the systems can be used for diagnosis design purposes even though they are not possible to simulate. Simulink is another alternative as a possible environment to use. One of the advantages Modelica have over Simulink is that in Modelica it is not necessary to define the causality, meaning that there is no need to define the direction of assignments in the model, which is useful because the new extended models with support for faults may have different types of behaviour, e.i. introductions of fault may change the "flow" of assignments and if the model-language is causal, then a new model must be designed for each "flow change".

OpenModelica is a compiler for the computer language Modelica. Another alternative to compile Modelica code in could be MathModelica. But MathModelica is a commercial product, so cost and availability becomes a issue and it does not give the same freedom of in-depth changes as an open-source product. OpenModelica is developed at PELAB, Programming Environments Laboratory, at, IDA, the Institution of Computer Sience, at LIU, Linköpings University and thus much assistance could be received from the developers and changes could quite straightforwardly be implemented into the OpenModelica compiler.

For analysis of the data Matlab is used. Matlab is straightforward to use and well known by most potential users. Further on, the tools and algorithms from others that are used in this study were already implemented in Matlab-code, making it a natural and convenient choice to continue development in.

8.1 Problems in OpenModelica

This section summarize the problems that was encountered in OpenModelica in this study.

As stated in Section 3.5 OpenModelica can not at the time of this study manage string equality, this will according to the developers be taken care of in the future. If a fault model should be simulated it needs to be modified as of now so that the condition variables are some other type than string. Modifications are also necessary in the terms of the extra equations that the behavioural modes of the different components introduce so that these do not cause a overdetermined system. The fault variables need also to be instantiated for the simulation to work, see Section 3.5. The order of the equations in analytical form does not agree as of now with the order of the structural equations from OpenModelica, as seen in Section 6.2, the exact cause of these inequalities in ordering has not been fully determined in this thesis. The analytical equations are not essential in the diagnosis analysis done in this study, furthermore the format of a list with analytical equations is probably not specially useful for analytical analysis. Thus the problem of how the analytical representation should look is left as future work. There seems to be lacking some compatibility of the tables that where part of the original drive line model that is used in Chapter 7, this is however not a problem in the fault modeling sense. The fault free models that are used as bases to expand to fault models through the methods presented in this thesis are assumed to be compatible with OpenModelica.

8.2 Conclusions

As seen in the previous chapters, the approach of designing fault models in OpenModelica and then exporting the corresponding system and perform diagnosis analysis on it in Matlab is feasible. But it is not without problems, the process includes several user interfaces and the data is transformed between several representations. This process of model building and diagnosis analysis via Modelica can thus be a bit lengthy. Here the freedom Modelica gives the designer in modeling may be a cause of concern, since all the different ways to model in Modelica should be compatible with the later stages of transformation of the data to a format that suits the diagnosis analysis algorithms. In this study quite some effort has been spent to make sure that the conditional equations, in Modelica modeled with if-expressions, are handled in a way that maintain the information they convey. In the future the same amount of work may have to be put into making sure that other Modelica formats get treated right, e.g when-statements.

The way to design fault models with the use of a standard library with components that has support for faults is tried in Chapter 3 and used in Chapters 5 and 7, this approach is also feasible but encounter some problems. It is a daunting task to implement all imaginable faults in all components, and even if the components in them self are considered complete in a fault support sense, new faults are introduced higher up in the hierarchy of a system when the components the system consist of interact with each other. Furthermore, a system with support for a great variety of faults may get very large with a multitude of variables that may not be used. The size of the system is in it self not a problem, as seen in Chapter 7 the analysis algorithms can handle quite large systems with the help of the reduction method in Section 5.3, but may cause some confusion and become difficult to overlook and understand. As an alternative only, the interesting faults could be implemented, these should be deviced in cooperation with a domain expert.

If the results from this thesis are to be used in the future will depend on if the freedom that Modelica gives in designing fault models are deemed worthwhile to accommodate so that the results from the modeling suits the diagnosis analysis. One way to do ease this transformation of data is to restrict the freedom of model designs in Modelica but then some of the benefit that Modelica brings is negated. It is shown here however that it is possible to design fault models in Modelica and then use the results to analyze the models regarding sensor placement and test design.

8.3 Future work

The communication between OpenModelica and Matlab, which as of now only is a one way communication via the process described in Section 4.2, should in the ideal case be expanded so that the diagnosis analysis results from the algorithms described in Chapters 5 and 6 is automatically transfered and implemented into the OpenModelica model. For this to work, some functionality of the handling of analytical equations is probably necessary. For instance the analytical reduction of minimal overdetermined systems of equations to residual equations that is done in Sections 6.3 and 7.4.2 would be good if it could be more automatic.

There exists as of now only basic support for handling of logical expressions in conditional equations, more advanced logic sentences containing several variables and operations such as AND and OR is not automatic handled. Thus some more functionality to handle logic would be necessary to automatizes the test design with support for behavioural modes in Section 6.4 for instance.

If the approach of a fault library is pursued the library of components that has support for faults could also be expanded in the future to include more advanced components.

Bibliography

[1] Modelica Association. http://www.modelica.org/libraries.

[2] Mogens Blanke, Michel Kinnaert, Jan Lunze, and Marcel Staroswiecki. *Diagnosis and Fault-Tolerant Control*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.

[3] Gertler J. and Singer D. A new structural framework for parity equation based failure detection and isolation. *Automatica*, 26:381–388, 1990.

[4] Mattias Krysander. *Design and Analysis of Diagnosis Systems Using Structural Methods*. PhD thesis, Linköpings universitet, June 2006.

[5] Lars Nielsen Lars Eriksson. *Vehicular Systems*. Vehicular Systems, ISY Linköping Institute of Technology, 2007.

[6] Meyer M, Le Lann JM, Koehret B, and Enjalbert M. Optimal selection of sensor-location on a complex-plant, using a graph oriented approach. *Computers & Chemical Engineering*, 18:535–540, 1994.

Appendix A

Appendix A. . .

System of equations produced by the OpenModelica compiler from the model of a circuit discussed in Chapter 5

```
equation
  0.0 = if C.bm == "NF" then C.i - C.C * der(C.v) else if C.bm ==
"FaultyResistor" then C.i - (C.C + C.C_fault) * der(C.v) else 0.0;
 0.0 = if C.bm == "NF" then C.i - C.p.i else if C.bm == "ShortCircuit"
then C.i - C.p.i else if C.bm == "OpenCircuit" then C.i else 0.0;
  0.0 = if C.bm == "NF" then C.v + -C.p.v + C.n.v else if C.bm ==
"ShortCircuit" then C.v + -C.p.v + C.n.v else 0.0;
 0.0 = if C.bm == "NF" then C.p.i + C.n.i else if C.bm == "ShortCircuit"
then C.p.i + C.n.i else 0.0;
  0.0 = if R1.bm == "NF" then R1.v - R1.R * R1.i else if R1.bm ==
"FaultyResistor" then R1.v - (R1.R + R1.R_fault) * R1.i else 0.0;
 0.0 = if R1.bm == "NF" then R1.i - R1.p.i else if R1.bm == "ShortCircuit"
then R1.i - R1.p.i else if R1.bm == "OpenCircuit" then R1.i else 0.0;
  0.0 = if R1.bm == "NF" then R1.v + -R1.p.v + R1.n.v else if R1.bm ==
"ShortCircuit" then R1.v + -R1.p.v + R1.n.v else 0.0;
 0.0 = if R1.bm == "NF" then R1.p.i + R1.n.i else if R1.bm == "ShortCircuit"
then R1.p.i + R1.n.i else 0.0;
  0.0 = if R2.bm == "NF" then R2.v - R2.R * R2.i else if R2.bm ==
"FaultyResistor" then R2.v - (R2.R + R2.R_fault) * R2.i else 0.0;
 0.0 = if R2.bm == "NF" then R2.i - R2.p.i else if R2.bm == "ShortCircuit"
then R2.i - R2.p.i else if R2.bm == "OpenCircuit" then R2.i else 0.0;
  0.0 = if R2.bm == "NF" then R2.v + -R2.p.v + R2.n.v else if R2.bm ==
"ShortCircuit" then R2.v + -R2.p.v + R2.n.v else 0.0;
  0.0 = if R2.bm == "NF" then R2.p.i + R2.n.i else if R2.bm ==
"ShortCircuit" then R2.p.i + R2.n.i else 0.0;
  0.0 = if R3.bm == "NF" then R3.v - R3.R * R3.i else if R3.bm ==
"FaultyResistor" then R3.v - (R3.R + R3.R_fault) * R3.i else 0.0;
 0.0 = if R3.bm == "NF" then R3.i - R3.p.i else if R3.bm == "ShortCircuit"
then R3.i - R3.p.i else if R3.bm == "OpenCircuit" then R3.i else 0.0;
```

```
  0.0 = if R3.bm == "NF" then R3.v + -R3.p.v + R3.n.v else if R3.bm ==
"ShortCircuit" then R3.v + -R3.p.v + R3.n.v else 0.0;
  0.0 = if R3.bm == "NF" then R3.p.i + R3.n.i else if R3.bm ==
"ShortCircuit" then R3.p.i + R3.n.i else 0.0;
  0.0 = if R4.bm == "NF" then R4.v - R4.R * R4.i else if R4.bm ==
"FaultyResistor" then R4.v - (R4.R + R4.R_fault) * R4.i else 0.0;
  0.0 = if R4.bm == "NF" then R4.i - R4.p.i else if R4.bm == "ShortCircuit"
then R4.i - R4.p.i else if R4.bm == "OpenCircuit" then R4.i else 0.0;
  0.0 = if R4.bm == "NF" then R4.v + -R4.p.v + R4.n.v else if R4.bm ==
"ShortCircuit" then R4.v + -R4.p.v + R4.n.v else 0.0;
  0.0 = if R4.bm == "NF" then R4.p.i + R4.n.i else if R4.bm ==
"ShortCircuit" then R4.p.i + R4.n.i else 0.0;
  G.p.v = 0.0;
  0.0 = if Op.bm == "NF" then Op.in_p.v - Op.in_n.v else if Op.bm ==
"FaultyOpAmp" then Op.in_p.v + -Op.in_n.v + Op.Op_fault else 0.0;
  Op.in_p.i = 0.0;
  Op.in_n.i = 0.0;
  u.signalSource.T_width = (u.signalSource.period * u.signalSource.width)
/ 100.0;
  when sample(u.signalSource.startTime,u.signalSource.period) then
  u.signalSource.T0 = time;
  end when;
  u.signalSource.y = u.signalSource.offset + (if time <
u.signalSource.startTime OR time >= u.signalSource.T0 +
u.signalSource.T_width then 0.0 else u.signalSource.amplitude);
  0.0 = if u.bm == "NF" then u.v - u.signalSource.y else if u.bm ==
"FaultyVoltageSource" then u.v + -u.signalSource.y + u.Voltage_fault
else 0.0;
  0.0 = if u.bm == "NF" then u.i - u.p.i else if u.bm == "ShortCircuit"
then u.i - u.p.i else if u.bm == "OpenCircuit" then u.i else 0.0;
  0.0 = if u.bm == "NF" then u.v + -u.p.v + u.n.v else if u.bm ==
"ShortCircuit"
then u.v + -u.p.v + u.n.v else 0.0;
  0.0 = if u.bm == "NF" then u.p.i + u.n.i else if u.bm == "ShortCircuit"
then u.p.i + u.n.i else 0.0;
  v0_sensor = u.p.v;
  v1_sensor = R1.n.v;
  v1c_sensor = Op.in_n.v;
  v1a_sensor = C.p.v;
  v1b_sensor = R3.p.v;
  i2_sensor = C.p.i;
  i3_sensor = R3.p.i;
  v3_sensor = C.n.v;
  v4a_sensor = R2.n.v;
  v4b_sensor = R3.n.v;
  v4_sensor = Op.out.v;
```

```
v2_sensor = Op.in_p.v;
i1_sensor = u.p.i;
R4.n.i + Op.in_p.i = 0.0;
R4.n.v = Op.in_p.v;
u.n.i + G.p.i + R4.p.i = 0.0;
u.n.v = G.p.v;
G.p.v = R4.p.v;
R2.n.i + R3.n.i + Op.out.i = 0.0;
R2.n.v = R3.n.v;
R3.n.v = Op.out.v;
C.n.i + R2.p.i = 0.0;
C.n.v = R2.p.v;
R1.n.i + R3.p.i + C.p.i + Op.in_n.i = 0.0;
R1.n.v = R3.p.v;
R3.p.v = C.p.v;
C.p.v = Op.in_n.v;
u.p.i + R1.p.i = 0.0;
u.p.v = R1.p.v;
```