Model-based diagnosis using MathModelica

Per Åkerlund

Reg.nr: LiTH-ISY-EX-3150
2001-06-01

Model-based diagnosis using MathModelica

Examensarbete utfört på Fordonssystem
vid Linköpings Tekniska Högskola
av

Per Åkerlund

Reg.nr: LiTH-ISY-EX-3150

Handledare: Mattias Nyberg, LiTH
Mats Jirstrand, MathCore AB

Examinator: Mattias Nyberg, LiTH

Linköping den 1 juni 2001

Avdelning, Institution
Division, department

Department of Electrical Engineering

Datum
Date

2001 - 06 - 01

Språk
Language

❑ Svenska/Swedish
☒ Engelska/English

❑ _____

Rapporttyp
Report: category

❑ Licentiatavhandling
☒ Examensarbete
❑ C-uppsats
❑ D-uppsats
❑ Övrig rapport

❑ _____

ISBN

ISRN

Serietitel och serienummer **ISSN**
Title of series, numbering

LiTH-ISY-EX-3150

URL för elektronisk version

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

Titel
Title

Model-based diagnosis using MathModelica

Författare
Author

Per Åkerlund

Sammanfattning
Abstract

In industrial processes, sudden faults must quickly be determined to avoid general failures. For this purpose model-based fault diagnosis can be used, which requires a consistent model of the real process. The powerful modelling tool MathModelica, based on the Modelica language, can be used to accomplish this. Systems for fault diagnosis could be both time-consuming and expensive if built manually. Instead, to automatically generate a fault diagnosis system, based on models built with MathModelica, would provide an efficient means of fault diagnosing. This thesis is about an algorithm which puts this into practice.

MathModelica has previously not been used for fault diagnosis, which makes this a pioneering work. Therefore, the algorithm is limited to consider only static electrical circuits and to diagnose constant voltage sources and linear resistors.

The algorithm takes a MathModelica model of a circuit and observations from the corresponding real system as input. Then a fault diagnosis system is generated and all possible diagnoses are obtained. The complexity of generating the diagnosis system grows very fast when the number of components is increased. Therefore, the capacity of the used computer puts limitations on the algorithm. An interesting extension would be to make the algorithm independent of the size of the circuit concerned, which could be done by considering subsets of the circuit.

Nyckelord
Keywords

Model-based fault diagnosis, Modelica, MathModelica, modelling, electrical circuits

Abstract

In industrial processes, sudden faults must quickly be determined to avoid general failures. For this purpose model-based fault diagnosis can be used, which requires a consistent model of the real process. The powerful modelling tool MathModelica, based on the Modelica language, can be used to accomplish this. Systems for fault diagnosis could be both time-consuming and expensive if built manually. Instead, to automatically generate a fault diagnosis system, based on models built with MathModelica, would provide an efficient means of fault diagnosing. This thesis is about an algorithm which puts this into practice.

MathModelica has previously not been used for fault diagnosis, which makes this a pioneering work. Therefore, the algorithm is limited to consider only static electrical circuits and to diagnose constant voltage sources and linear resistors.

The algorithm takes a MathModelica model of a circuit and observations from the corresponding real system as input. Then a fault diagnosis system is generated and all possible diagnoses are obtained. The complexity of generating the diagnosis system grows very fast when the number of components is increased. Therefore, the capacity of the used computer puts limitations on the algorithm. An interesting extension would be to make the algorithm independent of the size of the circuit concerned, which could be done by considering subsets of the circuit.

Contents

1

Introduction

This project is about model-based diagnosis of electrical circuits, modelled using the modelling and simulation tool *MathModelica*. An algorithm for diagnosing electrical circuits is described in this thesis.

This first chapter of the thesis gives a background to the work, a short description of MathModelica, an introduction to model-based diagnosis, limitations of the work and a reader's guide.

1.1 Background

When industrial systems are growing more complex, their needs for efficient maintenance and to quickly point out sudden faults are increasing. This kind of supervision can be provided by using *model-based diagnosis*, which requires a consistent model of the real system. Models are best simulated using computer programs and the language *Modelica* is developed for the purpose of simulating complete physical systems, where different areas of technology can interact.

The need of a consistent model using model-based diagnosis, requires an efficient modelling method. Further development of a certain real system leads to some structural changes in the model, and then it should be easy to re-model. The Modelica language offers this. Changes made in the real system also

implies that the corresponding diagnosis procedure needs to be updated, which would require much time and expenses if done manually. Instead, if there would be a way of updating the diagnosis procedure automatically, then generating systems for model-based diagnosis using Modelica models would be very efficient. An implementation of this kind of algorithm is possible in MathModelica, which is based on the Modelica language.

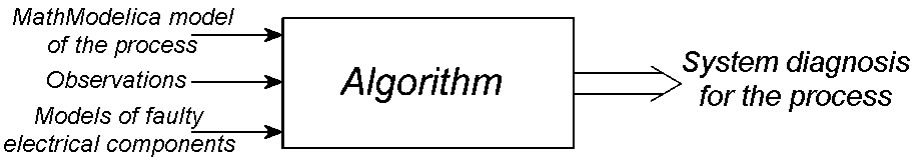

Figure 1.1 Generation of a system for diagnosing.

The field of model-based diagnosis is new for MathModelica and also for Modelica. Previously research within the area of this project has essentially been made in the field of *Artificial Intelligence* (AI). Former model-based diagnosis methods, close to the one presented in this thesis, are for example the *General Diagnostic Engine* (GDE) [2, 5] and *SOPHIE III* [1]. These methods are more developed than the algorithm described here, but they also lack the kind of powerful modeling functionality that MathModelica can offer. This makes it even more interesting to develop a new method for model-based diagnosis using MathModelica models.

The purpose of this project was to develop an algorithm that takes a model of an electrical circuit and from this generates a system for diagnosing a corresponding real circuit. Figure 1.1 shows how the algorithm as input takes a model of a process, models of faulty electrical components and observations (measurements) done on the corresponding real process. The result from the algorithm is the diagnosis for the process. All models are modelled using MathModelica.

1.2 MathModelica

At MathCore AB in Linköping, Sweden, the modelling and simulation tool MathModelica is developed. It is an environment for physical modelling based on Modelica, which is an

i) object-oriented,
ii) multi-domain and
iii) non-causal

language developed by the Modelica design group [9]. The language has a number of libraries [10] with components within different areas, e.g. electrical applications, mechanics and thermodynamics. The components are connected to each other like in a real system, which *i)* and *iii)* make possible. Different physical domains can interact in the same model, which is what *ii)* describes. The equations defining the behavior of the components are written explicitly, which gives *iii)*. All this makes it possible to accomplish a *full system simulation*. The DC-motor in figure 1.2 shows the possibility of connecting electrical applications to mechanical (*ii*). Every component is defined by equations explaining its physical behavior (*iii*), for example the resistor is modeled with Ohm's law, $U = R\,I$. This example model is found at MathCore's homepage [8].

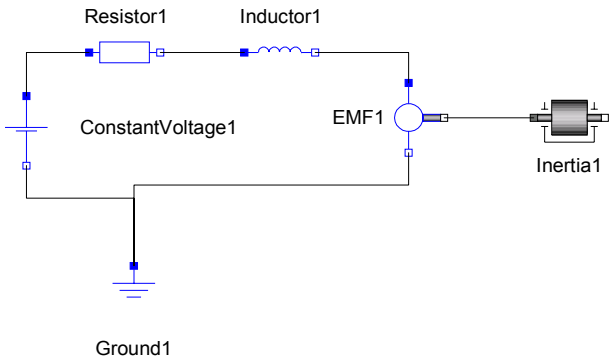

Figure 1.2 Graphical model of a DC-motor in MathModelica.

MathModelica uses its own syntax, but in close connection to Modelica's. The syntax of MathModelica is made to fit into Mathematica's standard, where the models are being implemented and presented in textual form. Figure 1.3 shows the MathModelica code of the DC-motor and in figure 1.4 the corresponding Modelica syntax is shown.

```
Model[DCMotor,
 Modelica.Electrical.Analog.Basic.Resistor Resistor1;
 Modelica.Electrical.Analog.Basic.Inductor Inductor1;
 Modelica.Electrical.Analog.Basic.Ground Ground1;
 Modelica.Electrical.Analog.Basic.EMF EMF1;
 Modelica.Electrical.Analog.Sources.ConstantVoltage
  ConstantVoltage1;
 Modelica.Mechanics.Rotational.Inertia Inertia1;
 Equation[
  Connect[ConstantVoltage1.p, Resistor1.p];
  Connect[Resistor1.n, Inductor1.p];
  Connect[Inductor1.n, EMF1.p];
  Connect[EMF1.n, Ground1.p];
  Connect[ConstantVoltage1.n, Ground1.p];
  Connect[EMF1.flange_b, Inertia1.flange_a]
 ]
]
```

Figure 1.3 Textual model of a DC-motor in MathModelica syntax.

```
model DCMotor
Modelica.Electrical.Analog.Basic.Resistor Resistor1;
Modelica.Electrical.Analog.Basic.Inductor Inductor1;
Modelica.Electrical.Analog.Basic.Ground Ground1;
Modelica.Electrical.Analog.Basic.EMF EMF1;
Modelica.Electrical.Analog.Sources.ConstantVoltage
  ConstantVoltage1;
Modelica.Mechanics.Rotational.Inertia Inertia1;
equation
connect (ConstantVoltage1.p, Resistor1.p);
connect (Resistor1.n, Inductor1.p);
connect (Inductor1.n, EMF1.p);
connect (EMF1.n, Ground1.p);
connect (ConstantVoltage1.n, Ground1.p);
connect (EMF1.flange_b, Inertia1.flange_a);
end DCMotor;
```

Figure 1.4 Textual model of a DC-motor in Modelica syntax.

On a higher level than the language, MathCore has developed a graphical *model editor.* The intension is essentially to work in the model editor where the visualization makes the models easy to interpret and simulation neat to handle

(the model in figure 1.2 is made here). From the model editor, all Modelica standard libraries are reachable [10], but also other components constructed in MathModelica. A new component is implemented using MathModelica or Modelica syntax in a Mathematica *notebook*. Such new components may be saved as MathModelica *packages* and after this they can be used in the model editor. Also the models in figure 1.3 and 1.4 are implemented and stored in notebooks. For more information about Mathematica see [13, 14].

MathModelica is using the Dymola Kernel, which is a part of the software Dymola developed by Dynasim [4]. When to simulate a model, the MathModelica code first is translated to Modelica syntax. Then it is sent to the Dymola kernel for translation to C, compilation and eventually simulation (i.e. solving an equation system).

In notebooks, both documentation and models with their simulation results can be integrated and stored (this thesis is a good example). Mathematica is also a powerful tool for manipulation of analytical and numerical expressions, which can be applied to models and their results. The essential part of this project has involved utilization of these facilities.

Version 2.0 of MathModelica is used in this project. To read more about this tool, see [8].

1.3 Model-based diagnosis

The diagnosis problem is to recognize if a correct built system is malfunctioning or not and if it is, localize and identify the fault. The result of this will generate one or several *diagnoses*. A diagnosis is a statement telling if the considered system is faulted or not and also possibly how it is faulted, if it is. The recognition of the system is made by possessing experience of the non-faulted system. In model-based diagnosis, the experience is represented as a model of the real system, which makes it very important for the model to be consistent with reality. To diagnose such a system, *observations* (i.e. measured values) from it are required so that these values can be compared with those generated by the corresponding model. A discrepancy between the two tells that there is one fault (or more) present.

1.3.1 About faults

The first step of the diagnosis procedure is to find out if a fault is present - *fault detection*. A next step is to determine what part or parts of the system that is faulted - *fault isolation*. To go further, there might be interesting to check the size and the time-dependence of the fault - *fault identification*.

The different fault related states that a component may be in is referred to as the *behavioral modes*, which includes one or more *fault modes* together with the *no-fault mode*. Each behavioral mode of a component is said to be defined by a *behavioral model,* i.e. how the component will function when it is in the no-fault mode or the different fault modes, respectively. A faulty (or non-faulty) system consisting of some components can then be diagnosed and said to be in a certain state, dependent on the present behavioral modes of its components. Here is an example to describe this.

Example 1.1

If a system S consists of two components A and B, each with the possible fault F, then the behavioral modes they can take (one at a time per component) is described by

$$A \in \{NF, F\},\ B \in \{NF, F\}$$

where NF stands for no-fault mode. When just *single faults* are assumed, the set of all possible diagnoses for the system can be written as

$$S \in \{NF, F(A), F(B)\}$$

If for a certain moment a fault in component A is detected, then the *single diagnosis* for S is $F(A)$.

The extension of example 1.1 is to add more fault modes F_i to the components and also assume multiple faults to occur, i.e. both A and B may be faulty at the same time. Example 1.2 shows this.

Example 1.2

If we assume *multiple faults* the set of possible diagnoses from example 1.1 would be extended to

$$S \in \{NF,\ F(A),\ F(B),\ F(A) \wedge F(B)\}.$$

The added diagnosis represents a *double fault,* i.e. both A and B are faulted. Furthermore, if we suppose one of the components to have more behavioral modes, for example

$$A \in \{NF,\ F_1,\ F_2\}$$

then two more possible diagnoses would be added the diagnosis set:

$$S \in \{NF,\ F_1(A),\ F_2(A),\ F(B),\ F_1(A) \wedge F(B),\ F_2(A) \wedge F(B)\}.$$

Note the difference between a *single fault* and a *single diagnosis*. A single diagnosis may be either a single fault or a multiple fault.

1.3.2 Diagnosis in two different areas

Within the area of *automatic control,* models are based on different mathematical equations (differential equations). For diagnosing such systems the signals of sensors and actuators are observed. To the equations describing a certain system, fault parameters are added for recognizing where faults are arising.

The other theoretical field where a lot of diagnosis research is carried out is artificial intelligence (AI). Here the diagnostic problem is more directed to the functioning or non-functioning of each component in the system. Compared to the view in control, this is a more logical way of seeing the systems, mathematically speaking. The term behavioral mode described above is taken from the AI-field. The work in this thesis is close to this area of diagnosis and therefore already developed AI methods has been studied in chapter 2.

For more readings in these two areas, see [12, 11, 6].

1.4 Limitations

Since neither Modelica nor MathModelica has been applied the diagnosis field before starting this project, the limitation was set to look at simple analog electrical circuits. The circuits have consisted only of the two linear, static components *resistor* and *constant voltage source* in varying combinations. A ground must always be connected to a circuit as a reference, but this component together with the wires are always assumed to be non-faulted.

The nature of the components makes all simulated values constant over time and for that reason, all measurements are also assumed to be time-invariant. Therefore, when making the diagnoses, just a single value at a certain point of time is considered. The algorithm is therefore not prepared to handle circuits with any dynamic characteristics, since this would require a sequence of measurements to be compared to a corresponding interval of simulation.

The goal when diagnosing a circuit was to look at *all possible diagnoses* for certain observations. In practice this means that a single diagnosis seldom is obtained from the diagnosis system.

1.5 Reader's guide

This report first gives an introduction to the work as well as a brief outline to the simulation environment MathModelica and to model-based diagnosis (this chapter). Chapter 2 describes two diagnosis algorithms developed within the field of AI.

In chapter 3, the procedure of generating and using the diagnosis system is presented, i.e. the implemented algorithm is described in words. Then follows chapter 4, which motivates the choices made when implementing the algorithm. Chapter 5 gives suggestions for further work, for example an interesting extension of the functionality of the algorithm. Last in the report, in chapter 6, a summary of the project results are presented.

1.6 Project partners

This work is a master thesis and was carried out in co-operation with Vehicular systems at Department of Electrical Engineering (ISY), Linköping University, and MathCore AB, Sweden, during spring semester 2001. Supervisors have been Mats Jirstrand at MathCore and Mattias Nyberg at ISY.

2

GDE and SOPHIE III

In this chapter, former research within the area of AI diagnosis is shortly introduced. Two methods, *General Diagnostic Engine* (GDE) and *SOPHIE III*, are illustrated by an example. The example is copied from a paper about the latter method by de Kleer and Brown [1] and GDE can be read about in [2, 5]. Both methods (algorithms) are described very shortly here and some details about them are intentionally left out. The purpose of this chapter is to give a brief look into what has been done before in the area of this thesis.

Figure 2.1 shows one small part of a bigger circuit, driven by some constant voltage. (The example is reduced compared to the one in the paper.) There are three resistors R1, R2 and R3 and two zener diodes D1 and D2. The resistance of each resistor is written inside the components and the breakdown voltages of the diodes are written beside them. Relevant nodes (points with equivalent potentials) are named N1 - N5. To specify certain voltage drops and currents through components some more parameters are defined as in table 2.1.

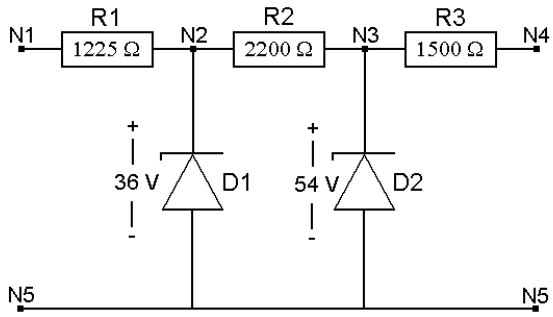

Figure 2.1 Fragment of a circuit.

Table 2.1 *Definition of parameters*

New parameter	***Between nodes / Through component***
U1	*N1, N5*
U2	*N2, N5*
U3	*N1, N2*
U4	*N2, N3*
U5	*N3, N5*
U6	*N1, N3*
I1	*R1*
I2	*D1*
I3	*R2*
I4	*D2*

2.1 SOPHIE III

SOPHIE III is a computer program, developed in the 1970's, for diagnosing electrical circuits. The system was primary used in a laboratory setting and to make it work with satisfaction, some presuppositions had to be made. For example, just *single faults* are assumed, *faults only occur in components* themselves (not in wires), the source is a *constant voltage,* and all components have the *same probability to fail*. SOPHIE III uses a *local propagator* called LOCAL to calculate expected values in the circuit, given one or more observations. SOPHIE III is not a general diagnosis system, since LOCAL only possesses knowledge of how *electrical* components work, i.e. what equations that characterize them.

Suppose that the measurements $U1 = 30\,\mathrm{V}$ and $U2 = 34\,\mathrm{V}$ are made in the example in figure 2.1. In SOPHIE III this would be stored as the two expressions

```
(V  (N1 N5)  (MEASUREMENT)  ())= 30
(V  (N2 N5)  (MEASUREMENT)  ())= 34
```

The first element in every expression explains the *type* of the value considered and the second tells the *location* of the value. Next element is the *reason* of the value, i.e. describes how LOCAL computed the value (above it was just a measurement). The last element contains the *assumptions* the value depends on, i.e. which components that must work correctly to propagate the value (which explains why it is empty for measurements).

After collecting observations LOCAL can now start propagating.

```
(V  (N1 N2)  (KVL N1 N2 N5)  ())  = 4
(I  R1       (RESISTORV R1)  (R1))= 0.003
(I  D1       (ZENERV D1)     (D1))= 0
```

In the first expression `KVL` stands for "Kirchoffs Voltage Law", which is here involving the three mentioned nodes. No assumptions are needed since the both measurements gives all necessary information. This first propagation makes it possible for LOCAL to calculate the second expression, namely the current through `R1` (because of the assumption `R1`). The third propagation gives that the current through D1 is zero, since the breakdown voltage is not reached according to the measurements.

LOCAL is, unlike the Modelica language, *casually dependent*, which implies that Ohm's law, $V = R\,I$, must exist in two forms. These two forms are denoted by

RESISTORV and RESISTORI. RESISTORV means that the voltage are known (similarly for ZENERV) while RESISTORI indicates that the electric current is known. (The first one represents the equation $I = V / R$ and the second $V = I R$.)

Some further propagations will be

```
(I  R2       (KCL N2)         (R1 D1))       = 0.003
(V  (N3 N2)  (RESISTORI R2)   (R2 R1 D1))    = 7.18
(V  (N3 N5)  (KVL N3 N2 N5)   (R2 R1 D1))    = 41.18
(V  (N3 N1)  (KVL N3 N2 N5)   (R2 R1 D1))    = 11.18
(I  D1       (ZENERV D1)      (D2 R2 R1 D1)) = 0
```

which are explained similarly. (KCL is "Kirchoffs Current Law".)

Now suppose the voltage drop between N1 and N3 is observed to 15 volts. This leads to a discrepancy as the propagated value is 11.18 volts. This means that one of the assumptions for this propagation is wrong, i.e. one of the components (R1 R2 D1) is faulted. This set is called a *nogood*. Using the notation from section 1.3, the possible diagnoses for this observation would then be

```
{F(R1), F(R2), F(D1)}
```

where F denotes any (unspecified) fault. Note that just single faults are possible. If the observation would be 11.18 volts, then all of (R1 R2 D1) are non-faulted for sure. This conclusion can be drawn since single faults are assumed and a faulty component is in SOPHIE III expected to appear faulty when it is.

Comparing other new observations with the propagations could yield more nogoods and since only one fault is assumed to be present at the same time, the intersection of all the nogoods will (may) reduce the number of components that are suspiciously faulted, i.e. components who are not members in all nogoods, cannot be faulted. For example, if we measure the electric current through R2 to 0.01 ampere, the obtained nogood will be (R1 D1). Then the possible components to be faulted are reduced to (R1 D1), which is the intersection of (R1 R2 D1) and (R1 D1).

By making further observations in this way, faults can eventually be isolated. This is how SOPHIE III is working. It proposes a next measurement using some algorithm to quickly localize the faulted component. This algorithm will not be described here. For more readings about SOPHIE III see [1].

2.2 GDE

As a comparison to SOPHIE III a later developed diagnosis algorithm, GDE, will here be described and also applied to the example in figure 2.1. GDE is maybe the most well known diagnosis algorithm within the AI area of diagnosis.

GDE, as well as SOPHIE III, uses local propagation but also a sort of database called *ATMS* (Assumption-based Truth Maintenance System) generally used within the area of AI. The important differences from SOPHIE III, is that the algorithm handles *multiple faults* and the local propagator of GDE is not made strictly for electrical circuits, but is more general. A similarity between the local propagators is however that they are both causal dependent.

Looking at the example, the observations from above would by the ATMS be stored as below (the text within round brackets is not stored).

⟦U1=30,{}⟧	(observation)
⟦U2=34,{}⟧	(observation)

where the first element is the measured quantity and its value. The other element contains *supporting environments*, which is a set of components which is the same set that for SOPHIE III is called assumptions. The information within brackets tells how the value was obtained. It does not illustrate any content in the ATMS.

Now the GDE algorithm can propagate the following and add to the database (similar to section 2.1):

⟦U3=4,{}⟧	(KVL)
⟦I1=0.003,{R1}⟧	(Ohm's law over R1)
⟦I2=0,{D1}⟧	(zener breakdown = 36 V)
⟦I3=0.003,{R1,D1}⟧	(KCL)
⟦U4=7.18,{R1,R2,D1}⟧	(Ohm's law over R2)
⟦U5=41.18,{R1,R2,D1}⟧	(KVL)
⟦U6=11.18,{R1,R2,D1}⟧	(KVL)
⟦I4=0,{D2,R1,R2,D1}⟧	(zener breakdown = 56 V)

Within the round brackets the current equation used in each propagation is showed and, as above, these are not stored by GDE.

Further observations will also be added to the database and with those additional two from section 2.1, this would give

```
⟦I3=0.01,{}⟧          (observation)
⟦U6=15,{}⟧            (observation)
```

With those added to the database GDE will discover a discrepancy between two pairs, as both `I3` and `U6` now exists with two different values. Therefore GDE generates the two nogoods

```
{R1,R2,D1}
{R1,D1}
```

which tells that the four possible *minimal diagnoses* are

```
F(R1)
F(D1)
F(R1)∧F(R2)
F(R2)∧F(D1)
```

where `F` is any unspecified fault. A minimal diagnosis is a diagnosis of which no subset is also an obtained diagnosis, is called a minimal diagnosis [3]. For example, the diagnosis `F(R1)∧F(R2)∧F(D1)` is not minimal, since it is a superset of `F(R1)∧F(R2)`. The diagnosis `F(R2)` alone is not a possible single fault, because the second nogood says that at least one of `R1` or `D1` is faulted but not `R2`.

GDE also suggests further measurements, but as for SOPHIE III, this is not described here. See [2, 5] for more about GDE.

2.3 A summary

Comparing SOPHIE III and GDE, tells that GDE, as the name says, is more general. SOPHIE III is developed to diagnose only electrical circuits, which GDE is not restricted to. Another difference between the two methods is that SOPHIE III just detects single faults while GDE also can find multiple faults. The local propagators for the two methods are both causal dependent. This means that the equation that describes the behavior of a certain component explicitly must be expressed in all the ways the variables can be extracted, i.e. for Ohm's law there are two ways, namely $I = V / R$ and $V = I R$.

The algorithm implemented in this project uses MathModelica for propagating values, which makes the modelling more simple and easier to grasp. An advantage of MathModelica is that it is non-causal.

Both GDE and SOPHIE III suggests further measurements, as mentioned. The system for diagnosing made in this project does not have this feature.

3

Generating a diagnosis system

The purpose of this work was to implement an algorithm that takes an existing MathModelica model of an electrical circuit and generates a *diagnosis system* for diagnosing a real circuit. Figure 3.1 shows a general outline of this diagnosis system. The algorithm was implemented as a script in Mathematica syntax [13, 14] with help from MathModelica commands [8].

This chapter describes how a diagnosis system is created, how the diagnosis components are built, the procedure of diagnosis calculation and finally how to use the algorithm.

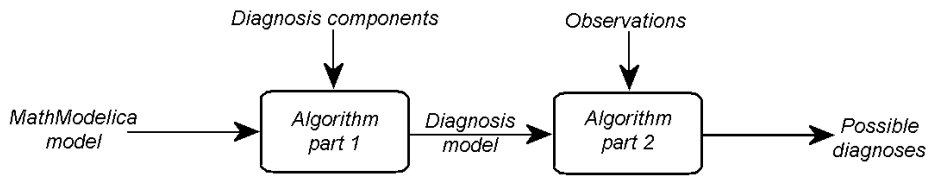

Figure 3.1 The diagnosis system.

The existing MathModelica model (here also referred to as the *original model* or

the *input model*) is assumed to consist of components from the Modelica standard library [10]. The basic idea when building the system in figure 3.1, is to change the existing Modelica standard components of the original model to corresponding *diagnosis components*. The diagnosis components are MathModelica models defining the different behavioral modes a certain component can be in. This makes the now changed original model able to exist in many shapes, depending on the different combinations of the behavioral modes of the diagnosis components. For each of these combinations a separate *behavioral circuit* is generated. All these disjunctive behavioral circuits are then assembled into a total model, the *diagnosis model,* which is simulated. This is the first part of the algorithm ("Algorithm part 1" in figure 1.1), which, referring to chapter 2, can be seen as the local propagator. The other part calculates the possible diagnoses from given observations of a corresponding real system ("Algorithm part 2"). The diagnosis model is the basis for making these diagnoses. The two algorithm parts together is referred to as the diagnosis system.

In Appendix A the complete code for this is presented.

To make it easier to follow, a very simple example circuit is visualizing the different phases. Figure 3.2 and 3.3 show this circuit as seen in the MathModelica model editor and the MathModelica code, respectively.

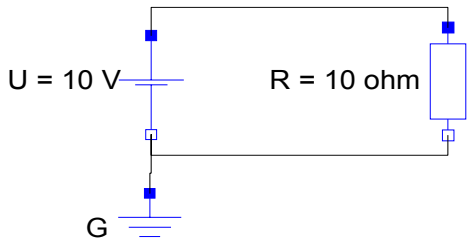

Figure 3.2 MathModelica model with a constant voltage source and one resistor.

```
Model[Circuit1,
 Modelica.Electrical.Analog.Basic.Resistor  R[{R == 10}];
 Modelica.Electrical.Analog.Sources.ConstantVoltage
  U[{V == 10}];
 Modelica.Electrical.Analog.Basic.Ground  G;
 Equation[
  Connect[U.p, R.p];
  Connect[R.n, G.p];
  Connect[G.p, U.n]
 ]
]
```

Figure 3.3 The model in figure 3.2 expressed in MathModelica code.

3.1 Diagnosis components

The diagnosis components are implemented as models in MathModelica, similar to the Modelica standard components in the standard library. The two components *resistor* and *constant voltage source* are both available in the standard library among other analog electrical components. Each diagnosis component is related to one corresponding standard component. In both of those components above, the superclass `oneport` (also from the standard library) is inherited and the diagnosis components are also built on this superclass. (The class `oneport` is a model representing any electrical component with two connect pins.)

The diagnosis components are saved as MathModelica packages and each one has its own package file, containing MathModelica models describing all behavioral modes, i.e. the behavioral models. Figure 3.4 illustrates the structural similarity and difference between one of the diagnosis components and its corresponding standard component. The package files of the diagnosis components also includes information on what component in the standard library it is related to, in figure 3.4 called *standard component relation*.

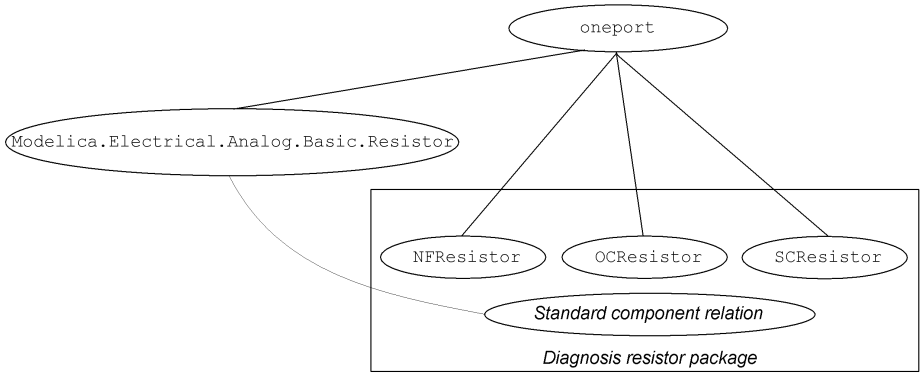

Figure 3.4 Tree showing how `oneport` *is inherited in the standard and diagnosis component, respectively, of a resistor.*

The model names `NFResistor` and so on in figure 3.4, are described in section 3.1.1. The long model name represents the standard component, see figure 3.3. See also section 4.1 for a further description regarding the choices when implementing the diagnosis components.

3.1.1 The component packages

The diagnosis resistor package (`DiagnosisResistor`) consists of three different behavioral models:

- no-fault (`NFResistor`)
- open-circuit (`OCResistor`)
- short-circuit (`SCResistor`)

The names within brackets, are the actual names of the implemented found in the code. Figure 3.5 and 3.6 shows the differences between the no-fault resistor and the open-circuit resistor. The whole diagnosis resistor package is found in Appendix B.2, where also an ideal resistor from the Modelica library is found (B.4).

```
Model[NFResistor,
 Extends[Modelica.Electrical.Analog.Interfaces.OnePort];
 Parameter Modelica.SIunits.Resistance R == 1;
 Equation[
  R i == v
 ]
]
```

Figure 3.5 No-fault resistor model.

```
Model[OCResistor,
 Extends[Modelica.Electrical.Analog.Interfaces.OnePort];
 Parameter Modelica.SIunits.Resistance R == 1;
 Modelica.SIunits.Resistance Ropen == 10^12;
 Equation[
  v == i Ropen
 ]
]
```

Figure 3.6 Open circuit resistor model.

Two behavioral models of the voltage source (`DiagnosisConstantVoltage`) are available:

- no-fault (`NFConstantVoltage`)
- empty battery (`EBConstantVoltage`)

Their definitions can be seen in Appendix B.3 and the standard component in B.4 for comparison.

3.1.2 Component combinations

It is obvious that for a great deal of components in a circuit, lots of fault combinations will be possible. The expression

$$\prod_{i=1}^{n} behaviorals_i \,, \tag{3.1}$$

where n is the total number of diagnosable components in the model and *behaviorals* is the number of behavioral modes for component *i,* represents the total number of combinations for a certain model. For example, for a circuit with three components connected, each having three behavioral modes, there are $3 \cdot 3 \cdot 3 = 27$ different combinations.

A feature of the algorithm makes it possible to decide how many of the combinations that should be considered. This is made by telling a maximum fault multiplicity to obtain in the resulting diagnoses (for example only double and single faults might be of interest).

3.1.3 Recognizing the components

The algorithm must be able to acquire knowledge of which diagnosis components that exist. This because it must be possible to relate the declarations of the components of the input model to the corresponding diagnosis components. Figure 3.8 shows which of the diagnosis components that are related to which of the standard components. To achieve this, a special MathModelica package is created, called `DiagnosisComponents` (Appendix B.1). Figure 3.7 visualizes from what instances the algorithm gets the information to accomplish what described by figure 3.8. The information consists of the names of the standard and diagnosis components.

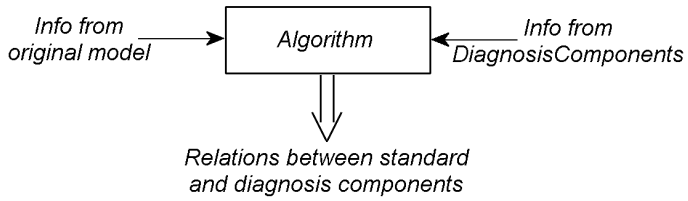

Figure 3.7 Recognition of components.

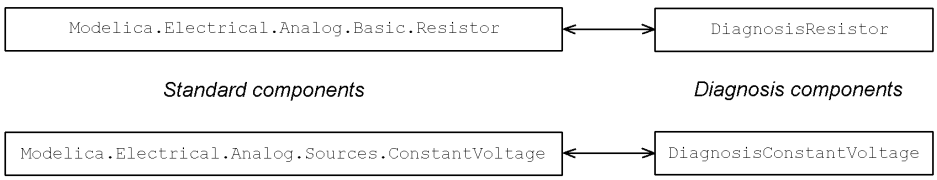

Figure 3.8 Relations between standard and diagnosis components.

The diagnosis component package contains a model `Electrical` that consists of a list with the complete names of all the standard components for which a diagnosis component package exists, i.e. the two in figure 3.8. When further electrical diagnosis components are implemented, the list will be added more elements. The intention of this structure is to associate the name with the actual area that the components are belonging to. For example, a future implemented diagnosis area could be `Mechanical`.

The algorithm assumes that all the diagnosis component packages are named as in this thesis. The packages for new components to be added in the future, shall therefore be named `Diagnosis`<*standard component name*>. Otherwise the recognition procedure in figure 3.7 will not work and the relations in figure 3.8 are then not found. The reason is that the algorithm uses the last part of the name of the standard component to identify what diagnosis component it is related to.

Furthermore, each behavioral model must be named in the same way as here, e.g. `NF`<*standard component name*>, but this is to standardize the look of the resulting diagnoses, i.e. how the result is printed on the screen (see section 3.4). Though, there may be more or less than two letters to describe the behavioral mode, e.g. `A`<*standard component name*> or `ABC`<*standard component name*>.

3.2 Generating and simulating the diagnosis model

From the original MathModelica model, the diagnosis model are being generated and then simulated. In figure 3.1 this is referred to as "Algorithm part 1". First in this procedure, the components of the original model are identified by their type and name. For those of which a corresponding diagnosis component exists the type and name are stored in pairs in a type list. For the example circuit in figure 3.2 and 3.3 this will yield the list

```
{{R, Modelica.Electrical.Analog.Basic.Resistor}
{U, Modelica.Electrical.Analog.Sources.ConstantVoltage}}
```

because the ground `G` does not have a corresponding diagnosis component. As seen in the example, a very long type name (declaration) is given. This is the complete Modelica standard library path to reach the components. To make the algorithm work, the declarations have to be expressed in this way. Otherwise they will not be treated as components that have corresponding diagnosis components, i.e. they will not be diagnosed. When building models in the

model editor this complete declaration path is automatically generated.

Now the standard components in the type list can be changed to their corresponding diagnosis components. Several combinations of the behavioral models are possible (see section 3.1), as each diagnosis component can be in at least two behavioral modes . The combinations in the example will be

```
{{R,NFResistor},{U,NFConstantVoltage}}
{{R,NFResistor},{U,EBConstantVoltage}}
{{R,OCResistor},{U,NFConstantVoltage}}
{{R,OCResistor},{U,EBConstantVoltage}}
{{R,SCResistor},{U,NFConstantVoltage}}
{{R,SCResistor},{U,EBConstantVoltage}}
```

which is the same number of combinations that expression (3.1) would yield.

For each of these combinations, a unique *behavioral circuit class* is defined, that declares one *behavioral circuit* each in the total diagnosis model. This is the diagnosis model for the example circuit:

```
Model[DiagModel,
 bmclass6 m6;
 bmclass5 m5;
 bmclass4 m4;
 bmclass3 m3;
 bmclass2 m2;
 bmclass1 m1
]
```

The behavioral circuit classes `bmclassi` declares the behavioral circuits `mi`. One of the behavioral circuit classes has the following definition:

```
Model[bmclass3,
 DiagnosisResistor.OCResistor R[{R == 10}];
 DiagnosisConstantVoltage.EBConstantVoltage U[{V == 10}];
 Modelica.Electrical.Analog.Basic.Ground G;
 Equation[
  Connect[U.p, R.p];
  Connect[R.n, G.p];
  Connect[G.p, U.n]
 ]
]
```

The standard components have here been changed into diagnosis components, each with a certain behavioral mode present.

Simulation of the diagnosis model results in a complete set of values. To be able to use these values to diagnose the real system, they need to be structured in some recoverable way. Therefore, a matrix with all the simulated values is generated, the *simulation matrix*, in which each row represents one of the behavioral circuits. The example circuit yields the simulation matrix

$$\begin{pmatrix} \{NF\} & \{current(R), voltage(R), current(U), voltage(U)\} \\ \{EB\_U\} & \{current(R), voltage(R), current(U), voltage(U)\} \\ \{OC\_R\} & \{current(R), voltage(R), current(U), voltage(U)\} \\ \{OC\_R, EB\_U\} & \{current(R), voltage(R), current(U), voltage(U)\} \\ \{SC\_R\} & \{current(R), voltage(R), current(U), voltage(U)\} \\ \{SC\_R, EB\_U\} & \{current(R), voltage(R), current(U), voltage(U)\} \end{pmatrix}$$

Note that `current(X)` states the electric current through the component `X` for the specific behavioral circuit that the current row represents. The notation `voltage(X)` similarly describes the voltage over the component `X`. The first elements in each row of the matrix above (`{NF}`,`{EB_U}` and so on), are explained in the section 3.3.

Ideal, the simulation matrix for the example circuit would look like

$$\begin{pmatrix} \{NF\} & \{1, 10, -1, 10\} \\ \{EB\_U\} & \{0, 0, 0, 0,\} \\ \{OC\_R\} & \{0, 10, 0, 10\} \\ \{OC\_R, EB\_U\} & \{0, 0, 0, 0\} \\ \{SC\_R\} & \{\infty, 0, -\infty, 0\} \\ \{SC\_R, EB\_U\} & \{0, 0, 0, 0\} \end{pmatrix}$$

The negative electrical current through `U`, is explained by the fact that the positive pin of the voltage source is connected to the positive pin of `R`. The definition of the components then say that `U` and `R` must have opposite direction of their currents.

This first part of the algorithm can be seen as a local propagator, according to the two methods described in chapter 2 and the code is shown in Appendix A.1.

3.3 Identifying the diagnoses

This section will explain how possible discrepancies between observations and the propagated values in the simulation matrix is detected.

The simulation matrix has as first element in every row an identifying element. This element tells what diagnosis that the following simulated values represents. In the second part of the algorithm, observations made from the real

system are matched against each row in the simulation matrix. For all the rows agreeing, the current identifying element (diagnosis) is returned and forms a resulting list of all possible diagnoses for the current observation case. The notations of the faults differs a little from how they were written in section 1.3 and chapter 2. In the diagnosis result, that the algorithm returns, the faults are denoted as `F_A`, instead of `F(A)` (`F` is the fault and `A` is the component).

Simulated values and observations are most likely not exactly equal, even if the model is built to reflect the real system (or a faulty real system). An observation is therefore said to match a simulated value within a *test interval* of five percent of the observation. When the simulation has returned a zero-value (or close to zero), an absolute test interval of [-0.01, 0.01] is used.

To illustrate how the possible diagnoses are obtained and presented on the screen, we go back to the simple example circuit in figure 3.2 and 3.3. Suppose that the resistor is short-circuit, the voltage source is non-faulted and the only observation is the voltage drop over the resistor, which is measured to 0 volts (or close to 0). If the supposition about the resistor was not known, perhaps the most obvious diagnosis would be `{EB_U}`. However, the voltage source is assumed to have an *inner resistance* (see chapter 4 for more about this), so `{SC_R}` would also be a possible diagnosis. The above observation also yields two more diagnoses and the total list of possible diagnoses would be

```
{{SC_R},{EB_U},{OC_R,EB_U},{SC_R,EB_U}}
```

As seen, the last two are double faults. This is explained by the fact that if the voltage source is empty, then it would be impossible to say anything about the state of the resistor, i.e. it could be in any of its behavioral modes.

The possible diagnoses above are visualizing exactly how this second part of the algorithm presents its result on the screen. Every diagnosis is assembled within one pair of curly brackets. The algorithm also have a feature of presenting just the minimal diagnoses (see section 2.2). For the case above, the minimal diagnoses would be

```
{{SC_R},{EB_U}}
```

since both of the double faults includes at least one of these single faults.
This part of the algorithm script is presented in Appendix A.2.

Note:
1. The test intervals are easily changed in the underlying function `TestValues[.]` in `GetFaultMode[.]`. See Appendix A.2.
2. The faults are denoted as e.g. `EB__U`. The sign "__" is used instead of "_", because the last one is a reserved character in Mathematica.

3.4 Using the diagnosis system

The two parts of the algorithm, described in the former sections, consists of a number of functions, which are all stored in one Mathematica notebook (see Appendix A). Before diagnosing, all these functions must be known by Mathematica, which is accomplished by evaluating them. Both algorithm parts have a *top function* each, which are described here.

To obtain the possible diagnoses for a certain system with respect to the observations, the two top functions are in turn evaluated. The original MathModelica model from which a diagnosis system is obtained, must also be defined and evaluated.

To the simple example circuit from before, we now add two more resistors connected in parallel with the first one. Figure 3.9 shows the new circuit, named `Circuit3`. (This is the same example as first in chapter 1, figure 1.1.) The corresponding MathModelica code is found in Appendix C.

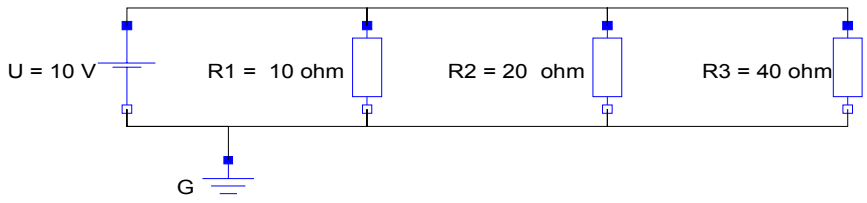

Figure 3.9 Circuit with a constant voltage source and three resistors in parallel.

3.4.1 Making the simulation matrix

The diagnosis system is generated by using the top function of the first part of the algorithm, which is

```
SimulateFaultModels[maxFaultSize,myModel]
```

where `myModel` is an existing model. The parameter `maxFaultSize` limits the diagnosis system to a certain fault-multiplicity (see also section 3.1). For example `maxFaultSize = 2` gives a system only treating double and single faults. If this parameter is omitted, all combinations of behavioral modes will be represented in the resulting diagnosis system.

Applying the above function on `Circuit3` (illustrated in figure 3.9) and limiting the diagnoses to maximum double and single faults, is done by typing

```
diagSystem32 = SimulateFaultModels[2, Circuit3];
```

The variable `diagSystem32` is used for storing the result of the function, i.e. the simulation matrix. Everything is now prepared for diagnosing the real system.

3.4.2 Obtaining the possible diagnoses

The observations must be arranged in a certain list structure; a structure that the diagnosis calculation can recognize. They shall be given together with their quantity on the form

```
{{quantity1, observation1}, {quantity2, observation2}, ... }
```

Quantities are stated as `R.v` for the voltage [V] over a component named `R` and `R.i` for the current [A] through the same component. It is important to call the component in question by its real name, i.e. the same as in the definition of original model. Measurements for every component and quantity must not be given, nor in a particular order.

For the new example circuit, a list of observations then could look like this:

```
observationList1 = {{R2.v, 10}, {R3.i, 0.25}, {R1.i, 1}};
```

With these values measured, we see that the circuit may be intact, but it also may not. Let's see what the diagnosis computation says about this fuzzy prediction.

To calculate the diagnoses, the following statement is used. It is the top function of the second part of the algorithm.

```
Diagnosis[myDiagnosisSelection,myObservations,myDiagnosisMatrix]
```

Here `myObservations` must be given as the list of observations mentioned above and `myDiagnosisMatrix` must be created by **`SimulateFaultModels`**. The parameter `myDiagnosisSelection` can be given one of the values `{All, Min}`, which gives, respectively, all the possible and the minimal diagnoses. Default value is `All` and is set if this parameter is left out.

After evaluating **`SimulateFaultModels`** once, the diagnosis computation can be done multiple times for different observations. However, only if changes are made in the original model (for example in figure 3.9) or if the fault multiplicity is changed, then a new simulation matrix have to be built.

For checking the above prediction of the observations in `observationList1` we evaluate

```
Diagnosis[All, observationList1, diagSystem32]
```

```
{{OC__R2}, {NF}}
```

This result says that either there is no fault in the circuit or the resistor `R2` might be open, i.e. the prediction from above was pointing in the right direction. Open circuit is diagnosed for `R2` since only the voltage is measured over this component, which does not give enough information to tell if it is broken or not; the voltage drop is still 10 V between the measuring points. Instead, if the current through `R2` would be observed to 0.5 A, then only the diagnosis `{NF}` would be made.

3.4.3 Some more examples

In the following diagnosis calculation for the above example (`Circuit3` in figure 3.9) an empty result is obtained, though the only observation that has changed is the current through `R1` which is now 0.9 A.

```
observationList2 = {{R2.v, 10}, {R3.i, 0.25}, {R1.i, 0.9}};

Diagnosis[All, observationList2]
```

```
{}
```

Even if the observation is just ten percent lower and could originate from such as tolerances in the resistor and accuracies in the measuring equipment, no

diagnosis can be calculated. This is depending on how the ranges in the test interval are set (see section 3.3). However, if this interval is set too wide, they will increase the risk of getting unrealistic or redundant diagnoses.

One last example of an observation list applied in the example in figure 3.9 is given to show the difference between `All` and `Min` when calculating the diagnoses. Let's say we measured all zero values:

```
observationList3 = {{R2.v, 0}, {R3.i, 0}, {R1.i, 0}};

Diagnosis[All, observationList3, diagSystem32]
```

```
{{SC__R1, EB__U}, {OC__R1, SC__R2}, {OC__R1, EB__U},
 {SC__R2, OC__R3}, {SC__R2, EB__U}, {SC__R2},
 {OC__R2, EB__U}, {SC__R3, EB__U}, {OC__R3, EB__U}, {EB__U}}
```

This results in a large number of possible diagnoses, mostly with double faults. Now if we want to look at the minimal diagnoses, the result obtained will be as follows.

```
Diagnosis[Min, observationList3, diagSystem32]
```

```
{{EB__U}, {SC__R2}}
```

All double faults are now gone, which is because every one of them includes either one of `{EB__U}` and `{SC__R2}` or both.

3.5 Using other components in the algorithm

Note that, even if the limitation here is set to diagnose only the static components resistor and constant voltage source, other components may be included in the input model. The algorithm is made to just diagnose the standard components for which corresponding diagnosis components exists. All the components in the input model that have not related diagnosis components, will remain the same during the diagnosis procedure. However, the limitation of looking at values in specific discrete moments cannot guarantee correctness of the diagnoses made when dynamic components are included. *But*, if such a component has a known time constant and if the component is stable, then it may be included in a circuit and the possibility of obtaining correct diagnoses will still be kept. Knowledge about when the circuit reaches stability, gives

information on how to adjust the simulation time of the diagnosis model and when to read the simulated values. Now the simulation time is set to two seconds, and the simulated values (which are stored in the simulation matrix) are read after two seconds. These time choices are arbitrary picked since the circuits in this thesis are assumed to have static characteristics. Adjustment of these values can be done in the functions `MakeTotalModel[.]` and `SimSalaBim[.]`. Both are found in the first part of the algorithm script, where the simulation matrix is created. If a circuit is constructed so it never will become stable, the now existing algorithm is not applicable.

4

Choices of implementation

In chapter 3, the diagnosis components were presented. The functionality of these components has appeared to be the key for making the diagnosis model (made by the first part of the algorithm) possible to simulate. In this chapter, the choices made when implementing the diagnosis components are explained. Problems occurring when using ideal diagnosis components are also discussed.

4.1 The structure of the diagnosis components

Mainly there are two ways of implementing the diagnosis components, either

i) as one model with a parameter controlling the behavioral mode or
ii) as one model per behavioral mode.

In this thesis the latter case (*ii*) is chosen (see Appendix B.2 and B.3), both because limitations of MathModelica and for practical reasons. The first alternative (*i*) may seem more natural to choose [7], since one standard component then just would have one corresponding diagnosis component. Written in MathModelica code this could for a resistor look like in figure 4.1. However, this is not possible, because MathModelica does not support defining enumerated types, as presumed in figure 4.1 by trying to use `MyBMType` to declare the variable `BehavioralMode`.

```
Model[DiagnosisResistor,
 Extends[Modelica.Electrical.Analog.Interfaces.OnePort];
 Parameter  Modelica.SIunits.Resistance  R == 1;
 Modelica.SIunits.Resistance  Ropen == 10^12;
 Modelica.SIunits.Resistance  Rshort == 10^-2;
 Parameter MyBMType BehavioralMode == NF;
 Equation[
  If [BehavioralMode === NF, R * i == v,
   If[BehavioralMode === OC, i == 0,
    If [BehavioralMode === SC, v == i * Rshort]]]
 ]
]
```

Figure 4.1 MathModelica code for a diagnosis resistor implemented as alternative i).

Instead, if `MyBMType` would be changed to an integer type and `BehavioralMode` for example could have the values {0, 1, 2}, then this diagnosis resistor would be functional. But then it would be impossible for the diagnosis algorithm to know which behavioral modes a certain component can take. For example, it does not know that a zero is representing the no-fault mode. This kind of knowledge of the diagnosis components is necessary for the algorithm to acquire, otherwise it would not be general enough. By choosing *ii)*, the names of each model can be associated with the current behavioral mode. The lack of being able to define enumerated types in MathModelica is therefore the strongest reason to implement as alternative *ii)*.

According to *ii)*, each behavioral mode should be represented as one MathModelica model, a *behavioral model*. They are named as, for example `NFResistor` and `SCResistor`, which clearly tells the corresponding behavioral mode and therefore easy can be detected by the algorithm. This also makes it easier to explicitly read the definitions of an arbitrary behavioral circuit, that the first part of the algorithm has generated (this can be done by following the example in Appendix D). To gather the behavioral models in a logical way, they are saved as MathModelica packages as described earlier (see section 3.1 and Appendix B).

4.2 Choice of behavioral models

The two types of faulted resistors (`OCResistor` and `SCResistor`) are the two most extreme thinkable faults. If a resistor breaks, for example caused by a too strong current, it most probable gets either short-circuit or open-circuit. Sometimes it may be more likely that an a certain resistor has the wrong resistance, but this type of faults are not treated here. Similar, the voltage source is said to be faulted only when it is totally empty.

4.3 Avoiding over- and underdetermined models

If MathModelica should be able to simulate a certain model, then that model has to generate a solvable system of equations. Therefore, this equation system must be neither overdetermined, underdetermined nor have a singularity. When creating behavioral circuits using ideal components, these kinds of equation systems appear in a number of situations.

Figure 4.2 shows an example of a circuit with a constant voltage source and two short-circuit resistors. Both the voltage source and the short circuit of the resistors are assumed ideal. (The resistors corresponds to the component `Short`, found in the Modelica standard library.) This model is not solvable in MathModelica, since it yields an equation system with a parameter solution, i.e. infinite number of solutions. The physical explanation to this, is that there is no information about the current flow through the resistors *R1* and *R2*, as no resistance is present, i.e. the electric current can be shared in all possible ways between the resistors. There can in fact be any circulating current in the loop consisting of *R1* and *R2*.

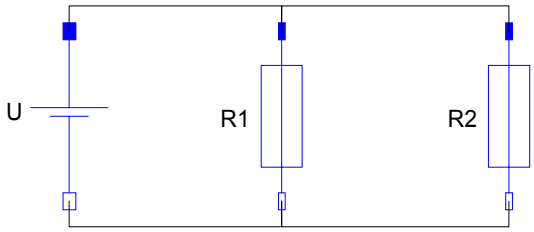

Figure 4.2 Circuit with a constant voltage source and two resistors in parallel. Both R1 and R2 are assumed short-circuit.

One solution to this problem is to use *non-ideal* components. The behavioral models defining open- and short-circuit resistors are each improved by a *open-circuit resistance* and a *short-circuit resistance*, respectively. The constant voltage source has an *inner resistance* added to it. The inner resistance is justified by looking at a real battery where this is always existing. For the diagnosis resistors, the two resistances are justified by the fact that physically there must still remain some resistance, in the first case very high and in the other very low. How this is put into the definitions of the two diagnosis components is shown in Appendix B.2 and B.3.

4.4 Ideal versus non-ideal components

The Modelica standard components considered in this thesis are ideal, but to make it possible to obtain a functioning diagnosis system, the diagnosis components had to be made non-ideal, which is explained in section 4.3.

The `NFResistor` model has exactly the same structure as the corresponding Modelica standard resistor. It may seem unnecessary to make another model with the same functioning, but it is in line with the reasoning in section 4.1, regarding the identifying of behavioral modes, together with keeping the structure of the diagnosis component packages consequent. The ideal components `Short` and `Idle` are already existing components from the standard library. They could seem to be a substitute for `OCResistor` and `SCResistor`, but from section 4.2 it is now known that non-ideal components is the only way to simulate faulty circuits.

Note: The only circuit components assumed ideal are the *wires*. To make also these non-ideal, a special wire resistor have to be implemented and put into the

circuit between every ordinary component terminal. This would make it possible to diagnose the wires, but also make the script much more complicated.

5

Future extensions of the diagnosis system

The algorithm developed in this project, takes an arbitrary electrical circuit and makes a diagnosis system for diagnosing resistors and constant voltage sources included in the circuit. Every of these two components in the circuit is considered in every diagnosis calculation. For circuits with a large number of these components, this will cause the first part of the algorithm (simulation of the diagnosis model and creation of the simulation matrix, see section 3.2) to grow complex and therefore require too much computer time. Also, if only a few observations are made on such a circuit, lots of possible diagnoses will be obtained and many of them could be redundant.

A suggestion of a general extension of the algorithm, regarding the fast growing of complexity, is discussed in this chapter. Some other proposals for widening the functionality of the diagnosis system are also presented.

5.1 Considering subsets of circuits

With the now existing algorithm it may be possible to consider smaller parts of a circuit, *but* then the user must explicitly build these circuit subsets and use them as the input model to the diagnosis system. This methodology would be

quite time-consuming (mostly for the user) every time the user wants to change the measuring points.

Instead, if the circuit subsets were automatically generated, depending on the locations of the observations, and put into the algorithm, then the diagnosis system would be much more powerful.

To describe qualitatively how this could be done, the example from section 3.4 is considered (`Circuit3`). It is visualized here again in figure 5.1. This circuit is not unreasonable big or complex for the now existing algorithm, but it is an appropriate example here. Any proposal of how the subsets can be chosen will not be presented, but only how to diagnose an already picked circuit subset.

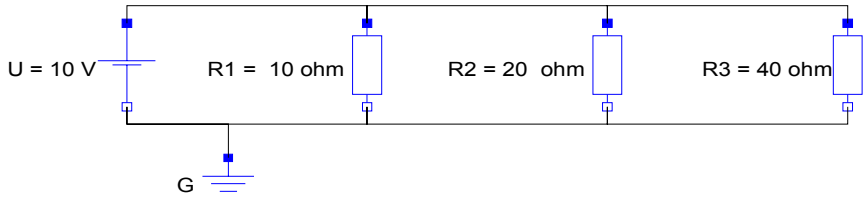

Figure 5.1 Circuit with a constant voltage source and three resistors in parallel.

Suppose that there are two different measurements made, *M1* and *M2*, where *M1* is the current through *R1* and *M2* is the current flowing from the positive pin of *U*. From these observations, the extension of the algorithm could for example pick a subset of the circuit that includes *U*, *R1* and *G*. This subset would look as in figure 5.2. The two resistors *R2* and *R3* are replaced by an *unknown part*, that defines the electric current to be equal in and out of it. (This type of component is possible to model in MathModelica.)

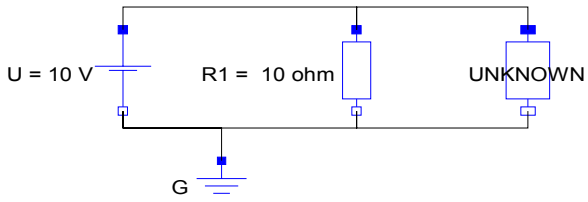

Figure 5.2 Subset of `Circuit3` *with two resistors replaced by an unknown part.*

By itself, the circuit subset in figure 5.2 yields an equation system that has infinite number of solutions, since there is no information on how much current is flowing through the unknown part (because of the unknown resulting resistance). *But*, we see that one small part of the circuit subset should be solvable - the part that includes the resistor *R1,* with knowledge of the voltage drop *U* over it.

Thus, if it was possible to simulate just those parts of the circuit subset that could yield any result, then the observations could be matched against that. For example, MathModelica could have a command named `SimulatePossibleParts` as an extension of the ordinary simulation command `Simulate`. For the observation *M1* from above, validation could be done, i.e. a diagnosis could be made. The second measurement *M2* though, will not bring enough information to tell anything about the present behavioral mode of *R1*. (However, *M2* can instead say something about *U*.)

A possibility when having the subset in figure 5.2 and the measurements *M1* and *M2*, is to add both observations to the equation system, which would then be overdetermined. This would be a way of telling if a certain subset is possible to validate or not. If the considered circuit subset has an overdetermined equation system, when including the observations, then at least one part of it can be validated. If this is put together with `SimulatePossibleParts` and we assume that we have some way of choosing all relevant subsets of the circuit, then the following subset algorithm could be usable:

1. Choose observations M to make

2. Pick a subset c_i of the circuit

3. If
i) c_i together with M yields an equation system that is overdetermined, then c_i should be saved.
ii) c_i together with M yields an equation system that is *not* overdetermined, then throw c_i.

4. If there are any c_i left, then go back to 2, else continue.

5. For all saved subsets c_i, use them as inputs to the diagnosis algorithm.

The input to the algorithm in the last step, should be a model with all the saved subsets c_i assembled.

To make it possible for this subset algorithm to work, some strategies have to be developed. First, the circuit subsets must be cleverly picked, regarding the observations. Second, a plan for deciding whether a subset c_i is overdetermined or not must exist. Third, a new simulation function, such like `SimulatePossibleParts`, must be implemented in MathModelica.

5.2 Other extensions

Besides the comprehensive extension described in section 5.1, there are some smaller additions, but not less necessary or important, that can widen the functionality of the algorithm.

One thing to add to the features of the algorithm in the future, is to change the matching of observations and simulated values in one single point of time into a signal-comparison over a time interval. If this could be done, it would be possible to implement much more electrical components, since most of them are either dynamic (e.g. capacitors) or non-linear (e.g. sinus voltage sources).

Another interesting extension is to improve the behavioral models of the diagnosis components. Now a resistor, for example, is either non-faulted, short-circuit or open-circuit. A widening of this could be that the diagnosis system would be able to determine if a resistor has the wrong resistance and to tell if

the resistance seems too high or too low. For example, if the real value should be 10 Ω but it is apparently just 5 Ω, then the result from the diagnosis system should tell that the resistance is too low. However, this may partly be possible to achieve if the validation thinking in section 5.1 can be used. Because if validation says that a certain resistor is neither correct, short- nor open-circuit, then it must hold that the resistor is faulted in an other way, for example it has the wrong resistance.

6

Summary

The developed diagnosis system takes a MathModelica model of an electrical circuit and observations from a corresponding real circuit and returns as result all the possible diagnoses for the specific case. The observations can be arbitrary picked by the user, i.e. the number and the location of the measurements to make.

The diagnosis system is able to diagnose constant voltage sources and resistors, which are both linear, static components. In a circuit, also other components can be included, but they will not be diagnosed. To diagnose circuits with a dynamic behavior, the now implemented algorithm has to be extended (it can be used in those cases the considered system always reaches a stable value after some time). A necessity for making the diagnosis system work, was to implement non-ideal diagnosis components, unlike the Modelica standard components that are modeled ideal. Ideal diagnosis components often leads to unsolvable equation systems.

Comparing the diagnosis system in this thesis with the two methods SOPHIE III and GDE, the local propagator in this thesis can be seen as the first part of the algorithm, i.e. the part generating and simulating the diagnosis model. A great advantage of using MathModelica when propagating is that it is non-causal, unlike these other two methods. SOPHIE III only treats single faults, which is not a limitation for neither this diagnosis system nor GDE.

Electrical circuits with a large number of components will in the implemented diagnosis system require too much computer time. It may be even too complex for an ordinary desktop computer to handle at all. Also, when just a small number of observations are made, according to the size of the circuit, components far from the measuring points seems unnecessary to consider in the diagnosis procedure. This would make it interesting to extend the algorithm, so that just interesting subsets of the circuit are considered regarding the observations. Such an extension requires MathModelica to be able to simulate all possible parts of a certain model, i.e. if the whole model cannot be simulated. Also some research to develop algorithms is needed, for example to decide how subsets of the circuit should be picked and how to treat those of the subsets that are actually not mathematically solvable.

Industrial systems changes are continuously made as a result of further development. To quickly update the model-based diagnosis systems of these industrial systems, the models of them must be easy to re-model. Also, a fast way of generating a new corresponding diagnosis system would be needed. Together with MathModelica, this is what the diagnosis algorithm in this thesis can provide - an automatic generation of a diagnosis system using models built by a powerful modeling tool.

References

[1] J. de Kleer and J. S. Brown, *Model-based diagnosis in SOPHIE III*, found in [8]

[2] J. de Kleer and B.C. Williams, *Diagnosing multiple faults*, Artificial Intelligense 32 (97-130), 1987; also found in [8]

[3] J. de Kleer, A. K. Mackworth and R. Reiter, *Characterizing diagnoses and systems*, Artificial Intelligence 56 (197-222), 1992

[4] Dynasim AB, www.dynasim.se

[5] K. D. Forbus and J. de Kleer, *Building problem solvers*, MIT Press, 1993

[6] W. Hanscher, L. Console and J. de Kleer, *Readings in Model-Based Diagnosis*, Morgan Kaufmann Publishers, 1992

[7] K. Lunde, *Object-Oriented Modeling in Model-Based Diagnosis*, Modelica 2000 Workshop Proceedings, The Modelica Association, October 2000

[8] MathCore AB, www.mathcore.com, Information about MathModelica, 2001

[9] Modelica Design Group, www.modelica.org, Information about Modelica language, December 2000

[10] Modelica Libraries, www.modelica.org/library/library.html, Documentation of the Modelica standard library, version 1.3.2 beta, July 2000

[11] M. Nyberg and E. Frisk, *Diagnosis and Supervision of Technical Processes*, Vehicular systems, Dep. of electrical engineering, Linköping University, 2000

[12] R.J. Patton, P.M. Frank and R.N. Clark, *Issues of Fault Diagnosis for Dynamic Systems*, Springer-Verlag, 2000

[13] S. Wolfram, *The Mathematica Book*, Third edition, Wolfram Media/ Cambridge University Press, 1996

[14] Wolfram Research Inc, *The Mathematica Book online*, www.wolfram.com, 2001

Appendix A

The algorithm of the diagnosis system

A.1 Creation and simulation of fault models

Generating and simulating the diagnosis model, given an original MathModelica model.

Starting MathModelica:

```
Needs["MathModelica`"]
```

The variables of the original model, with the corresponding types, are fetched and sorted into lists.

```
GetVarsAndTypes[mod_] :=
  Cases[
   GetType[mod],
   HoldPattern[
     Declaration[TYPE[tname_, ___],
      VariableComponent[name_, ___]]]
```

```
    :> List[name, tname],
    {0, ∞}
   ];


GetComponentPositions[types_] :=
Module[
 {diagAreas, expr, standardComponentNames, pos},

 diagAreas =
  Cases[
   GetDefinition[DiagnosisComponents],
   HoldPattern[Model[name_ = l_]] :> name,
   {0, ∞}];

 expr =
  Map[
   GetDefinition,
   MapThread[
    Member, {Table[DiagnosisComponents, {Length[diagAreas]}],
     diagAreas}
   ]
  ];

 standardComponentNames =
  Cases[expr, x_Member → x, {0, ∞}];

 Do[pos[i] = Position[types, standardComponentNames[[i]]],
  {i, Length[standardComponentNames]}];
 Sort[Fold[Join, pos[1],
   Table[pos[i], {i, 2, Length[standardComponentNames]}]]]
]


MakeTypeList[mod_] :=
Module[
 {varsAndTypes, posList, posListFirst},

 varsAndTypes = GetVarsAndTypes[mod];
 posList = GetComponentPositions[varsAndTypes];
 posListFirst = Table[First[Part[posList, i]],
   {i, Length[posList]}];

 Part[varsAndTypes, posListFirst]
```

```
]


MakeVarList[mod_] :=
 Module[
  {typeList},
  typeList = MakeTypeList[mod];
  Table[First[Part[typeList, i]], {i, Length[typeList]}]
 ]
```

Template for creating a new model with diagnosis components instead of the Modelica standard components.

```
ModelTemplate[mod_, name_, typeComb_] :=
 Module[
  {varList, TypeRule, TypeRuleList},

  varList = MakeVarList[mod];

  TypeRule[v_, t_] :=
   HoldPattern[
     Declaration[
      TYPE[tname_, rest1___],
      VariableComponent[v, rest2___]
     ]
    ]
    :>
    Declaration[
     TYPE[t, rest1],
     VariableComponent[v, rest2]
    ];

  TypeRuleList[v_] := Table[TypeRule[v[[i]], typeComb[[i]]],
    {i, Length[v]}];

  GetDefinition[mod, Format → MathModelicaFullForm] /.
   Join[{ToExpression[mod] → name}, TypeRuleList[varList]]
 ]
```

Creating all combinations of the diagnosis components associated with the standard components in the MathModelica model.

```
FaultTypeCombinations[size_, mod_] :=
 Module[
  {typeList, standardComponentList, FaultTypes,
   GetStdCompDefinition, GetStandardComponent, typePos,
   faultTypeList, NumOfNF, CheckNF, AllCombs},

  typeList = MakeTypeList[mod];

  standardComponentList =
   Flatten[Union[
     Cases[typeList, {v_, Member[x___, sc_]} :→ List[sc]]]
   ];

  DefinitionEval[comp_] :=
   GetDefinition[
    ToExpression["Diagnosis" <> ToString[comp]]
   ];

  Map[DefintionEval, standardComponentList];

  FaultTypes[comp_] :=
   DeleteCases[
    ListModelNames[
     ToExpression["Diagnosis" <> ToString[comp]]],
    Member[x___, comp]
   ];

  GetStdCompDefinition[x_, y_] := GetDefinition[Member[x, y]];

  StandardComponent[comp_] :=
   Flatten[Cases[
     GetStdCompDefinition[
      ToExpression["Diagnosis" <> ToString[comp]], comp],
     x_Member :→ List[x],
     {0, ∞}]
   ];

  typePos =
```

```
 typeList /.
  Table[
   {var_, First[StandardComponent[
       standardComponentList[[i]]]]} → {i},
   {i, Length[standardComponentList]}];

faultTypeList =
 ReplacePart[
  typeList,
  Table[FaultTypes[standardComponentList[[i]]],
   {i, Length[standardComponentList]}],
  Table[{i}, {i, Length[typeList]}],
  typePos
 ];

NumOfNF[comb_] :=
 Apply[
   Delete[comb, #] &,
   {Position[
     Map[
      StringPosition[#, "NF"] &,
      Map[ToString, comb]
     ], {}]
   }
  ] // Length;

CheckNF[comb__] :=
 If[
  NumOfNF[List[comb]] ≥ Length[typeList] - size,
  List[comb]
 ];

AllCombs[faultTypes__] :=
 DeleteCases[Flatten[Outer[CheckNF, faultTypes],
   Length[faultTypeList] - 1], Null];

Apply[AllCombs, faultTypeList]
]
```

Making model instancies for all behavioral combinations and then creates a total diagnosis model, by declaring a model for each behavioral model-class.

```
MakeAllModels[mod_, typeCombs_, names_] :=
 Module[
  {modelTable},
  modelTable = Table[mod, {Length[names]}];
  ReleaseHold[MapThread[ModelTemplate,
    {modelTable, names, typeCombs}]];
 ]
```

```
MakeTotalModel[mod_, obj_, faultCombs_] :=
 Module[
  {classNames, DeclareFunction},

  classNames = Table[ToExpression["bmclass" <> ToString[i]],
    {i, Length[faultCombs]}];

  MakeAllModels[mod, faultCombs, classNames];

  Model[DiagModel,
   Null;];

  Within[DiagModel];
  DeclareFunction[type_, var_] := Declare[type var];
  Do[DeclareFunction[classNames[[i]], obj[[i]]],
   {i, Length[classNames]}];
  EndWithin[];

  res = Simulate[DiagModel, {t, 0, 2}]
 ]
```

Making diagnoses out of all behavioral combinations.

```
FaultCombs2Diagnoses[combs_, mod_] :=
 Module[
  {varList, typeList, FMKeys, Glue, FMkey2Diagnosis,
   CompleteDiags},

  varList = MakeVarList[mod];
  typeList = MakeTypeList[mod] /. {v_, Member[x__, sc_]} → sc;

  FMkeys =
   ToExpression[
    StringReplace[
     ToString[combs /. Member[_, x_] → x],
     Table[ToString[typeList[[i]]] → "",
      {i, 1, Length[typeList]}]
    ]
   ];

  Glue[a_, b_] :=
   ToExpression[ToString[a] <> "␣" <> ToString[b]];

  FMkey2Diagnosis[key_] :=
   If[
    key === Table[NF, {Length[varList]}],
    {NF},
    MapThread[Glue, {key, varList}]
   ];

  CompleteDiags = Map[FMkey2Diagnosis, FMkeys];

  Fold[
   DeleteCases[#1, #2, {0, ∞}] &,
   CompleteDiags,
   Table[ToExpression["NF␣" <> ToString[varList[[i]]]],
    {i, Length[varList]}]
  ]
 ]
```

Assembling the diagnoses and simulated values into one total matrix, the

simulation matrix, which is used for diagnosing the model. The simulated values are concidered at the time 2 seconds.

```
SimSalaBim[faultSize_, inputModel_] :=
 Module[
  {faultTypeCombs, objNames, varList, typeList, quantities,
   AllDiags, simVarsValues, EvaluateMatrix, AddSimValue,
   simQuantities},

  faultTypeCombs = FaultTypeCombinations[faultSize,
    inputModel];
  objNames = Table[ToExpression["m" <> ToString[i]],
    {i, Length[faultTypeCombs]}];
  varList = MakeVarList[inputModel];
  quantities = {v, i};

  AllDiags = FaultCombs2Diagnoses[faultTypeCombs, inputModel];

  simVarsValues =
   Flatten /@ Outer[Member, objNames, varList, quantities];

  MakeTotalModel[inputModel, objNames, faultTypeCombs];

  EvaluateMatrix[matr_] :=
   Map[Replace[#, # :> #[2]] &, matr, {2}];

  AddSimValue[FM_, Values_] := {FM, Values};
  simQuantities = Flatten[Outer[Member, varList, quantities]];

  MapThread[
   AddSimValue,
   {Prepend[AllDiags, {QQ}],
    Prepend[EvaluateMatrix[simVarsValues], simQuantities]}
  ]
 ]
```

The top function in two forms with different input parameters.

```
SimulateFaultModels[faultSize_, inputModel_] :=
  SimSalaBim[faultSize, inputModel];
SimulateFaultModels[inputModel_] :=
  SimSalaBim[Length[MakeTypeList[inputModel]], inputModel];
```

A.2 Diagnosis test

Calculates all possible diagnoses. Observations are matched against the simulated values.

`QQRow` gives the position of the row containing the quantities, that are included in the simulation.

```
QQRow[simValueMatrix_] :=
  First[Flatten[Position[simValueMatrix, {QQ}]]];
```

Functions that separates quantities from measured values from the given observations list, and sort them into lists:

```
ObsValues[observationList_] :=
  Flatten[
   DeleteCases[observationList, x_Member, {0, ∞}]
  ];

ObsIndecies[observationList_, simValueMatrix_] :=
  Flatten[
   Cases[observationList, x_Member :> x, {0, ∞}]
    /.
    q_Member :>
     Position[
      Last[Part[simValueMatrix, QQRow[simValueMatrix]]], q]
  ];
```

Compares observations with one row in the simulation matrix and returns the corresponding diagnosis, if they are equal.

```
GetFaultMode[simMatrixRow_, observations_] :=
 Module[
  {TestValues, CheckValues},

  TestValues[simValue_, singleObs_] :=
   If[
    Abs[simValue] < 0.01,
    singleObs < 0.01,
    (* Fix value. Assumes observations of order ~1
      V, ~1 A *)
    If[
     simValue < 0,
     1.05 * singleObs ≤ simValue ≤ singleObs * 0.95,
     0.95 * singleObs ≤ simValue ≤ singleObs * 1.05
     (* Interval of 5 % of singleObs is considered *)
    ]
   ];

  CheckValues[simValues_] :=
   MapThread[TestValues, {simValues, observations}] ==
    Table[True, {Length[simValues]}];

  If[
   CheckValues[Last[simMatrixRow]],
   First[simMatrixRow]
  ]
 ]
```

Creating a test matrix, containing only the values that corresponds to the quantities measured.

```
TestMatrix[observationList_, simValueMatrix_] :=
  Module[
   {deleteQQ, GetSimFMList, GetSimValueList, PickTestValues},

   deleteQQ = Delete[simValueMatrix, QQRow[simValueMatrix]];
   GetSimFMList[] := Cases[deleteQQ, {x_, y_} :→ x, {1}];
   GetSimValueList[] := Cases[deleteQQ, {x_, y_} :→ y, {1}];
   PickTestValues[simValueRow_] :=
    Part[simValueRow, ObsIndecies[observationList,
      simValueMatrix]];

   MapThread[
    List,
    {GetSimFMList[],
     Map[PickTestValues, GetSimValueList[]]}
   ]
  ];
```

Calculating all possible diagnoses.

```
DiagnosisResult[observationInput_, simValueMatrix_] :=
 Module[
  {findDiagnosis, possibleDiagnoses},

  findDiagnosis =
   Distribute[
    GFM[TestMatrix[observationInput, simValueMatrix],
     {ObsValues[observationInput]}],
    List,
    GFM,
    List,
    GetFaultMode
   ];

  possibleDiagnoses = DeleteCases[findDiagnosis, Null]
 ]
```

Modifying the result when the user chooses just to see the minimal diagnosis.

```
SubSetQ[set_, elems_] :=
  Fold[
   And,
   True,
   Map[MemberQ[set, #] &, elems]
  ];


MinDiag[res_] :=
  Module[
   {min, GetSets, sets},
   min = {};
   GetSets[ls_] :=
    Module[
     {},
     sets = Select[ls, SubSetQ[#, Sort[ls][[1]]] &];
     min = Append[min, Sort[ls][[1]]];
     Complement[ls, sets]
    ];
   NestWhile[GetSets, res, ! Equal[{}, #] &, 1];
   min
  ];


DiagnosisChoice[res_, choice_] :=
 Module[
  {},
  Which[
   choice === Min,
   MinDiag[res],

   choice === All,
   res
  ]
 ]
```

The top function in two forms with different input parameters.

```
Diagnosis[choice_, observationInput_, simValueMatrix_] :=
 DiagnosisChoice[
  DiagnosisResult[observationInput, simValueMatrix],
  choice]

Diagnosis[observationInput_, simValueMatrix_] :=
  DiagnosisChoice[
   DiagnosisResult[observationInput, simValueMatrix],
   All]
```

Appendix B

Diagnosis components

Here the code of the diagnosis components are presented. The start and end statement of each package below is written in pure Modelica syntax. In section B.3 the code of the components from Modelica standard library is shown for comparison. They are written in Modelica code.

B.1 The DiagnosisComponents package

```
package DiagnosisComponents
```

```
Model[Electrical =
  {Modelica.Electrical.Analog.Basic.Resistor,
   Modelica.Electrical.Analog.Sources.ConstantVoltage}]
```

```
end DiagnosisComponents;
```

B.2 The DiagnosisResistor package

```
package DiagnosisResistor
```

```
Model[Resistor = Modelica.Electrical.Analog.Basic.Resistor]
```

```
Model[NFResistor,
 Extends[Modelica.Electrical.Analog.Interfaces.OnePort];
 Parameter Modelica.SIunits.Resistance R == 1;
 Equation[
  R i == v
 ]
]
```

```
Model[OCResistor,
 Extends[Modelica.Electrical.Analog.Interfaces.OnePort];
 Parameter Modelica.SIunits.Resistance R == 1;
 Modelica.SIunits.Resistance Ropen == 10^12;
 Equation[
  v == i Ropen
 ]
]
```

```
Model[SCResistor,
 Extends[Modelica.Electrical.Analog.Interfaces.OnePort];
 Parameter Modelica.SIunits.Resistance R == 1;
 Modelica.SIunits.Resistance Rshort == 1/10^3;
 Equation[
  v == i Rshort
 ]
]
```

```
end DiagnosisResistor;
```

B.3 The DiagnosisConstantVoltage package

```
package DiagnosisConstantVoltage
```

```
Model[ConstantVoltage =
  Modelica.Electrical.Analog.Sources.ConstantVoltage]
```

```
Model[NFConstantVoltage,
 Extends[Modelica.Electrical.Analog.Interfaces.OnePort];
 Parameter  Modelica.SIunits.Voltage  V == 1;
 Modelica.SIunits.Resistance Rinner == 1/10;
 Equation[
  v == V - -i Rinner
 ]
]
```

```
Model[EBConstantVoltage,
 Extends[Modelica.Electrical.Analog.Interfaces.OnePort];
 Parameter Modelica.SIunits.Voltage V == 1;
 Modelica.SIunits.Resistance Rinner == 1/10;
 Equation[
  v == i Rinner
 ]
]
```

```
end DiagnosisConstantVoltage;
```

B.4 Ideal components from Modelica library

```
model Resistor
extends Modelica.Electrical.Analog.Interfaces.OnePort;
parameter SIunits.Resistance R=1;
equation
R*i=v;
end Resistor;
```

```
model ConstantVoltage "Source for constant voltage"
   parameter Modelica.SIunits.Voltage V=1 "Value of constant
voltage";
   extends Modelica.Electrical.Analog.Interfaces.OnePort;

equation
   v = V;
end ConstantVoltage;
```

Appendix C

Example circuit

The MathModelica code for the example circuit `Circuit3`.

```
Model[Circuit3,
  Modelica.Electrical.Analog.Basic.Resistor R1[{R == 10}];
  Modelica.Electrical.Analog.Basic.Resistor R2[{R == 20}];
  Modelica.Electrical.Analog.Basic.Resistor R3[{R == 40}];
  Modelica.Electrical.Analog.Basic.Ground Ground;
  Modelica.Electrical.Analog.Sources.ConstantVoltage
   U[{V == 10}];
  Equation[
   Connect[R1.p, U.p];
   Connect[R2.p, R1.p];
   Connect[R3.p, R2.p];
   Connect[R3.n, R2.n];
   Connect[R1.n, R2.n];
   Connect[U.n, R1.n];
   Connect[Ground.p, U.n]
  ]
 ];
```

Appendix D

Example: Definition of a diagnosis model

This example shows how an original model is changed by the script. The circuit consists of three resistors in parallel connected to a constant voltage source.

```
Model[Circuit3,
 Modelica.Electrical.Analog.Basic.Resistor  R1[{R = 10}];
 Modelica.Electrical.Analog.Basic.Resistor  R2[{R = 20}];
 Modelica.Electrical.Analog.Basic.Resistor  R3[{R = 40}];
 Modelica.Electrical.Analog.Sources.ConstantVoltage
  U[{V == 10}];
 Modelica.Electrical.Analog.Basic.Ground  Ground;
 Equation[
  Connect[U.p, R1.p];
  Connect[R1.p, R2.p];
  Connect[R2.p, R3.p];
  Connect[R3.n, R2.n];
  Connect[R2.n, R1.n];
  Connect[R1.n, Ground.p];
  Connect[Ground.p, U.n]
 ]
]
```

Creating and simulating all combinations of behavioral circuits.

```
SimulateFaultModels[2, Circuit3];
```

Now all behavioral circuits are put together in one total model, the diagnosis model, but the single behavioral classes are still readable alone. These can be read by typing...

```
GetDefinition[bmclass7]
```

```
Hold[Model[bmclass7,
  DiagnosisResistor.OCResistor R1[{R == 10}];
  DiagnosisResistor.SCResistor R2[{R == 20}];
  DiagnosisResistor.NFResistor R3[{R == 40}];
  Modelica.Electrical.Analog.Basic.Ground Ground;
  DiagnosisConstantVoltage.NFConstantVoltage U[{V == 10}];
  Equation[
   Connect[R1.p, U.p];
   Connect[R2.p, R1.p];
   Connect[R3.p, R2.p];
   Connect[R3.n, R2.n];
   Connect[R1.n, R2.n];
   Connect[U.n, R1.n];
   Connect[Ground.p, U.n]
  ]
 ]]
```

...and yet we can see that the four components have new declarations. Now there are different diagnosis component types declaring the components!