MASTER

Operation Analyzer
a software approach for improving fault diagnosis at Philips Medical Systems

Kurvers, M.J.M.

Award date:
2007

TECHNISCHE UNIVERSITEIT EINDHOVEN
Department of Mathematics and Computer Science

Operation Analyzer

A software approach for improving fault diagnosis at Philips Medical Systems

By
M.J.M. Kurvers

Supervisors:

Michel Chaudron (TU/e)
Pieter Buts (Philips Medical Systems)

Eindhoven, October 2007

Preface

This thesis is submitted in partial fulfillment of the requirements for a Master's Degree in Computer Science and Engineering at the Technical University of Eindhoven. It contains a summary of the work that I've done from December 2006 to October 2007 at Philips Medical Systems in the area of fault diagnosis.

Philips Medical Systems produces very complex systems, used in medical care all over the world. Even though the products are of high quality, the components of these systems are subject to wear and aging, and can break down eventually. When a system breaks down, it is important that it gets repaired as quickly as possible. To repair the machine, the service engineer first needs to find out which component is broken, and this process is called fault diagnosis.

In this thesis we are investigating how to improve fault diagnosis at Philips Medical System, such that the downtime of a system is reduced and less non-faulty components are replaced.

I would like to thank everybody who has helped me during my Master's project. In particular I would like to thank my supervisor at Philips Medical Systems, Pieter Buts who contributed in a lot of ways to my project. I would also like to thank my supervisor at the Technical University of Eindhoven, Michel Chaudron. Finally I would like to thank all my colleagues at Philips Medical Systems for the nice time I had working there.

Index

List of figures

List of tables

Abbreviations

C/V	Cardio Vascular
CAN	Controller Area Network
CT	Computed Tomography
FIPS	Fault Isolation Procedures
FRU	Field Replaceable Unit
FSE	Field Service Engineer
KB	Knowledge Base
KBS	Knowledge Base System
MBD	Model Based Diagnosis
MR	Magnetic Resonance
NN	Neural Network
PMS	Philips Medical Systems
RBR	Rule-based Reasoning

Terms

Target (Software)	The system that is under diagnosis. In the case of this project the target is the Cardio/Vascular X-Ray system of PMS and the target software is its controlling software.
Diagnostic Method	The diagnostic method that we are designing and according to which the diagnosis process will be executed. This will also be referred to as the method.
Diagnostic Solution	The entire diagnostic solution that is needed to improve the fault diagnosis process at PMS. This includes the diagnostic method and all components of the final implementation, such as a diagnostic engine, a presentation layer, observation servers, observation points built-in in the target software, defined diagnostic rules and defined diagnostic models.
Diagnostic Engine	The engine that will perform the diagnosis. This engine needs behavioral and diagnosis information and derives a diagnosis from this information.
Diagnostic Model	A model that describes the correlation between observations that are done and the healthiness of components. The diagnostic engine uses models to diagnose the system. A diagnostic model is defined for one diagnostic module (and hence can be used to diagnose that module). A diagnostic model will also be referred to as model.
(Diagnostic) Model Rule	A diagnostic model consists of one or more (diagnostic) model rules. Each model rule describes the correlation between observation points and components, especially which impact the signaling of an observation point has on the healthiness of certain components. A diagnostic model rule is part of a model and will always be referred to as either diagnostic model rule or model rule so no confusion can arise with diagnostic rule. Whenever a “rule” is mentioned a diagnostic rule is meant and not a model rule.
Diagnostic Module	A defined subsystem of the Target on which diagnostic models are made to diagnose that subsystem. This will also be referred to as module.
Diagnostic Rule	A rule that describes a number of diagnostic models and the order in which they need to be checked. This will be referred to as rule if no confusion can arise with the (diagnostic) model rule.
Observation Point	One point in the target software that can be signaled when the control flow passes there. Observation points are used to gather observations (behavioral information) from the system and are used in diagnostic models.
Observation Server	An observation server will be built-in in the target software and is used by the diagnostic engine to retrieve information about observation points (and hence behavioral information) from the target.

1 Introduction

Machines nowadays are getting more and more complex. This is also true for systems in a medical environment. At the same time we are getting more dependable on these systems and therefore any downtime of the system should be kept to a minimum. Part of keeping the downtime to a minimum is being able to quickly diagnose what is broken in a system when it stops functioning. This process is called fault diagnosis and it will be the main topic of this thesis.

1.1 Fault Diagnosis

To discuss the topic of fault diagnosis, first the terminology must be unambiguously described. The master's project is carried out at Philips Medical Systems in the development group where software is developed for Cardio/Vascular X-ray systems. By *machine* or *system* we will refer to the complete system, meaning both hardware and software components.

With a *module* or *subsystem* we will refer to a part of the machine (either hardware, software or both) that has its own responsibility within the machine. We shall use *component* to refer to a part of a module or subsystem that works together with other parts in that module or subsystem. Philips Medical Systems has also defined the notion of *Field Replaceable Unit (FRU)*, which indicates a part of the system (at least one component, up to a complete module), which can be replaced as a whole.

By *error* we mean an observation that can be made of the machine behaving in a way that was not specified. A *fault* or *defect* is the malfunctioning of a hardware and/or software component. A fault could manifest itself as an error, but it could also be dormant (because the faulty component is not used, or its malfunctioning is masked by the behavior of other components). Lastly we shall use *diagnosis* or *fault diagnosis* to refer to the task of locating the fault that has led to an error. This fault then is the *root cause* of the error.

So by *fault diagnosis* we refer to the task of finding the fault that is the root cause of an error in the system under diagnosis.

In this master's project we will create a *diagnostic system*. This is a system that performs fault diagnosis. Unfortunately, diagnostic systems are rarely perfect. Sometimes it misses a fault and indicates that the component is working correctly, while it is actually broken. This is called a *false negative*. A diagnostic system might also indicate that something is broken, while it actually is working correctly. This is called a *false positive*.

In the context of this master project, fault diagnosis is used to find faults in the hardware. The software is assumed to be correct. Therefore we shall use the term fault diagnosis as the process of finding the faulty hardware component(s).

1.2 Philips Medical Systems

Philips Medical Systems is concerned with designing and building systems for the medical industry. One of these systems, as already mentioned in the previous paragraph, is the Cardio/Vascular X-Ray system. A picture of such a system is shown in Figure 1.1.

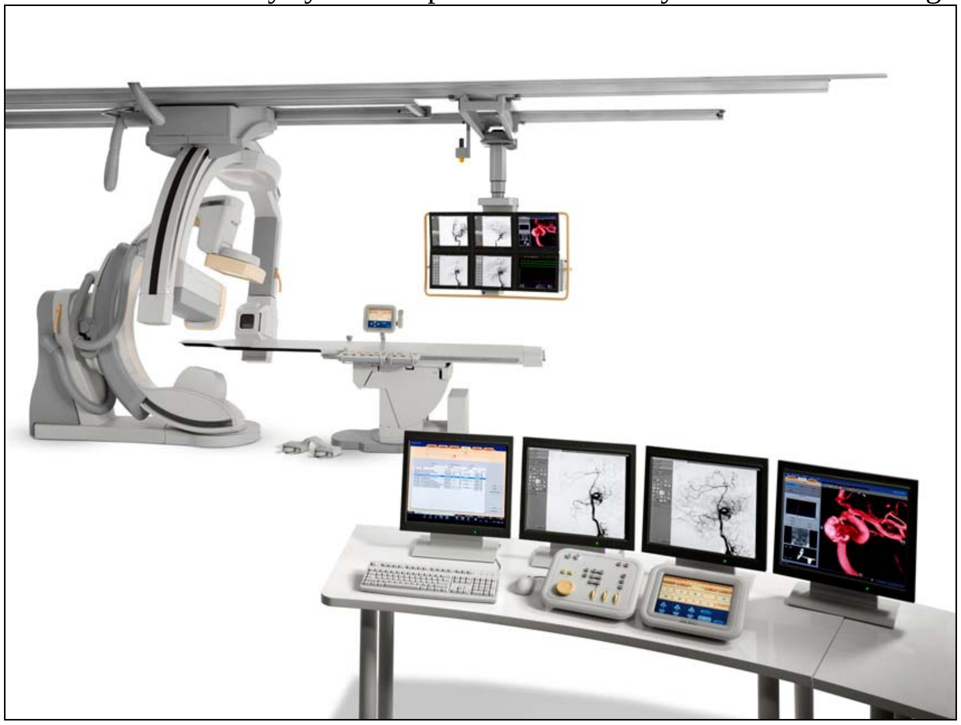


Figure 1.1 Cardio/Vascular X-Ray system

This system is used in hospitals to create diagnostic images of patients using X-rays. These images can for instance be used for detecting blood clots in someone's brain, but also to guide the placement of stents in someone's arteries to keep them open in case of artery blockage.

Apart from the hardware shown here, there are more parts such as power supplies, computers, X-ray generators, coolers and various control boards. These parts are placed in cabinets which are not shown in the picture above.

1.3 Fault Diagnosis at Philips Medical Systems

As with all systems, the Cardio/Vascular X-Ray system at Philips Medical Systems is continuously evolving. More functionality is added to the system which leads to an increase of the complexity of the hardware and the software. This increase of complexity has given rise to the problem that finding a fault in the system becomes increasingly difficult.

One of the aspects of this continuous evolvement is that there are many different versions of the Cardio/Vascular X-Ray system in hospitals. Besides differences in versions, there are also different configurations possible. These two factors lead to the fact that the system is hard to diagnose for the field service engineer who is sent to the hospital when a machine breaks down. In the current situation, it takes at least a few hours before a fault is located, and sometimes it takes up to several days or even weeks. Such a downtime is not acceptable, and therefore Philips Medical Systems has started a number of projects that have as goal to improve overall reliability. One of those projects that has as goal to decrease the downtime of the systems is the project that is described in this thesis. Its goal is to see whether it is possible to create a software solution that helps decrease the time it takes to diagnose a system after it has broken down and to accurately pinpoint the problem.

1.4 Outline

The project will be further described in chapter two of this thesis. That chapter will also give a clear description of the goal of this project. Chapter three gives an analysis of fault diagnosis at Philips Medical Systems and will point out some aspects that need improvement. In chapter four the research will be described and the conclusions from this research will be given. The method that is designed will be described in chapter five and the design of the prototype using this method is discussed in chapter six. Validation of this method will be discussed in chapter seven and finally chapter eight and nine give the conclusions and some recommendations for future research and improvements.

2 Project Description

This chapter describes the project in more detail.

2.1 Problem description

The Cardio/Vascular X-Ray system is a complex system consisting of several hardware modules operating together that are controlled by software. The sequence of events in the system is based on the design of the system and its configuration. E.g. operator controls send a signal to a processing unit that starts a movement, depending on several internal and external hardware conditions. For fault analysis it can be hard to find the location of the problem. It could be a fault in the operator control unit, a fault in the communication between the units, an internal state, a hardware/sensor fault, etc. The analysis is also complicated by the fact that there can be many configurations (combinations of hardware modules and software versions).

2.2 Goal

The goal of this project is to design a method that improves diagnosing a C/V X-Ray System from Philips Medical Systems. Improvement must lead to a quicker and more accurate diagnosis that eventually lowers the total time to repair the system and the number of non-broken components that are exchanged.

2.3 Scope

The scope of the project is limited in a number of ways. This project is meant as a feasibility study, in which a method has to be designed, a limited prototype has to be created and this prototype has to be verified to determine whether the method works.

By limited prototype we mean a prototype on a part of the functionality of the machine. Together with experts at Philips Medical Systems a part will be selected that is big enough to demonstrate the feasibility of the method, but small enough to be created in the limited time available.

The scope of the project will also be limited to finding hardware faults, since this is the goal of the project. Therefore, the software will be assumed correct.

3 Fault Diagnosis Analysis

To design a method for improving fault diagnosis, we first need to analyze what properties are important for that method. Therefore we need to analyze the assignment and examine the current situation of working. From this we can conclude if and how improvements can be achieved.

3.1 Analysis of assignment

Philips Medical Systems produces high quality systems incorporating strict safety measures to avoid harm to patients and medical personnel. However, these systems are made of moving and electric parts and are therefore subject to wear and electronic fluctuations. This means that eventually something, either mechanical or electrical, will break in the machine causing it to malfunction.

The main goal of the assignment is to find broken hardware components in a system that is malfunctioning. This means that software errors are not part of the project scope, although it would be valuable if the diagnostic system could indicate that all involved hardware appears to be functioning correctly and suggest that it might be an error related to the software.

The target audience for the final solution of this project is field service engineers (FSE) that are low to medium experienced. Since PMS is broadening its business to more and more countries over the world, there is also a rising need for field service engineers who are able to solve problems with these systems. The reason for this is that they only get a short training, and that they are trained to be able to provide service to not just CV machines, but also CT, MR and other PMS systems. The solution should therefore focus on the large group of field service engineers that is low to medium experienced and help them find the faulty component quicker.

3.2 Analysis of current situation

When a machine is no longer operating, a call is made to the service desk which decides whether it is necessary to send a field service engineer to the location or not. If a field service engineer is dispatched to the site, his task is to determine which component is broken in the machine. He is also responsible for replacing the broken component. This all should happen as quickly as possible, since any downtime of the machine means that time is lost in which patients could be diagnosed.

The task of finding the broken component or broken components in a system is called the diagnosis of the system. During this diagnosis, the field service engineer can use many tools. The FSE usually starts with an interview with the operator to get a clear picture of what the symptoms are. Then he will perform a visual check of the system, where he checks various leds that indicate the operational status of parts of the system. After that he will inspect the log files, hoping to see some log entries that might indicate where the problem lies. After these steps, the field service engineer has many options to try and find the cause of the problem. He might:

- Use an online knowledgebase to try if such an error has occurred earlier and if there is a given solution for it
- Perform a number of functional tests
- Lookup the status of the Power-on-self-test (POST) of various subsystems
- Use technical drawings
- Use Fault Isolation Procedures (FIPS)
- Use various mechanical tools to test parts of the system
- Use various external software tools to test functionality*

*This is only available to certain very experienced engineers since these tools might seriously damage the system if used incorrectly. Since our focus is on low to medium experienced field service engineers, we assume that they will not use these tools.

Most of the above methods are quite low level; they require that the FSE already knows in which part of the system the fault is. So the main problem with the current situation is that there is no good method available to the service engineer that he can use to quickly determine in which part of the system the fault may be. The LEDS only cover a small part of the system, and the information in the log files is fairly unclear to the FSE.

This problem is further underlined in interviews at PMS, where it is indicated that one of the main problems is that too much non-faulty hardware is exchanged in the current situation. This happens in cases where the FSE has no good idea where the problem lies and he replaces components to see if one of them was faulty. This leads to higher maintenance costs than necessary, since spare parts are expensive.

The interviews with FSE’s also indicate that most of the methods mentioned above are not used. One reason is because they are quite difficult to apply, and another is indeed because they take a lot of time, and before a FSE can invest this time in such a method he needs to be quite certain that the problem is indeed in that part. The last problem indicated is that FSE’s are not specialists for the PMS Cardio/Vascular X-Ray system, they are trained and have to service all systems of PMS (MRI, CT, Cardio/Vascular X-Ray, General X-Ray, Ultrasound). This is the reason that they only have medium knowledge about the system under diagnosis.

Therefore there is a real need for a diagnostic solution that can be applied to the entire system and that can quickly narrow down the search area to a small part of the system that presumably contains the fault. Ideally, the solution should indicate a number of components that are possibly broken ranked from the component that has the highest probability of being broken to the component that has the lowest probability of being broken. If the list is still too long or the top of the list contains a number of components that are close in probability of being broken then the FSE can also use any of the above methods to find the faulty component.

3.3 High Level Requirements

From the analysis of the assignment and the current situation and interviews with people from PMS, we are able to derive a number of requirements that should be met by the diagnostic solution that is to be designed. These requirements are listed in Table 3.1.

Nr.	*Requirement*	*Priority*
1	The solution should be able to cope with different hardware configurations of the machine.	High
2	The solution should be easily maintainable with future upgrades/changes in the hard and/or software design.	Medium
3	The solution should be extendible in fault diagnosis according to field experience.	High
4	Extension of the solution as described by req. 3 should not lead to changes in the target software.	High
5	The solution should be able to perform fault diagnosis on-line	Medium
6	Fault diagnosis should be performed with minimal physical intervention from a Field Service Engineer	Low
7	Presentation of the diagnosis should be clear to the Field Service Engineer	Medium
8	The presented diagnosis should be reliable	Medium/High
9	It must be possible to integrate the solution within the current Field Service Framework	Medium/High
10	The solution should be able to quickly narrow down the search area of the fault to a small part of the system	High

Table 3.1 Requirements of diagnostic solution

The priorities of the requirements are assigned by the project supervisor at Philips Medical Systems. All the requirements will be briefly described below.

3.3.1 The solution should be able to cope with different hardware configurations of the machine

There are many possible configurations for the cardio vascular systems that are under diagnosis. For instance, a system can have multiple input units, different tables and/or different arcs that house the X-ray tube and detector.

The solution that will be designed must be able to cope with these different hardware configurations, preferably, with minimum user effort.

3.3.2 The solution should be easily maintainable with future upgrades/changes in the hard/software design

The machines that are produced at Philips Medical Systems are in continuous development. According to new user experience and new hardware developments, the design of the machine changes frequently to incorporate these new developments.

The solution that will be designed should be able to cope with these changes in the hardware and/or software design and these changes should preferably be reflected in the solution with minimum effort.

3.3.3 The solution should be extendible in fault diagnosis according to field experience

During the design of the machine, the developers keep in mind what could go wrong and try to incorporate this knowledge in for example the logging. This helps finding those problems more easily. However, experience has shown that in practice, these machines also break down in ways that were not foreseen by the designers.

Therefore, the solution should be adaptable to find errors that were not thought of during design, but show up according to field experience. This way knowledge gained in the field can be shared amongst all other field service engineers.

3.3.4 Extension of the solution as described in req. 3 should not lead to changes in the target software

Since the software created at Philips Medical Systems is used in medical machines, there are very strict rules for testing the software before it can be released for public use. If changes are made to the target software, it must be tested extensively before it can be released. This testing costs about three weeks for a full team of testers.

The solution should be designed in such a way that it is extendible as described in requirement 3, but this extension should not require any changes to the target software, such that the target software does not need to be re-tested.

3.3.5 The solution should be able to perform fault diagnosis on-line

The solution should be able to perform the fault diagnosis while the machine is functioning and within a reasonable amount of time. Reasonable amount of time is specified as that the user should not have to wait for minutes before the diagnosis is made.

3.3.6 Fault diagnosis should be performed with minimal physical intervention from a Field Service Engineer

The solution should be able to perform fault diagnosis with minimal input from the field service engineer. Input from a field service engineer takes time, and ideally the solution should be able to diagnose the machine automatically without user input.

A second reason for this requirement is that Philips Medical Systems has recently implemented the ability to remotely connect to the machine. Having a solution that can perform fault diagnosis without any specific actions taken by the Field Service Engineer could greatly improve Field Serviceability since the Field Service Engineer can then be deployed to the location with prior knowledge about what could be wrong with the machine.

3.3.7 Presentation of the diagnosis should be clear to the Field Service Engineer

The solution that is to be designed should also focus on the presentation of the diagnosis. This presentation should be clear to the Field Service Engineer, given that they have a different level of knowledge about the machine than a developer has.

3.3.8 The presented diagnosis should be reliable

The solution should give a reliable diagnosis to the Field Service Engineer. In case of a false positive, expensive components could be replaced and the problem would still exist. In case of false negative even more time will be spent on other non-malfunctioning components, which also costs a lot of money. For example, if the communication to a certain component is unavailable, the diagnosis should not be that the component is malfunctioning, but that the communication channel is broken.

3.3.9 It must be possible to integrate the solution within the current Field Service Framework

There exists a current diagnostic framework within the C/V System of Philips Medical Systems to perform certain maintenance tasks and diagnostic tests on the system. The solution that is to be designed does not have to be designed within this framework, because it is a feasibility project to see whether the method works in practice. However, when Philips Medical Systems decides to use the method, it might be integrated within this diagnostic framework, so the solution has to be designed in such a way that integration within this framework is feasible.

3.3.10 The solution should be able to quickly narrow down the search area of the fault to a small part of the system

There currently exist many methods to find a fault in a certain part of the system. However, there is a need for a method that can diagnose the whole system and then indicate in which part of the system the fault is located. The diagnostic solution should primarily focus on narrowing down the search area for the fault to a small part of the system. If possible it would be nice if the solution can also indicate which component(s) are likely to be broken, but this last step in diagnosis can also be achieved by using some of the already available methods.

3.4 Assumptions

From the analysis of the solution and the system, a number of assumptions can be derived that hold for this assignment. When designing the diagnostic solution, these assumptions will be taken into account. That means that the diagnostic solution will work in a situation where the assumptions hold, but might not work when one or more of the assumptions do not hold. The assumptions are listed in Table 3.2.

Nr.	*Assumption*
1	The system can be divided into a number of modules, each of which has its own defined function
2	The different modules in the system are mostly passing signals to each other
3	Components in modules are either broken or functioning correctly
4	The software is functioning correctly

Table 3.2 Assumptions on system under diagnosis

3.4.1 The system can be divided into a number of modules, each of which has its own defined function

The system that is under diagnosis consists of a large number of components. These components can be grouped into modules, where each module has its own defined function. For instance, there is a power module whose task it is to convert the incoming power and distribute it to the other modules/components that need power (and with the right voltage/power).

With this assumption the diagnostic solution can diagnose the modules separately.

3.4.2 The different modules in the system are mostly passing signals to each other

Most of the communication between the modules in the system can be seen as signals being sent from one module to another. Observations can be done whether or not signals are passed to another module. If a signal is sent then the observation is True if the signal is not sent then the observation is False.

With this assumption it is known for defined situations which signals should be sent and received, and which signals should not. The diagnostic solution can use this information to derive a diagnosis.

3.4.3 Components in modules are either broken or functioning correctly

In real systems it can happen that a certain component breaks partly, in which case it can work sometimes or it can show strange behavior. For instance when a cable is partly broken, some controls might be working when the system is in a certain position, but when the system is moved to another position, the same controls (that could be completely independent of the moving controls) might not work anymore (because the partly broken cable is more stressed in that position, breaking its conductivity). These kinds of errors are very hard to find and for this project we consider them out of scope.

With this assumption the diagnostic solution will focus on persistent errors and assumes that if a component is broken it will never show working behavior.

3.4.4 The software is functioning correctly

Since the goal of our diagnostic solution is to find hardware faults and we are trying to find them using software, we assume that the software is working correctly. This assumption is necessary to avoid making the project too complex.

With this assumption the diagnostic solution does not have to take software errors into account.

3.5 Analysis conclusion

One of the most important conclusions from the interviews with people from PMS is that the Cardio/Vascular X-Ray system contains enough information to let experts determine what the problem is in a reasonable amount of time. When an FSE cannot find the problem and the 1st and 2nd line helpdesks cannot help him, finally the problem is forwarded to a developer at PMS and using log and trace files they can determine the problem most of the time. This means that all the necessary data is already available in the system and minds of developers.

The key improvement to fault diagnosis is to find a way in which a diagnostic system can incorporate this developer knowledge and use it together with the information that is already available inside the system and derive a diagnosis from it.

4 Research Survey

This chapter gives an oversight of the research that has been conducted into the area of fault diagnosis.

4.1 Information sources

As information sources for fault diagnosis, my analysis and research has been limited to literature and interviews. These interviews have been done with different people from different departments within Philips Medical Systems. The departments that were interviewed were Development, Service Innovation and Field Monitoring Team of the Cardio Vascular Business Unit.

The development department is the group in which this project is carried out. It consists of the people (architects, designers, developers, testers) that design and create software for the system. This department is used as an information source since they know how the system is built, how the architecture for the system was designed and how it was implemented. Furthermore, they have a good understanding of what might go wrong and some of the people working in this department also have contacts with service engineers that work on these machines in the field (the Field Service Engineer).

The Service Innovation department supports the Field Service Engineers with information. They have a knowledge base system in which a Field Service Engineer can look for similar problems as the ones that he encounters, and he can look for possible solutions in this knowledge base. The Service Innovation department is also responsible for creating so-called Fault Isolation Procedures (FIPS), which embody an effect-to-cause reasoning to isolate the fault in the system. Another important task for the Service Innovation department is to define which parts of a machine are to be replaced as a whole in case of malfunctioning. These parts are called Field Replaceable Units, and in case of a hardware malfunction it is the task of the Field Service Engineer to locate which Field Replaceable Unit is malfunctioning and replace that one.

The Service Innovation department also works closely with people from the development department to get the necessary information for updating their knowledge base, update the FIPS and create service documentation inside the machine software.

The Field Monitoring Team department is responsible for assisting Field Service Engineers when a machine with new technology is shipped to a customer. At that time, there is not much experience with the machine nor do the FSE’s have much experience with it. In these situations, the Field Monitoring Team helps with finding faults and solving errors in machines.

4.2 Comparison criteria

To make a comparison between various fault diagnosis methods, certain criteria of interest must be pointed out that will be used for the comparison. The criteria that are presented in this paragraph are mostly derived from the requirements as described in [2]. In addition, the criteria are also based on the "desirable attributes of a diagnostic system" as described in [3]. The reason why not all desirable attributes from [3] are used as comparison criteria is because they are not all equally relevant for the situation at Philips Medical Systems. Only the relevant attributes are used as comparison criteria.

The comparison criteria that are important for this project are:

- Accuracy
- Resolution
- Completeness
- Variability/Configurability
- Extendibility
- Representation
- Automation
- Development costs
- Maintenance costs

For each criterion will be described why it is important to Philips Medical Systems.

Accuracy

Accuracy is an important criterion for any diagnostic system. Accuracy corresponds to the correctness of the diagnosis. A high accuracy diagnosis will contain little false positives or false negatives. This is an important aspect for PMS because a false positive might lead to the replacement of a non-defective component, leading to superfluous costs. False negatives are just as bad as they might steer the FSE away from the faulty component leading to higher repair times. This means more hourly costs of the FSE and a higher downtime of the machine. So accuracy of the diagnosis is important.

Resolution

Another important criterion for any diagnostic system is resolution. Resolution is the ability of a diagnostic system to minimize the set of fault candidates. A diagnostic system with a low resolution will give a large set of possible faults. Within the situation of PMS this means that the FSE has to spend a longer time determining which of the proposed components is the faulty one. This leads to longer repair times, leading in turn to higher hourly costs of the FSE and higher downtime of the machine. High resolution means that there are only few fault candidates proposed by the diagnostic system, however, this increases the probability of the actual fault being not in the set of proposed candidates.

Completeness

Completeness is the property that the actual fault is part of the set of proposed fault candidates. So the higher the completeness, the higher the probability that the actual fault is part of the diagnosis result. However, there is a tradeoff between resolution and completeness. Having a diagnosis system that gives the set of all components as a fault candidate is complete, but its resolution is very low. Therefore, the right balance must be found between completeness and resolution. The importance of completeness within PMS is already described under the head accuracy; missed faults leads to extra costs because the FSE is steered away from the actual defect. Secondly, missed faults might also decrease the confidence in the diagnostic system.

Variability/Configurability
A method that can deal with variability of the system is important to Philips Medical Systems since their C/V system can have many different configurations. The machine is divided into different components, for instance the table and the arc. Different versions are available of these components, since all are under continuous development. A hospital can choose to upgrade their table for instance, while keeping other components. This can lead to many different configurations of the whole machine, and ideally, this should be made transparent in the diagnostic system because making a separate diagnostic system for each possible configuration would mean that not one but more than ten diagnostic systems should be made. This leads to an unwanted increase in development and maintenance costs. Therefore a diagnosis method should be able to diagnose all different configurations of the system without an exponential increase in development and maintenance costs.

Extendibility
The fact that different components are under continuous development also leads to the fact that hardware and software can change over time. New hardware and changes to the software should not lead to rigorous changes in the diagnostic system. Neither should additions lead to much change in the diagnostic system, other than adding new diagnostic functionality.

Representation
Service engineers that are employed by Philips Medical Systems are multi-modality trained. This means that they are not just trained on one specific system, but on many different systems, including MR, General X-ray, C/V X-ray, CT, Nuclear Medicine and Ultrasound systems. The downside of this is that service engineers do not have specific in-depth knowledge of all these machines. Therefore, it is important for the diagnostic system to give a clear description of its diagnosis that is meaningful to the service engineer.

Automation
Automation of the diagnosis is also important to Philips Medical Systems. To make the diagnostic system as automated as possible, the observations of the machine should be done by the diagnostic system. Ideally, the state of all components in the machine should be determined by software without human interaction. For security reasons however, certain actions are not allowed to be done fully automatically. For instance, movement and X-ray control either must be manually engaged or require a dead-man's control key when done automatically. Therefore, manual input will always be necessary to start the diagnosis for example.
The downside of choosing not to have any human measurements on the machine is that resolution of the diagnosis cannot be as high as when human measurements could be done on the machine. For instance, it could be impossible for software to distinguish

between the case in which a certain hardware switch is broken, or that the cable that connects that hardware switch is broken.
So human interaction is allowed, although it should be minimized.

Development costs
Since PMS is a commercial company, development costs are an issue. Development costs of the solution should be reasonable, but the precise definition of reasonable cannot yet be made. When a prototype of the solution is finished, a much clearer view should be available on costs that would be involved in developing the solution for the whole system. The decision whether the costs are reasonable and whether it is useful to extend the solution to the whole system can then be made by the PMS management.

Maintenance costs
For maintenance costs the same holds as for development costs. If this solution is used in practice, its costs for maintaining and extending the diagnosis capabilities should be reasonable. This also can only be decided after a prototype is available that offers a clearer view on the costs.

4.3 Fault Diagnosis Methods

This paragraph will give an overview on different fault diagnosis methods that are known today. For each method, strong points and weak points will be given. The comparison using the criteria described in the previous chapter will be given in the next paragraph.

4.3.1 Overview

There are many different methods available for fault diagnosis. These can be classified in a number of ways. One way of classifying them is by what information they use for fault diagnosis. This is done in [3], [5], [6] and [7]. The major distinction is made by classifying methods that use process model information and classifying methods that use process history information.

4.3.2 Process model based methods

Methods that fall under this category are methods that use certain knowledge about the process that is under diagnosis. This knowledge can be captured into a model of the process. The process knowledge can for instance be knowledge on how components are interconnected or how communication between processes is done. From this knowledge models can be constructed that describe the conditions of nominal or faulty behavior. These models can then be compared with observed behavior of the device and a diagnostic engine can then draw conclusions whether the functioning of the device is faulty or not. Using this information, it can also try to find the root cause of the error.

For model based fault diagnosis, mainly two types of models can be used. These are the fault models and consistency based models [10], [11]. The differences between these two subcategories of model-based diagnosis will be described in the following paragraphs.

4.3.2.1 Fault models

One subcategory of model-based diagnosis is methods that use fault models to describe the system. Fault models embody a form of effect-to-cause reasoning. A symptom (effect) is observed and using the fault model can be reasoned back to a possible cause of this symptom.

The way in which fault model based diagnosis globally works, is shown in Figure 4.1.

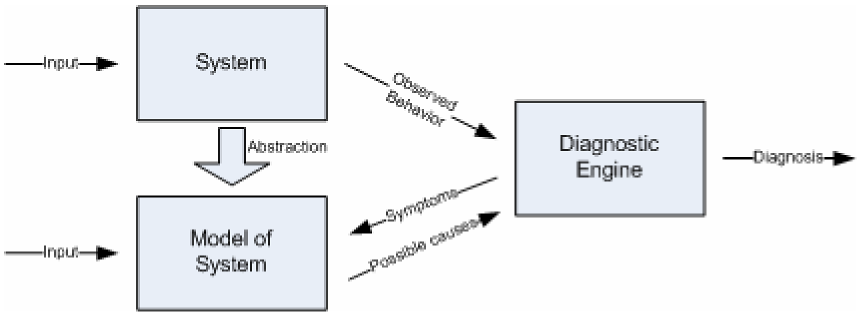


Figure 4.1 Fault model based diagnosis

Designers make a model of the system that is to be diagnosed. This model contains abductive information about symptoms and possible faults. Observations of symptoms in the system are then used for abductive reasoning in the model, which leads to a set of possible faults that are consistent with the observations. This set of faults is then presented as the diagnosis.

However, the abductive reasoning that is used leads to several problems. First of all, effect-to-cause reasoning uses rules that are of the form:

$$Fault \rightarrow Symptom$$

Then based upon the observed symptoms, a conclusion is draw:

$$Symptom \vdash Fault$$

This is logically unsound [12], [8] and therefore every *Fault* that has *Symptom* as a consequence might be a cause of the error. This leads to many hypotheses being generated [3], especially because in fault models the relationships are more complex than described above (due to chaining, common causes and common effects). Consider the following rules:

$$F_1 \rightarrow F_3$$
$$F_2 \rightarrow F_3$$

$$F_3 \rightarrow S$$

Observation of Symptom *S* leads to the following conclusion:

$$S \vdash F_1 \vee F_2$$

I.e. both fault F_1 and F_2 could have caused the symptom.

To solve this, additional effect information (more observed variables) is needed to gain a higher level of resolution.

Another problem of this reasoning is that there might be some fault F_u that is yet unknown, but which also has *S* as a consequence. This fault will not be hypothesized and therefore this diagnostic method is incomplete when not all fault possibilities have been exhaustively explored (and this is not feasible for complex systems).

As an example, consider the simple system of a power supply, switch and a light bulb shown in Figure 4.2.

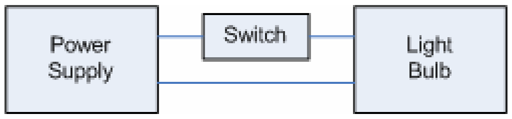


Figure 4.2 Light bulb example

A fault model for this simple example could be:

"The fuse in the power supply is blown" → *"There is no power"*
"The power supply is overheated" → *"There is no power"*
"The power supply is short-circuited" → *"There is no power"*
"There is no power" → *"There is no light when the switch is on"*
"The switch is broken" → *"There is no light when the switch is on"*
"The light bulb is broken" → *"There is no light when the switch is on"*

This extremely simple example already shows that the symptom *"There is no light when the switch is on"* can have many different causes. Moreover, this simple example already is incomplete. The following rules could be added for example:

"The voltage setting for the power supply is improperly set" → *"There is no power"*
"The light bulb is improperly fitted" → *"There is no light when the switch is on"*

This demonstrates that it is really hard (if not impossible) to catch all ways in which a component can fail in advance.

4.3.2.2 Consistency based models

Consistency based fault diagnosis uses one or more models to describe the structure and the behavior of a device. The main advantage of consistency model based diagnosis is that knowledge of specific faulty behavior of subcomponents is not needed. The structural and behavioral models describe how the device is supposed to work and from this information it can be derived if a certain component is not working correctly.

The way in which consistency model based diagnosis globally works, is shown in Figure 4.3.

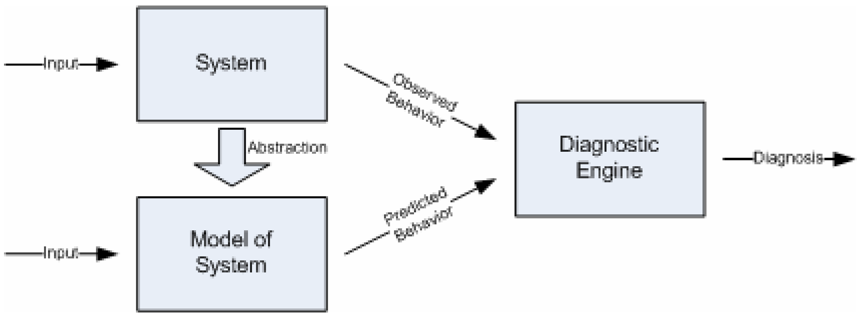


Figure 4.3 Consistency model based diagnosis

Designers make a model of the system that is to be diagnosed. This model is an abstraction of the system. Observations of the system are then compared to predictions of the model. This comparison can be done manually and automatically. The diagnostic engine can then draw conclusions from differences or similarities and can turn these into a diagnosis.

Consistency model based diagnosis uses rules of the form:

$$\neg Fault \rightarrow \neg Symptom$$

This means that as long as a certain *Fault* has not occurred, a certain *Symptom* should also not occur. Now if *Symptom* occurs, the following reasoning is used [8]:

$$Symptom \vdash F$$

This is logically sound which means that one can deduce (using logic reasoning) that *Fault* has occurred.

Modeling the incomplete light bulb example using consistency based models gives the following:

"The power supply is working" → *"The system is powered"*
"The switch is working" → ((*"The system is powered"* ∧ *"The switch is on"*) ↔ *"The light bulb is powered"*)
"The light bulb is working" → (*"The light bulb is powered"* ↔ *"The light is on"*)

If we now observe ¬*"The light is on"* ∧ *"the switch is on"* and we assume correct functioning of all components, we get a contradiction[1]. Since we have observed ¬*"The light is on"* ∧ *"the switch is on"* there must be an error in our assumption about the correct functioning of all components, i.e. one of the components must not be working correctly. Removing any of the assumptions that a device is working correctly would no longer give a contradiction, so for this system the set of fault candidates is:

(¬*"The light bulb is working"*, ¬*"The switch is working"*, ¬*"The power supply is working")*

This small example shows the main difference between consistency based models and fault models. The impact of the different techniques looks small in this example, but on larger examples where the propositions have much more mutual correlation, the difference gets bigger.

Because contraposition is logically sound, a diagnostic engine can systematically remove one of the "component is working" assumptions and try if it can still reach a contradiction. As soon as a contradiction can no longer be derived, that component is a possibly candidate for being broken.

Secondly, because of soundness, if removing an assumption leads to certain behavior that was not observed, that component can be rejected as a possible candidate. For example, if there is a second light bulb that is connected to the same power supply (without a switch), the following rule could be added to the model:

"The 2nd light bulb is working" → (*"The system is powered"* ↔ *"the 2nd light is on"*)

If we now observe ¬*"The light is on"* ∧ *"the switch is on"* ∧ *"the 2nd light is on"* we again get a contradiction when assuming the correct working of all components. However, when we investigate the assumptions about working components, we come to the conclusion that removing the assumption that the power supply is working correctly still leads to a contradiction[2]. This means that the power supply is no longer a fault candidate.

The diagnostic power that is available through this reasoning does however require clever models. For instance, if we would have chosen to extend the model in this way:

"The power supply is working" → *"The 2nd light bulb is powered"*
"The 2nd light bulb is working" → (*"The 2nd light bulb is powered"* ↔ *"the 2nd light is on"*)

The observation ¬*"The light is on"* ∧ *"the switch is on"* ∧ *"the 2nd light is on"* would still give the power supply as a possible candidate[3]. Therefore modeling is a non-trivial activity [13] that should be performed with care using knowledge from experts.

[1] Proof can be found in Appendix A under Proof 1
[2] Proof can be found in Appendix A under Proof 2
[3] Reasoning why is given in Appendix A under Reasoning 3

4.3.2.3 Consistency vs. fault models

From the previous analysis can be concluded that consistency model based diagnosis clearly has some advantages over fault model based diagnosis [10][13]. The main advantage is that not all fault modes of a component have to be investigated and modeled. Instead, the normal behavior can be modeled and a consistency based diagnostic system can then indicate which component is not functioning as is modeled. Another advantage is that the reasoning used by the diagnostic system is logically sound and therefore it will lead to a fewer number of candidates than fault model based diagnosis (as is explained earlier).

However, the last example of the previous paragraph (where the power supply still is a possible fault candidate although the 2nd light bulb is on) is still worrying.

When we investigate this example further, we conclude that humans are able to reason that if the (only) power supply is broken, it is never possible for a light to be on. From a logic point of view, it is possible to add this "fault behavior" information to our consistency based model. To recapitulate, we have to model:

"The power supply is working" → *"The system is powered"*
"The switch is working" → ((*"The system is powered"* ∧ *"The switch is on"*) ↔ *"The light bulb is powered"*)
"The light bulb is working" → (*"The light bulb is powered"* ↔ *"The light is on"*)

To which the following rules were added:

"The power supply is working" → *"The 2nd light bulb is powered"*
"The 2nd light bulb is working" → (*"The 2nd light bulb is powered"* ↔ *"the 2nd light is on"*)

Now if we add the known fault behavior information:

¬*"The power supply is working"* → (¬*"The 2nd light bulb is powered"* ∧ ¬*"The system is powered")*

Then our "less clever" model will also rule out the power supply as fault candidate. So it might be useful to add fault information to the model. However, this gives again the problems that also occur in fault model based diagnosis, namely that a component can fail in a way that was not foreseen, leading to the exclusion of a possibly broken component from the fault set. I.e., if the power supply would be partly broken in a way such that the 2nd light bulb does receive power, but the switch does not, then the diagnosis that the power supply might be broken is not part of the fault set. So care must be taken when adding fault behavior to consistency based models.

4.3.3 Process history based methods

Process history based methods use information that has been gained from the device that is under diagnosis. This information could be statistics or cause-effect information that has been gathered over time during the operation of the device. An example of this information is a rule-based sheet that contains effects that might be observed from the device along with possible causes (and solutions) for each effect.

Figure 4.4 gives a global view of process history based reasoning.

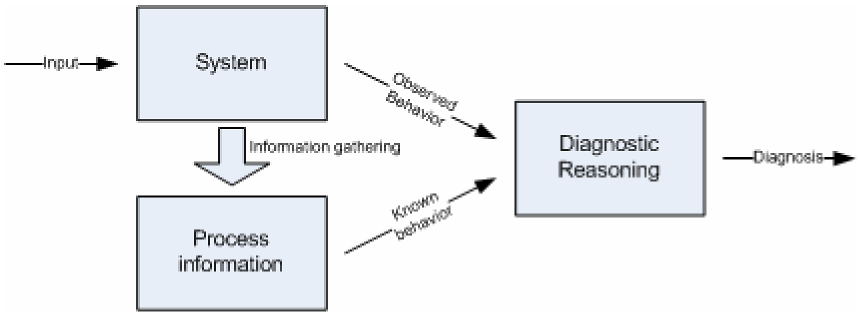


Figure 4.4 Process history diagnosis in general

During design and especially field operation, information is gathered about the (mal)functioning of the system. This information is stored, and can later be used in diagnostic reasoning when an error occurs. The diagnostic reasoning in this case can be done manually, but it can also be automated by a diagnostic engine.

There are many process history based diagnosis methods, Neural Networks, Bayesian Networks, Rule-based, Case-based and statistical analysis are examples of them.

The downside of all process history based diagnosis methods is that they need large quantities of process information to train their diagnosis capabilities. This process information should consist of information of the observed behavior of the system at the time of a fault. The fault and its accompanying observed behavior should then be linked together and are used as input for the process history based diagnostic methods.

Unfortunately this information is not stored at the moment. When Field Service Engineers fix a system a work sheet is filled in with basic information of the symptoms and what actions were taken to fix the system. However, this information is not detailed enough to be used as process information since not all symptoms are precisely described (often only the main problem is described) and usually many components are replaced and it is then not known what the actual faulty component was. Because of the unavailability of this information, process history based methods are not considered when looking for a solution.

4.4 Comparison

In the next paragraph, a comparison will be given between various fault diagnosis methods. This comparison will be used in the last paragraph to motivate a conclusion of which method is the best to use. Although the comparison will be made as objective as possible, the considerations in this chapter are made while considering the situation at PMS. This means that the claims that are made of the various methods might not hold for every situation, but they hold when applied to the situation at PMS from my point of view. As a last point it has to be remarked that not all comparison criteria are precisely quantifiable, so that the evaluation and comparison might also be subjective to my opinion.

4.4.1 Fault Diagnosis Method evaluation

Before the global comparison, first the selected methods will be discussed using the comparison criteria, described in paragraph 4.2. The following methods will be discussed: Rule-based reasoning, Neural Networks, Statistical Analysis, Fault Trees and Consistency Models. Although paragraph 4.3.3 concludes that process history based fault diagnosis methods are not an option within PMS because of the lack of process history data, they are included in this comparison. The reason for this is that if they prove to be a good choice then PMS might decide to start storing process history data for use in future projects.

4.4.1.1 Rule-based reasoning

Rule based reasoning (RBR) is a form of process history based fault diagnosis. Rule based methods use abductive reasoning to reason from symptoms to causes. The main disadvantage of RBR is that the knowledge base gets very large.

Since the KB used for RBR gets very large and there might be many possible causes for a symptom, resolution of RBR is not very high. Because of this low resolution, the accuracy is also quite low. On the other side, if the KB is sufficiently large, it has a good completeness since the actual fault is likely to be in the set of proposed faults.

To describe variability or configurability of the system within RBR would require different cause-effect rules for different configurations (either by using different knowledge bases, or by adding the configuration as a conjunct to the antecedent part). Both are not desirable, so RBR does not score high on variability/configurability.

Extending RBR to incorporate diagnosis knowledge of new hardware is as easy as adding new rules. However, changes in existing hardware and/or software would require the whole knowledgebase to be revised (as to check whether the rules still hold with the new hardware/software setup). Therefore, RBR scores only medium on extendibility.

In the previous two paragraphs we have claimed that RBR needs a large KB to work, and that this large KB has a negative impact on changes in the existing hardware and/or software. This implies that RBR scores low on both development costs and maintenance costs.

RBR does not incorporate any features that allow an intuitive representation of the reasoning that was used to come to its diagnosis. The only form of representation is a list of possibly fault components based on abductive reasoning. This abductive reasoning can be done automatically, so it is possible to automate this method.

4.4.1.2 Neural Networks

Diagnostic systems based on neural networks (NN) also fall under the category of process history based methods. However, unlike RBR they do not require experts to create a huge KB with diagnostic rules. Instead, a NN can be trained to “learn” to recognize patterns and attach a diagnosis to them. The main problem with NN is that it requires a lot of data to train a NN and that it is hard to create a NN that is really predicting outcomes rather then just replaying the “learned” outcomes.

A good trained NN can have a high accuracy and high resolution, however especially new problems might not be found if the NN is either over trained or under trained. Therefore, this method scores medium on completeness.

If the NN is trained with the variability of the system as its input, it can learn to diagnose the different errors of the different configurations. Therefore, variability is good. Extendibility might be a problem however, since a trained network will have to be retrained to incorporate the new hardware. This again takes a large amount of training data. The same goes for changes in the current software and/or hardware of the system.

A NN does not have any representation facility, it cannot explain how it came to a certain diagnosis. So it scores low on presentation. Neural networks are meant for automated diagnosis, so it scores high there.

Finally, the costs of development are high, because of the need of the large training set. New training is needed when the system is extended with new hardware and/or software, or when the current hardware and/or software is changed. Therefore, maintenance costs are also high. Since this new training also requires new process information (which is not available when a new or changed feature is introduced), its extendibility is low because only after a certain amount of new process information is gathered can the NN be trained with this process information.

4.4.1.3 Statistical Analysis

Statistical analysis is mostly used to detect disturbances in a system based on statistical quality analysis. It can measure various parameters of a process and use statistical information on these parameters to predict the quality of the system (and it can try to predict that a certain component may break down based on this quality analysis of the system).

Statistical analysis scores badly on accuracy, resolution and completeness. This is because it can only predict the breakdown of a component based on statistics. Therefore, a component could break before its breakdown is predicted, or it could be working while, according to statistics, it should have broken down. Also some components might not have parameters through which a breakdown could be predicted.

On variability and extendibility this method scores good since new components can just have their individual statistics added to the complete diagnostic system, as long as their healthiness can be predicted through (observable) parameters. The same holds for the different components that are used in different configurations.

Since statistical analysis can only indicate that a fault is likely to occur in a certain component, it scores low on representation. Automation however is quite easy, only the necessary parameters need to be monitored and compared to statistical data. Development costs are medium since a lot of statistical information is needed before any good predictions can be made. Finally, maintenance is good; after the initial statistics are available only new statistics for new or changed components need to be added.

4.4.1.4 Fault Trees

Fault trees are a fault diagnosis method that uses causal chains to find the possible root causes of a symptom. Fault trees are a form of model-based diagnosis using fault models. This means that fault trees suffer from the disadvantages that are listed in paragraph 4.3.2.1.

The disadvantages described in paragraph 4.3.2.1 lead to the fact that fault trees score only medium on accuracy, resolution and completeness. They are better than RBR and statistical analysis since fault trees use more knowledge of the internal structure and functioning of the system.

To describe variability with fault trees it suffices to add nodes with the variability information to the tree on decision points where this variability information matters. This means that the variability information might be copied into different places in the tree, but no completely different trees have to be made for all different configurations. Therefore, fault trees score medium on variability. They score high on extendibility, because when extending the system, only fault information for the newly added components has to be added to the tree.

Fault trees also score high on representation, since the fault model represents an internal “workflow” sheet of the system. A representation of the fault tree can give a FSE more insight to the way the diagnostic engine came to its diagnosis. The fault tree can also be automatically checked, so on automation it also scores high.

The hardest part of all model based diagnosis methods is creating the model. For fault trees this means that experts of the system must make a division into modules and components and must clearly state what components are dependent of each other and in what ways a component can fail (and possibly how this failure can propagate through the system). Covering a high percentage of the errors in a system takes less effort than with RBR, so fault trees score medium on development costs. For maintenance costs it takes far less effort to extend the model than to create the model. However, maintaining the model to incorporate changes to the hardware/software still takes quite some effort. So although overall maintenance takes less effort than with RBR or NN, it still takes quite some effort so it scores medium.

4.4.1.5 Consistency Models

The next diagnostic method that is taken into the comparison is consistency model based diagnosis. This form of model-based diagnosis is more elaborately described in paragraph 4.3.2.2. The main disadvantage of consistency model-based diagnosis is that it is hard to create an adequate model.

Consistency model based diagnosis uses a model of the systems behavior and/or structure. The accuracy of the diagnosis depends on the detail of the model, but with a detailed enough model it scores high on accuracy. An accurate model also scores high on resolution, since the set of faulty components should be traced back to a small set (if the model is detailed enough). However, since the model will always be an abstraction of the system, there are always aspects that are not incorporated in the model. Some of these aspects might have an unforeseen impact on the system and cause an error, which will then not be detected (since it is not incorporated into the model). Therefore, it only scores medium on completeness.

Consistency model based diagnosis also does not have a natural way of dealing with variability of the system. The parts that can differ must be modeled separately and a way must be found to incorporate these different models into the model of the complete system. On the field of extendibility, this technique scores good. When extra hardware and/or software is added to the system, this can be incorporated into the existing model. Therefore, it scores high on extendibility.

This method is also good for representation, since it has an internal model of the structure and/or behavior of the system. This can be presented to the FSE and can be used to clarify its diagnosis, giving it a high score on representation too. Secondly, consistency model-based diagnosis is especially used in automated fault diagnosis.

The downsides of model-based diagnosis as discussed earlier lead to a medium score on development and maintenance costs. Development of consistency based models is hard, and requires the knowledge of experts. On maintenance costs, this technique scores a little worse than fault model based diagnosis methods since changes to the existing hardware and/or software will lead to bigger changes in the model than with fault model based diagnosis. However, the changes are still smaller than with RBR or NN.

4.4.1.6 Bayesian Networks

Bayesian networks could be classified as both a process history based method and as a process model based method. Bayesian networks use probabilities on conditional influences to calculate the belief of a certain hypothesis being true or false (hence it could be considered process history based), but the conditional influences also express causal relationships between nodes (variables) and hence it could be considered a process model based method (since the relationships are designed using white-box knowledge of the system under diagnosis).

The nodes in a Bayesian network represent process variables or certain conditions that can occur in a system. The directed arcs between the nodes represent causal influence between the nodes. The nodes also have a certain probability of occurring, or a certain probability of influencing another node.

Using Bayesian networks it is possible to calculate the probability of a certain component being broken, given a (possibly incomplete) set of observations. Components can then be ranked in order of possibility of the component being broken given the observations.

Bayesian networks can also be fully created by experts, or they can be trained like neural networks. The Bayesian networks we consider for our purpose are those that are fully created by experts, since the large amounts of data needed for training the networks are not available.

An advantage of Bayesian networks is that they can be used in situations were a full behavior model is not possible (the complexity increase is less when the systems get bigger than with model based diagnosis), or when the input/observable data is incomplete or contains flaws.

The disadvantage of Bayesian networks is that process data is needed to give probabilities to the nodes, which requires either a certain amount of process history data to be available, or these probabilities have to be estimated by experts.

4.4.2 Comparison overview

For the comparison overview a scale from -- to ++ will be used where -- means that the diagnostic method scores very bad on that property and ++ means that the diagnostic method scores very good on that system. Table 4.1 shows the results of the comparison.

	Accuracy	Resolution	Completeness	Variability	Extendibility	Representation	Automation	Development costs	Maintenance costs
Rule-based reasoning	-	--	+	-	0	-	+	-	-
Neural Networks	+	+	0	+	--	-	+	-	-
Statistical Analysis	-	-	-	+	+	-	+	0	+
Fault Trees	0	0	0	0	+	+	+	0	0
Consistency Models	+	+	0	-	+	+	+	0	0
Bayesian Networks	0	0	+	-	+	0	+	0	+

Table 4.1 Comparison of fault diagnosis methods

Any conclusions from this overview will be made in paragraph 4.5.

4.5 Research conclusion

First a short conclusion on each of the compared diagnosis methods will be given. After that a general conclusion will be given, and this conclusion will be used to decide on the direction that will be followed.

4.5.1 Individual conclusions

This paragraph will give the conclusions for the individual methods applied to the situation at PMS. After that a general conclusion will be given in the next paragraph.

Rule-based reasoning
Pure Rule-based reasoning will not be very useful for PMS because the system that is under diagnosis is a very complex system. This means that the knowledge bases that have to be used will get too large and very hard to maintain. Plus Rule-based reasoning by itself does not take any advantage from extra information that is available about the structure and modules of the system.

Neural Networks
Neural networks are not useful in the current situation of PMS since the large quantities of training data that are needed are not available. The way in which current problems are

solved, it is not always clear what the root cause of an error was. Sometimes many components are exchanged in one go, and if the machine then works there is no further investigation of which component was the faulty one. Furthermore, the root cause of errors is not always logged after the error is solved, so there is no central information store that has all this information. If neural networks were to be used, first such a central information store must be installed and the general way the field service engineers work must be changed.

Secondly, it is not certain that NN provides the level of fault diagnostics needed at PMS since there is many input data and there is no precise method on deciding which size of a NN is optimal for this problem. Therefore, NN are not further considered.

Statistical Analysis
Statistical analysis on its own will not provide a good way of fault diagnosis for the situation at PMS. Many of the components do not have observable parameters of which the quality of that component can be predicted. Secondly, as with neural networks, the large amount of statistical data that is needed is currently not available.

Fault statistics could however be easily implemented in addition to another method, since it can just be used to increase the accuracy and resolution of that method using statistics.

Fault Trees
Fault trees could be a good method for fault diagnosis at PMS, and manual forms of fault model-based diagnosis are already used within PMS. The downside of this method is that it remains hard to capture all ways a component could fail and because of the large and complex coherence of the components in the C/V systems of PMS these fault trees would get extremely large, and therefore hard to maintain.

Consistency Models
Consistency based models are a good method for fault diagnosis at PMS. It solves the problem of having to capture all ways a component could fail in advance and also provides good reasoning representation capabilities of its internal reasoning. However, the problem remains that in the C/V systems of PMS these models get very large and complex and are therefore hard to maintain. A small disadvantage compared to fault trees is that certain specific errors are less easily captured (because in pure consistency model based diagnosis, it is not possible to directly describe errors in terms of observable events).

Bayesian Networks
Bayesian networks could be quite useful in the current situation of PMS, since they can be constructed using model based knowledge and they can also incorporate statistical data about component failure probabilities. More importantly, they scale better to bigger systems (the complexity increases at a lower rate) than pure model based methods, but on the downside they might produce a lower resolution than consistency based models.

4.5.2 General conclusion

The main conclusion that is also drawn in the literature is that there is no single solution that satisfies all requirements of diagnostics [3][16]. Hence a combination of some of these methods must be used to create a solution that is optimal for the given requirements.

From the research done in various papers can be concluded that process model based diagnosis has preference over process history based diagnosis. The reason for this is that diagnostics can be done without large quantities of available fault information. Within model-based diagnosis, consistency based models also have a preference over fault models. The reason for this is that not all fault modes of a component need to be captured in advance.

However, making a consistency based model for a full-scale application would not be feasible in practice [10][16]. A system wide model would have to incorporate state and time, since these play a role in the C/V System of PMS. Modeling state and time adds extra complexity to the solution, and should be avoided if possible. Furthermore, a model of the whole system would be very large, making extendibility and maintainability quite poor.

Therefore, a method needs to be chosen that would not require making a model of the complete system, but that could still benefit from the advantages offered by consistency model based diagnosis.

Recommendations for the method to be designed

The recommendation is to design a method that uses a form of “loose” model based diagnosis. In addition, this form of diagnosis should not be performed on the system as a whole, but the system should first be divided into a number of smaller modules. Then this form of diagnosis can be performed on these smaller modules.

The way the diagnosis on the smaller modules should be done is by comparing observed behavior with a model. The model should then describe how the system is supposed to behave in a certain situation. This behavior should be measured from observations whether control signals have passed at a certain place or not. A method to gather this information is elaborated in chapter 5.

To avoid the addition of state and time to the diagnostic solution, models should be described in terms of signals that have passed at certain points in the code. To be more concrete, between two given moments (for instance, the start of the diagnosis and a defined time period later), the model should describe whether a signal should have passed there or not.

The places in the code where there should be checked whether the control flow has passed there should be considered by experts, but some places that are generally interesting can be defined. Such places are:

- Interfaces between software and hardware

- Software decision blocks (where software decides whether something is allowed or not)
- Software exception handlers

Though software errors are out of scope, it is important to place observation points in software decision blocks so that the diagnostic system knows that a certain action was disabled (or not) by the software. This avoids that such a situation is mistakenly interpreted as a hardware failure.

Furthermore it is interesting to place observation points in software exception handlers that are related to the hardware. The software contains internal checks of hardware input and output signals. These checks might determine that something is wrong (for instance a sensor value is over a certain threshold) and raise an exception. An observation point here would be valuable because the diagnostic system could deduce what could be the cause of this observation.

Since modules are assumed to have their own defined behavior, it should be known for certain situations what output signals the module should produce. If these output signals are produced then the module is assumed to function ok. If these signals are not produced, the fault should be inside the module. This reasoning introduces a form of dependencies, i.e. if a certain module B relies on module A, then module A should be checked before module B, because the malfunctioning of module A implies that module B also cannot function (while the fault is not in module B). This means that the solution must take care of handling these dependencies in the proper order.

A simple example of this reasoning scheme is shown in Figure 4.5.

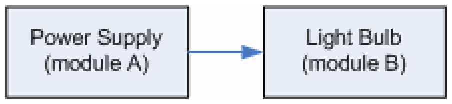


Figure 4.5 Module that has a dependency on another module

Here the light bulb (module B) cannot function without the power supply (module A) functioning correctly. Therefore we say that the light bulb has a dependency on the power supply (indicated by the arrow between the modules) and therefore the power supply should be checked on correct output before the light bulb is checked.

5 Diagnosis Method Description

In this chapter we will describe the method that we have designed to improve fault diagnosis in more detail.

5.1 Introduction

Before explaining the method in detail, we will first give a short introduction.

5.1.1 General diagnosis process

The diagnosis process can be generalized as shown in Figure 5.1.

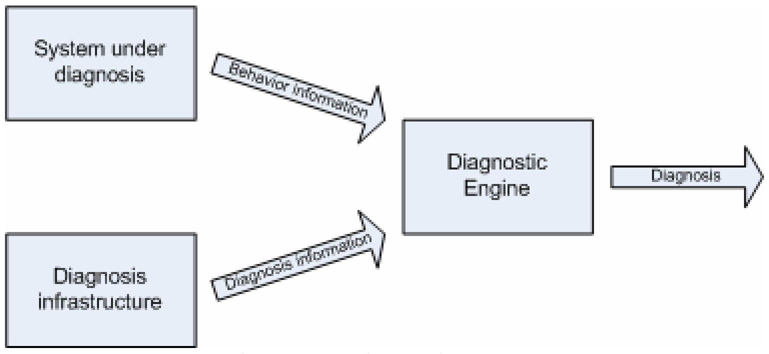


Figure 5.1 Diagnosis process

Behavioral information is collected from the system that is under diagnosis. This might be external observable information, but also internal information (such as logging) or information that is obtained through measurements.

Besides behavioral information, diagnosis information is needed. This diagnosis information is used by the diagnostic engine to determine if the system is functioning correctly. If the system is not functioning correctly, this information is used to determine what components could be broken.

5.1.2 Diagnosis process for Philips Medical Systems

As shown in Figure 5.1 the diagnosis process has three entities. We will now investigate what the requirements on each of the entities are for the specific situation at Philips Medical Systems such that we can define a diagnosis method that satisfies the goal.

System under diagnosis
The system that is under diagnosis needs to supply behavioral information to the diagnostic engine to allow it to derive a diagnosis of the current situation of the system. There are many sources of information that may potentially be used. These sources are explained in Table 5.1.

Information source	*Explanation*
External Observables	We define external observables as observations that can be made without using additional tools. Leds that indicate the status of some component are external observables, but also a visible movement, a clicking sound of a clutch or any information on the screens can be seen as external observables.
Logging	The software of the system generates logging which contains information about events that occurred in the system. This logging might also indicate where problems could be.
Functional tests	The functional tests can provide basic information whether some function of the system works or not.
POST information	The Power-On-Self-Test performs a number of tests during startup of the functional unit. The results of these tests can provide some information about basic features that are not functioning.
BIST information	The Built-In-Self-Test performs a number of tests on a functional unit. The results of these tests can provide some information about basic features that are not functioning.

Table 5.1 Available sources of behavioral information

Logging and POST information can be extracted from the system automatically. External observables can only be extracted manually and Functional tests and BIST information can be extracted automatically, but these tests need to be initiated manually. Since we are looking for an automatic software solution with minimal manual input, we do not consider external observables as an option.

We also will not consider Functional tests or BIST information because they already require knowledge about in which part of the system the fault may be. But the software solution that we are looking for actually has as goal to give the service engineer a good idea in which part of the system the fault may be. Another remark about the POST, BIST and functional tests is that they cover only a certain percentage of the faults that can occur (estimated by PMS at between 60% and 70%).

This leaves the choice between choosing the Logging and POST as information sources to get behavioral information and creating a new source of behavioral information. The disadvantage of using the logging is that it is in an almost free format and the logging only provides some information, mostly about errors that have occurred. But in diagnosis, it might be just as interesting to know whether some component is working correctly since it can then be removed from the list of possible candidates.

Since we are interested in behavioral information concerning errors but also concerning correct functioning of components, and we do not want to be dependent on the presence of logging at a certain point where we want information, we choose to add our own behavioral information sources to the system.

Diagnosis information
Diagnosis information is information that is needed by the diagnostic engine to derive a diagnosis when given the behavioral information gathered from the system under diagnosis. In our method, this diagnosis information consists of models that describe the behavior of the system under normal and under fault conditions. Using this information, the diagnostic engine can deduce whether the system (or a certain part of it) is functioning correctly (in which case its behavior should correspond with the model) or whether its behavior deviates from the modeled normal behavior. In the latter case we speak of an error.

As derived from the research conclusions in chapter 4 we have chosen not to make models that fully describe the behavior and structure of the system under diagnosis. Instead, we have chosen to model only certain selected behavior, where we will try to capture a large amount of errors with only a moderate amount of modeling information. Therefore, the diagnosis information should be supplied by hand. In this way, human experts can use their knowledge and experience to create “smart” models that achieve the goal of finding a large part of errors with moderate modeling effort. This last decision also means that we are not looking into ways of automated creation of models.

Diagnostic system
The diagnostic system should be able to combine the behavioral information of the system under diagnosis and the diagnosis information and derive a conclusion. As stated in the requirements, the diagnostic system should do this with minimal user input. Since the research conclusions suggest that we should gather the observable information in a defined period of time, user input is still needed to define the starting point and the end point of the diagnosis. Since the diagnostic method consists of measuring signals, it means that the user also has to give the right input to the system. The right input here is the input that produces, under normal condition, the right flow of signals that is also modeled. This last step could be automated (the diagnostic software could simulate this input), but this has two drawbacks. First the hardware up to the point where the diagnostic system simulates the input is not tested. Secondly, automatic movements are not allowed due to safety restrictions (they still require a “dead-mans” control key to be pressed).

5.2 Method description

When looking back at paragraph 5.1 it can be seen that designing a diagnostic method involves defining the following three entities:

- What **behavioral information** is captured from the system
- What **diagnosis information** is needed
- How the **diagnostic engine** uses this information to create a diagnosis

The relationship between these three entities is already shown in Figure 5.1. We will now design these three entities in the following paragraphs. Note that a lot of terms are used in this chapter. These are explained in the list of terms that is included before the index.

5.2.1 Diagnosis information

The entity from Figure 5.1 that we will define first is the diagnosis infrastructure. The diagnosis infrastructure provides the diagnosis information that is used by the diagnostic engine together with the behavioral information to create a diagnosis. All the diagnosis information that is needed will be described in this paragraph.

In the method that we are designing, the diagnosis information is made up of the following three parts:

- Diagnostic models
- System information
- Diagnostic rules

We will define all three parts in the following paragraphs.

5.2.1.1 Diagnostic models

The diagnosis information contains information that is used by the diagnostic engine to create a diagnosis. The research conclusions suggest that a form of loose modeling is used to avoid having to make models that describe the full behavior of (a certain part of) the system. We propose to make models that directly link a (combination of) observation points to a conclusion. This conclusion will then be about the healthiness of components.

We will first give an example of such a link between an observation point and the conclusion. The input modules of the system are connected via a CAN network to the controlling software. If the controlling software receives a message from this CAN network then the conclusion can be drawn that the CAN network is functioning correctly. This link can be described as is shown in Figure 5.2.

“Observation point CAN_message_received is signaled”

$\Rightarrow$

“The CAN network is functioning correctly”

Figure 5.2 Example of informal model rule

Here we use the implication symbol from propositional logic to indicate that the conclusion part *“The CAN network is functioning correctly”* holds if the condition part “*Observation point CAN_message_received is signaled*” holds.

In true model-based diagnosis, this reasoning is done by a diagnostic engine [4][10][13], but as already said in the research conclusions we have chosen to create “smart models” that do not describe the full structure and/or behavior, and therefore these conclusions cannot be made by the diagnostic engine and must be incorporated into the diagnosis information. In this paragraph we will describe how these conclusions are incorporated into diagnostic models.

A diagnostic engine must make conclusions about the healthiness of components with the help of diagnostic models. More precise this means that the models must describe what influence a combination of signaled observation points have on the conclusion about the healthiness of a component. An example of this was already given in Figure 5.2. From now on we will call such a conclusion a diagnostic model rule (or simply model rule). We can now define models (or diagnostic models) as a collection of model rules. But first, the precise format of these model rules will be defined.

Model rules

Model rules consist of an observation part and a conclusion part. Both will be precisely described. We will start by investigating what is needed on both parts.

The observation part consists of an observation that is done in the system. In the previous example a single observation is used in the observation part, but we propose allowing combinations of observations to be used here. This might allow more advanced conclusions to be made (an example will follow later).

The other part that needs to be designed is how the conclusion part is shaped. Restricting to the two conclusions that a component is either functioning correctly (healthy) or is broken is not enough. There are situations where an observation could lead to the conclusion that any of a number of components *could* be broken. Therefore more gradations are necessary. An option could be to use a continuous domain, i.e. the domain of real numbers and assign each component a “healthiness score” between 0.0 (definitely broken) and 1.0 (definitely healthy). The downside of doing this is that the model rules in the models get more complex, since for every rule it needs to be described how the healthiness score of that component is affected.

Therefore, we have chosen to create a gradation with the minimal number of options that are necessary. These gradations are:

- Definitely healthy
- Unknown
- Possibly broken
- Definitely broken

We will now explain why we need at least these four gradations. Initially, before the diagnosis, nothing is known about the healthiness of the components. We need a gradation to represent this and use the gradation “Unknown” for those components. If no further conclusions are drawn about these components then the state of that component will remain “Unknown”.

Some observations might lead to the conclusion that a component is definitely working (see the example earlier). In this case these components need to be removed from the list of possible fault candidates. We use the “Definitely healthy” gradation for that.

Other observations might lead to the conclusion that something is definitely broken. These parts have to be exchanged. The gradation “Definitely broken” is used for this.

The last gradation is “Possibly broken”. This gradation is used when an observation leads to the conclusion that any of a certain number of components could be broken, but that it is not clear which of those components is actually broken.

Taken the previous definitions into account, we can now give a more formal description of model rules. Model rules will be of the form:

$$\textit{Observations} \Rightarrow \textit{Healthy}(C_1) \wedge \textit{PossiblyBroken}(C_2) \wedge \textit{Broken}(C_3)$$

Where *Observations* is a logical condition consisting of observation points (that are *true* when they are signaled and *false* when they are not signaled) and the logical operators *and*, *or*, *not* and *equals*. C_1, C_2 and C_3 are sets of components where the components in C_1 are given the gradation to be definitely healthy, the components in set C_2 are possibly broken and the components in set C_3 are definitely broken.

Example model

Now that the diagnostic models have been defined, we will give an example of a diagnostic model to further illustrate their use. For this example we will introduce a simple system shown in Figure 5.3.

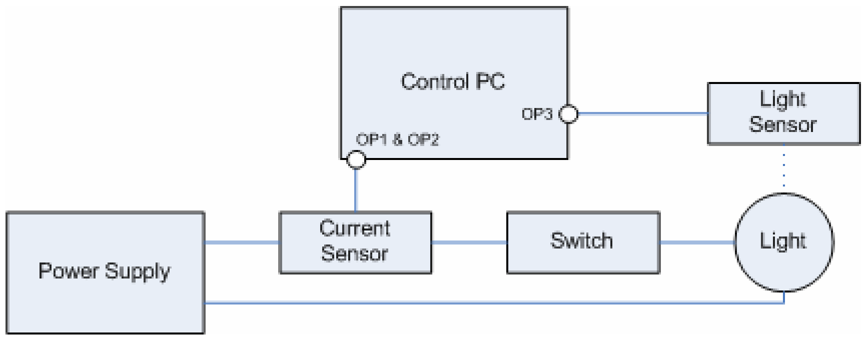


Figure 5.3 Example system

This system consists of a simple light circuit with a power supply, current sensor, switch and light. The Control PC can read the readings of the light and current sensor. The switch is controlled by an operator.

A model to diagnose this system could be called "switch_on_light" which should describe the behavior of the observation points when the switch is switched on.

The following assumptions are made on the model:

- The control PC works
- All connections work

- The current sensor works in such a way that it will always pass current, even when it is broken, plus the current sensor has intelligence such that it can differentiate between a too high current level and a normal current level

The model describes the following components:

- Power supply
- Current sensor
- Switch
- Light
- Light sensor

Furthermore, the following observation points (*OP1..OP3)* are inserted in the software that controls this system (Their locations are also shown in Figure 5.3):

Observation Point OP1

Name	`POWER_SENSOR_NORMAL_CURRENT`
Description	This observation points is signaled when the power sensor detects that the current going through it is within “normal” boundaries.

Observation Point OP2

Name	`POWER_SENSOR_TOOHIGH_CURRENT`
Description	This observation point is signaled when the power sensor detects that the current going through it is above the “maximal current” boundary.

Observation Point OP3

Name	`LIGHT_SENSOR_LIGHT_DETECTED`
Description	This observation point is signaled when the light sensor detects the light from the light bulb.

Using this information, the following five model rules (*R1..R5)* are defined for this model:

R1:

$$\texttt{POWER\_SENSOR\_NORMAL\_CURRENT} \Rightarrow \textit{Healthy}(\text{“Power supply”, “Switch”, “Current sensor”, “Light”})$$

R2:

$$\texttt{POWER\_SENSOR\_TOOHIGH\_CURRENT} \Rightarrow \textit{Healthy}(\text{“Current sensor”, “Switch”}) \wedge \textit{Broken}(\text{“Power supply”}) \wedge \textit{PossiblyBroken}(\text{“Light”})$$

R3:

$$\texttt{LIGHT\_SENSOR\_LIGHT\_DETECTED} \Rightarrow \textit{Healthy}(\text{“Power supply”, “Switch”, “Light Sensor”, “Light”})$$

R4:

```
LIGHT_SENSOR_LIGHT_DETECTED AND
NOT(POWER_SENSOR_NORMAL_CURRENT OR
POWER_SENSOR_TOOHIGH_CURRENT)
```

$\Rightarrow$

Healthy(“Power supply”, “Switch”, “Light Sensor”, “Light”) ∧ *Broken*(“Current sensor”)

R5:

```
NOT(POWER_SENSOR_NORMAL_CURRENT OR
POWER_SENSOR_TOOHIGH_CURRENT OR
LIGHT_SENSOR_LIGHT_DETECTED)
```

$\Rightarrow$

PossiblyBroken(“Power supply”, “Switch”, “Current sensor”, “Light Sensor”, “Light”)

The diagnostic model *M* for this example can then be constructed as the set of the model rules. This is described in the following formula:

$$M = \{R1, R2, R3, R4, R5\}$$

We will now also give a textual description of all five model rules:

The first rule defines that if normal current is detected this means that the power supply, the switch the current sensor and the light are working correctly. This is based on the domain knowledge that if the power supply is broken there can be no current flowing through the sensor. If either the switch or the light bulb is broken, the power circuit would be interrupted which also means that no current could flow through the sensor. Lastly, if the current sensor detects current then it must be working correctly.

The second rule defines that if too high current is detected that the current sensor and switch are working correctly, the power supply is broken and the light might also be broken (damaged by the high current).

The third rule indicates that if light is detected the by the sensor then this light must be emitted by the light bulb. This can only be the case if the power supply, switch, light and light sensor are working correctly.

The fourth rule is an example of a model rule that uses more than one observation point to draw more “advanced” conclusions. This rule captures the domain knowledge that if light is detected then the power supply, switch, light and light sensor are working correctly (we established this with the last rule). Therefore we know that there must be current flowing through the current sensor. If the current sensor does not detect any current then it must be broken.

The last rule indicates the case where no current and no light is detected. From the domain knowledge we know that any of the main components (power supply, switch, and light) could be broken, or that both sensors must be broken. In our method we can only represent this by giving all those components the gradation of “Possibly Broken”.

A priori failure probability

In the previous paragraph four gradations for the healthiness of components are given. A total ordering on the probability of a component being broken given one of the four gradations can be defined as:

“Definitely healthy” < “Unknown” < “Possibly broken” < “Definitely Broken”

This should be read as “Any component marked definitely broken has a higher probability of being broken than any component that is marked possibly broken”. In turn it also holds that “Any component marked possibly broken has a higher probability of being broken than any component that is marked unknown”.

Although this gives a basic ordering, it is not enough to give a list of components ranked on probability of the component being broken. Diagnosis result could be that a number of components are given the gradation “Possibly broken” without any further ranking between those components. As stated in the previous paragraph we have chosen to have only those four gradations to avoid the complex situation where each model rule should describe how a components healthiness score is affected by the observation. This means that another way is needed to rank the components that are given the same gradation on probability of being broken.

The Lydia project [20] combines model based diagnosis with statistical failure probabilities (a priori failure probabilities) to create a list of fault candidates (component or set of components) ranked from the candidate that has the highest probability of being broken to the candidate that has the lowest probability of being broken. After discussion with PMS employees it is decided to also use this method, because of many components it can be estimated what their a priori failure probability is. Therefore it is suggested as a good method to be able to further distinguish the individual components that have the same gradation, without adding a large amount of complexity to the model rules.

5.2.1.2 System information

In the research conclusions we suggested that the system should be divided into a number of modules, and that the diagnostic models are created for these modules. The way the system is divided into modules is also part of the diagnosis information and this will be described now.

The system must be divided into a number of modules. For this division we are looking for parts that have a clear function in the system and that have defined input and/or output signals. For instance, a power module has a clear function to deliver power to the various other modules of the system that need power. It also has a defined output signal, which is the current that it delivers to the various other modules. Another example is the input module which has as function to forward the requests that are inputted by a user to the control PC. It also has a defined output signal, namely the (movement) request signals.

A requirement that was defined in paragraph 3.3 that is closely related to this division into modules is requirement 3.3.1: The solution should be able to cope with different hardware configurations of the machine. The division into modules has indeed led to a number of modules that are either optional, or that are mutually exclusive (for instance 3 different table types of which only one can be connected to the system). When investigating how to represent this information a solution called “Feature Diagrams” [21] has been found.

An example of a feature diagram is shown in Figure 5.4.

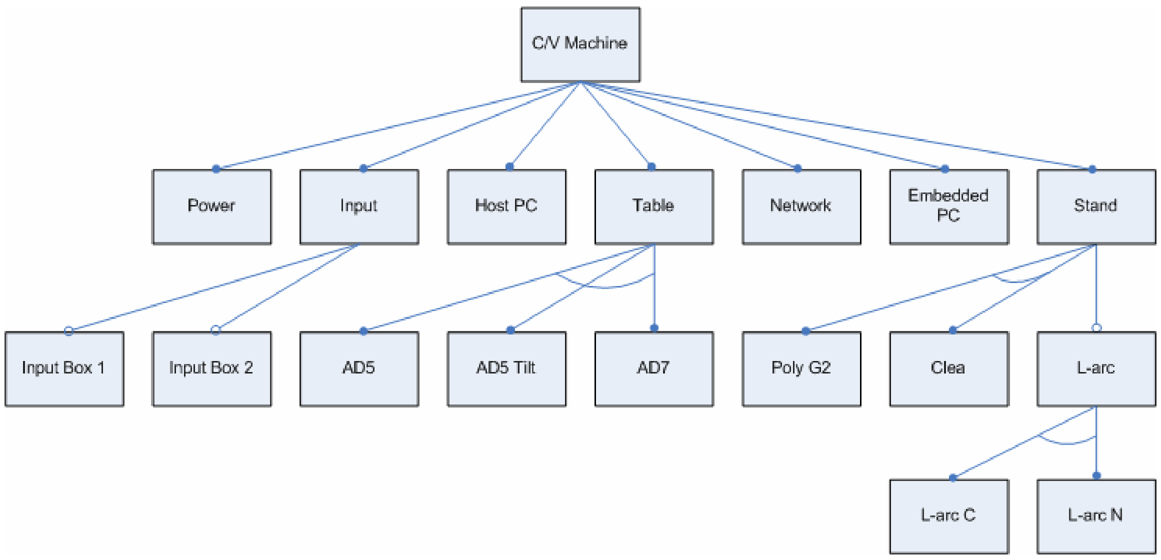


Figure 5.4 Feature Diagram of Cardio Vascular X-Ray system (partly)

This figure shows a part of a feature diagram for the C/V X-Ray System of PMS. The diagram is not complete; it only is used to present the method.

In this feature diagram, the complete system is represented by the root of the tree. The children of the root represent the different parts that the system is made of. These parts can be hardware modules or software modules, but also a combination of these. We will refer to these parts as modules. Modules with a solid dot are mandatory; modules with an open dot are optional. In the picture above, the L-arc module is an example of an optional module.

Each module can be further divided into smaller modules until, at the leaves of the tree, we have modules that have a clear function in the system and that have defined input and/or output signals. Furthermore these modules must be small enough such that it is possible to create diagnostic models for these modules (without the diagnostic models getting too complex).

In the diagram, the Input, Table and Stand modules are further divided. For the Table module there is a choice between 3 different tables. This choice is described by the arc that is drawn between the 3 options. The AD5, AD5 Tilt and AD7 modules are the leaves of the tree, which indicate that these modules have a clear function within the whole system. It is for these modules that the diagnostic models, discussed in paragraph 5.2.1.1, will be created.

5.2.1.3 Diagnostic Rules

After the system has been divided into modules, diagnostic rules must be created. Diagnostic rules describe which diagnostic models need to be checked to diagnose certain functionality of the system. However, there might be certain dependencies between the models of the different modules, as is depicted in Figure 4.5. These dependencies are the reason that the models must be checked in a certain order. Since these dependencies cannot be inferred from the feature diagram or the diagnostic models this means that the dependencies must be captured in the diagnosis information.

Diagnostic rules define the order in which the diagnostic models need to be checked. An example of such a chain is shown in Figure 5.5. The arrows in the figure represent the order in which the diagnostic models are checked. They do not represent information that is passed from model to model since every model is treated as a separate entity by the diagnostic engine. This means that when checking a model, no information of models that are checked earlier is used.

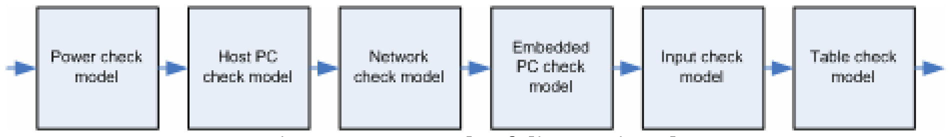


Figure 5.5 Example of diagnostic rule

The diagnostic engine will first perform diagnostics on the “power check model” that could be defined in the “power module”. If the diagnosis of this model is ok, then the next model is checked by the diagnostic engine. The check whether a diagnostic model was ok or not is done by the diagnostic engine. This will be discussed in paragraph 5.2.3. The sequence of actions performed to check the diagnostic rule shown in Figure 5.5 is schematically depicted by the diagram in Figure 5.6.

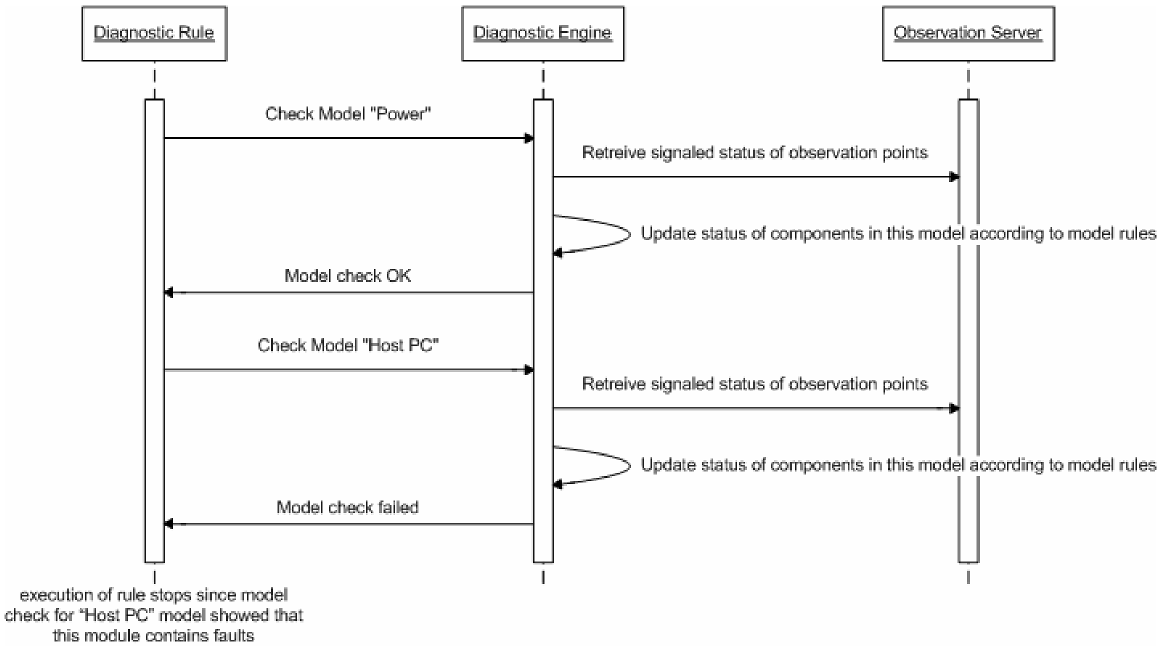

Diagnostic Rule
Diagnostic Engine
Observation Server
Check Model "Power"
Retreive signaled status of observation points
Update status of components in this model according to model rules
Model check OK
Check Model "Host PC"
Retreive signaled status of observation points
Update status of components in this model according to model rules
Model check failed
execution of rule stops since model check for "Host PC" model showed that this module contains faults

Figure 5.6 Schematic overview of the checking of a diagnostic rule

The chain that is shown in Figure 5.5 is completely linear. However, there might also be situation where not all models are linearly related. Two examples of chains in which the models are not linearly related are shown in Figure 5.7 and Figure 5.8.

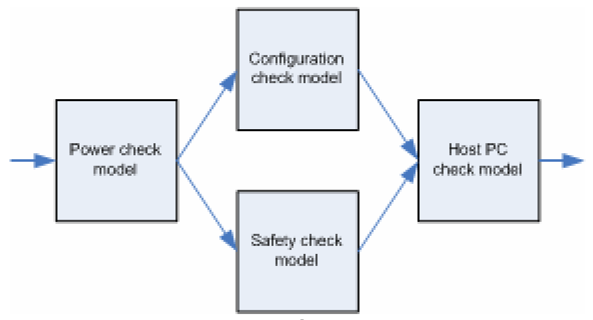


Figure 5.7 Example of dependencies in rule

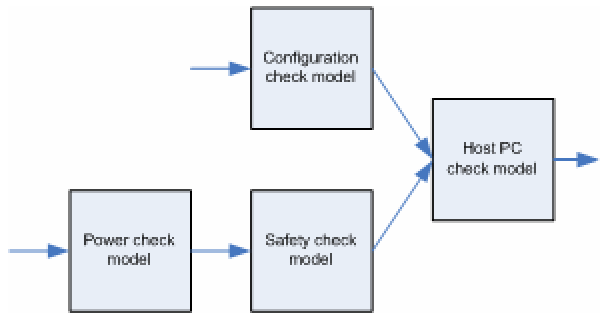


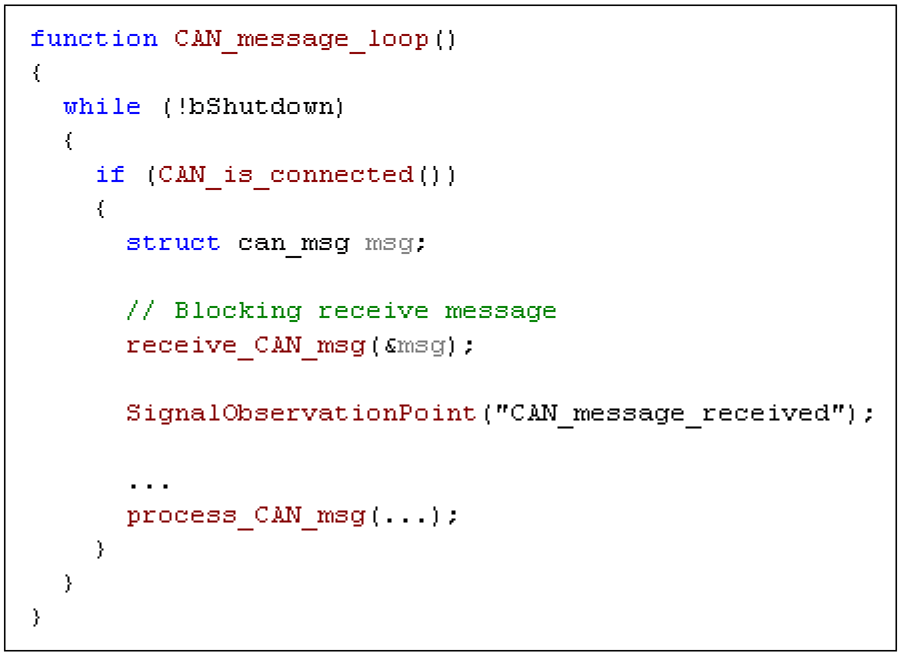
Figure 5.8 Another example of dependencies in rule

This means that the diagnostic engine must also be able to deal with these kinds of dependency relations. A possibility is that the diagnostic engine transforms these chains into a linear chain. This is only possible if the chain does not contain circular dependencies, so circular dependencies should be avoided.

5.2.2 Behavioral information

Paragraph 5.2.1 describes how conclusions are drawn given certain observations. In this paragraph we will describe how these observations are gathered from the system that is under diagnosis.

As described in paragraph 5.1.2 we have chosen to add our own behavioral information to the software that is under diagnosis. The research conclusions in paragraph 4.5.2 indicate that behavior should be captured as signals that indicate whether the control flow has passed a certain point in the code in a defined period of time. An example of this is shown in Figure 5.9.



```
function CAN_message_loop()
{
  while (!bShutdown)
  {
    if (CAN_is_connected())
    {
      struct can_msg msg;

      // Blocking receive message
      receive_CAN_msg(&msg);

      SignalObservationPoint("CAN_message_received");

      ...
      process_CAN_msg(...);
    }
  }
}
```

Figure 5.9 Pseudo code example of signaling of observation points

The example shown is a pseudo-code example; it is used for indication. At the point in the code that reads `SignalObservationPoint("CAN_message_received");` a method should be inserted that will generate a signal such that the diagnostic engine knows that the control flow has passed there. These points in the code will be called "observation points" from now on.

The behavioral information that is gathered this way consists of a number of observation points in the code of which the diagnostic engine can check whether they were signaled in a certain interval, i.e. whether the control flow has passed there in the period of the diagnosis. The diagnostic engine can combine this behavioral information with the diagnosis information to draw conclusions about the state of (parts of) the system. For instance, in the example above the conclusion might be drawn that the CAN network is functioning correctly, since a message over the CAN network has been received.

In paragraph 6.4.5 we will design observation points at source code level. For now we will suffice with the abstract idea that an observation point becomes TRUE whenever the statement that contains the observation point is executed by the processor of a computer. If the statement containing the observation point is not executed it will remain FALSE.

5.2.3 Diagnostic Engine

The last entity from Figure 5.1 that needs to be defined is the diagnostic engine. The diagnostic engine is responsible for processing the behavioral information and the diagnosis information and deriving a diagnosis.

The task of the diagnostic engine is to check the models and derive the healthiness status of the components using observations. This means that the diagnostic engine should be able to retrieve the observation information from the target software.

To perform system wide diagnosis, the diagnostic engine will interpret a diagnostic rule and check the models in the order that is described by the diagnostic rules. Since every model is defined for one module, the healthiness of the modules can be inferred by the diagnostic engine. This is done by checking whether all components within that module have been identified as healthy (or by checking whether no component has been identified as possibly broken).

A second method for inferring healthiness of modules is to specify an “entry” and “exit” observation for every model. These observations can then be used to detect whether a (combination of) signal(s) has entered the module and whether a (combination of) signal(s) has exited the module. If the entry observation was not signaled, it can be concluded that the control flow never reached that module (and that the problem thus must be earlier in the chain). If the exit observation was not signaled, it can be concluded that the control flow never leaves that module and that the problem must be in that module. Note that these observation points must be optional, since they cannot be defined for every module or model. For instance, the power module does not have an input signal, it just produces output and the table module only has input; it has no defined output signals.

When a module is identified as not working correctly, the diagnostic engine must give a list of components within that module, ranked by the probability of being broken. For this list the diagnostic engine uses a ranking function that takes the following factors into account:

- Whether a component has been marked as definitely healthy
- Whether a component has been marked as possibly broken
- Whether a component has been marked as definitely broken

- A priori failure probability of component

From these factors the diagnostic engine will construct a list with candidates that are possibly broken, starting with the component that is the most likely to be broken to the component that is the least likely to be broken. The precise mathematical definition of the ranking function is given in paragraph 6.4.2.

A side note with the gradations is that if a component is marked as both definitely healthy and possibly broken, it is considered definitely healthy by the definition of definitely healthy. When a component is marked as both possibly broken and definitely broken it is considered broken. Lastly, if a component is marked both definitely healthy and definitely broken it is considered to be a modeling error, and an error message should be produced.

The factors described above are based purely on defining which component has the highest probability of being broken. However, in practice it might also be useful to consider other factors. Consider the case where the result of the diagnosis is that the power supply has the highest probability of being broken and that a fuse has the second highest probability of being broken. While probabilities indicate that the power supply is the best candidate to be replaced, it is in practice often much more cost efficient to exchange the fuse first, and only if that does not solve the problem replace the power supply. The reason for this is that a fuse is far less expensive than a complete power supply unit and most of the time also takes less time to replace.

Therefore two extra factors are taking into account. These are:

- The time it takes to exchange a component
- The costs of a component

Both of these factors are known within PMS for the defined FRU's.

Besides the main task of deriving the healthiness of modules and components, the diagnostic engine is also responsible for presenting the diagnosis to the user and for dealing with the variability of the system. Both these subjects will be discussed now.

5.2.3.1 Diagnostic model overloading

The diagram shown in Figure 5.4 graphically represents the Cardio/Vascular X-Ray system with its optional modules (such as the L-Arc module) and its modules of which different versions are available (such as the Table module). But the diagnostic engine should also be able to deal with this, preferably with as little extra effort as possible compared to when the system had no optional / choice modules.

For this we design a feature we call diagnostic model overloading. We will illustrate this feature with an example. Suppose we have the diagnostic rule as shown in Figure 5.10.

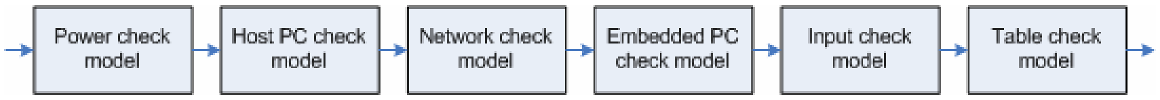


Figure 5.10 Example diagnostic rule

The whole diagnostic rule is defined for checking the table movements, i.e. the case where the table movements do not work or one of the table movements does not work. It consists of six diagnostic models. Each diagnostic model is linked to a module from Figure 5.4 and is responsible for checking that module.
Now as can be seen in Figure 5.4, a system can have the AD5 table, the AD5 tilt table or the AD7 table. The AD7 table is completely different in both hardware and software and the AD5 tilt table has more hardware than the AD5 table. This means that different diagnostic models must be created for the different tables.

Now by our definition of diagnostic rules, also three different diagnostic rules have to be created. When taking the different possibilities for the input module into account, even more diagnostic rules have to be created. But we would like to have one general rule for checking the table movements, independent of the underlying hardware.

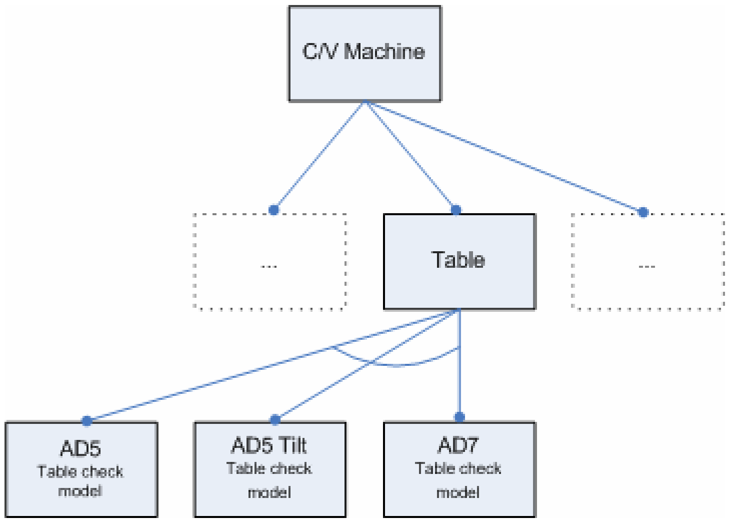


Figure 5.11 Diagnostic models created for the modules

In Figure 5.11 a part of the feature diagram is shown again. For each of the three tables a diagnostic model has been created. Now instead of making three different diagnostic rules, one with each of those models, we define one diagnostic rule in which we define that we want to execute the “Table check model” of the “Table” module (instead of the “Table check model” of either the “AD5”, “AD5 Tilt” or “AD7” module).

Of course, the “Table” module itself does not have a model called “Table check model”, so it is the responsibility of the diagnostic engine to execute the right model depending on the configuration of the system (if the system is configured with an “AD7” table, it should execute the “Table check model” model of the AD7 table).

5.2.3.2 Diagnosis presentation

From interviews with field service engineers it has become clear that the diagnosis presentation is a very important aspect. The presentation of the diagnosis should be clear and understandable for the field service engineer. If this is not the case, then the diagnostic system is likely to be ignored by the field service engineers.

For now we propose to let the diagnostic engine give a graphical representation of the modules in the system and let colors describe the healthiness of each module. Besides this representation, also a list of components in each module should be given and this list should indicate for each component whether it was marked as healthy, broken or possibly broken. This list should also rank the components based on which component is the best candidate for replacement.

Besides the given markers, the presentation of the diagnosis results will be considered out of scope for this project.

6 Design of prototype diagnostic solution

In this chapter the design of the prototype of the diagnostic solution will be described. This prototype implements the method that is described in chapter 5. An overview of the architectural design will be given and the designs of the different blocks will be shown. Some important design decisions regarding problems that were encountered will also be explained and motivated.

6.1 Architecture overview

Figure 6.1 gives an overview of the system that is to be designed. All grayed components are part of the diagnostic solution.

The software that controls the system is already available (and depicted by the white boxes). However, observable information needs to be collected from this software. This means extra additions are needed (depicted by the observation server blocks).

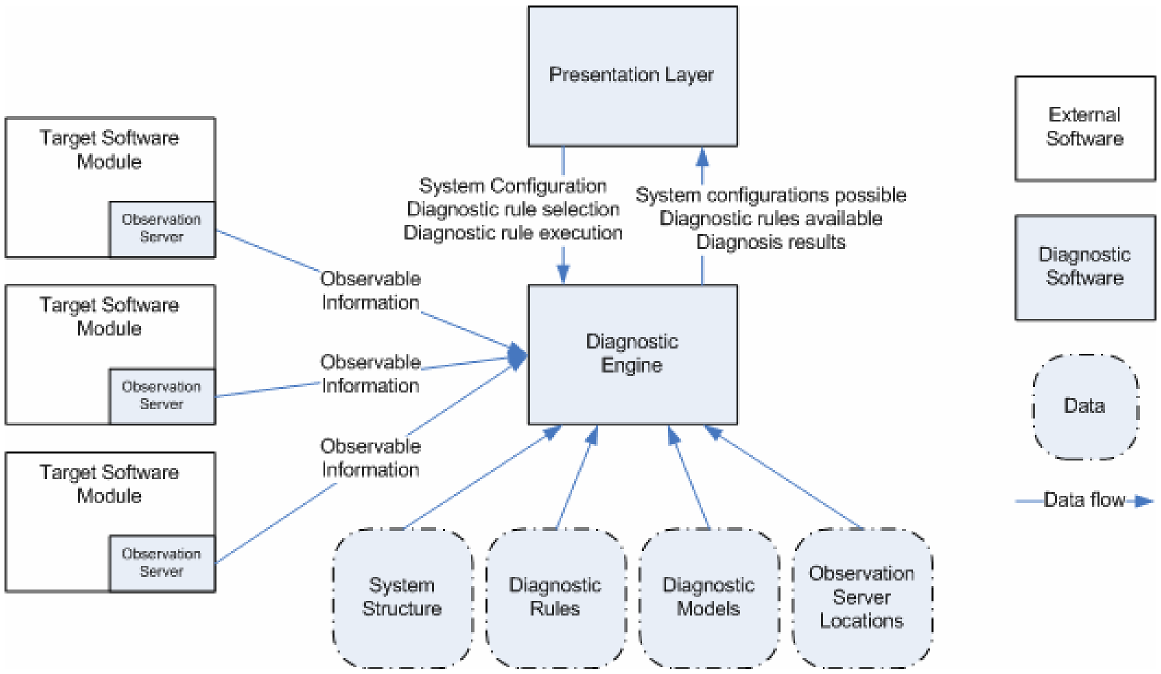


Figure 6.1 Architectural overview of the Diagnostic System

As can be seen from the diagram, the main part of the diagnostic system is the Diagnostic Engine. This module is responsible for executing a diagnostic rule, collecting the observable information from the target software and deriving a diagnosis. It collects the observable information by contacting the observation servers that are placed in the target software modules. Using the observable information and interpreting the data of the system structure, diagnostic models and diagnostic rules the diagnostic engine can derive a diagnosis.

The diagnostic engine is aware of the diagnostic capabilities of the system via the system structure description and the system configuration that it receives as input. It can then receive the command to run a test (execute a diagnostic rule) from the presentation layer and can execute this test by checking all diagnostic models involved in that rule using the observable information from the Target Software. The result of the diagnosis is then sent back to the presentation layer.

In the following paragraphs the designs of the two main components, the Diagnostic Engine, and the Observation Server, will be shortly described. The presentation layer is out of scope and will not be designed in this project.

6.2 Diagnostic Engine Design

The diagnostic engine is the heart of the diagnostic system. Its task is to collect behavioral information from the system under diagnosis, execute diagnostic rules, interpret the diagnostic models and finally derive a diagnosis using the available information. For the design of the diagnostic engine UML is chosen because this method is good for describing designs and it is the standard within PMS.

First the UML static structure design will be shown, then the dynamic behavior is designed using UML sequence diagrams. The classes in the diagrams below only show a limited number of functions to give an idea of the responsibilities for that class.

6.2.1 Top level design

To decrease the complexity of the design, it has been split into a top level design consisting of the main class and four groups of classes. For each group of classes a more detailed design is shown after the top level design. Figure 6.2 shows the top level design.

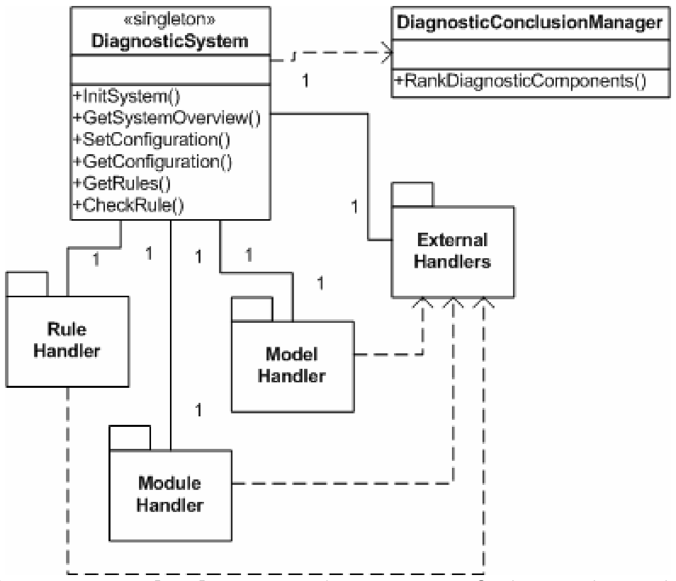


Figure 6.2 Top level UML static structure of Diagnostic Engine

The DiagnosticSystem class is the external interface of the system. It is responsible for the creation of all handler classes and is the entry point for the presentation layer. This class uses the other handler classes to retrieve a list of diagnostic rules and also to execute a rule. The results are then passed to the DiagnosticConclusionManager class which creates a ranked list of components from the information. This ranked list of components is then passed to the presentation layer.

The handlers are groups of classes that have their own function in the complete design. To limit complexity of the design, the individual classes are shown in separate diagrams which are discussed in the following four paragraphs.

6.2.2 Rule Handler design

The rule handler group of classes is responsible for handling the rules. The design is shown in Figure 6.3. The DiagnosticRuleManager class reads all diagnostic rules from file and filters the rules that are available on the active configuration. It is also responsible for checking a rule. For both these tasks the DiagnosticRuleManager class communicates with the Module Handler group of classes. The DiagnosticRule class represents one rule and stores all the data that is needed for one rule. A diagnostic rule is made up of several diagnostic models, and every model is linked to one step in a diagnostic rule. Therefore a DiagnosticRule object has at least one DiagnosticRuleStep object in which data is stored about the model that needs to be checked in that step and to which module that model belongs.

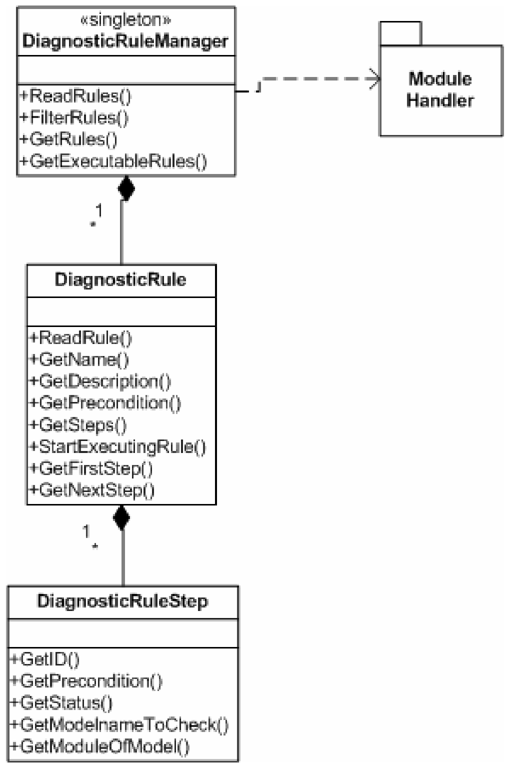


Figure 6.3 Design of Rule Handler group of classes

6.2.3 Module Handler design

The module handler group of classes is responsible for handling the modules. Figure 6.4 shows the design. The DiagnosticModuleManager class reads the system structure file and creates an internal data structure of all the modules. This class is also responsible for storing the active configuration. The system structure is stored in a tree form, which is done by the classes TreeNode and DiagnosticModuleNode. The TreeNode class implements the behavior of the tree (such as adding children), while the DiagnosticModuleNode class is responsible for adding all the specific information from a diagnostic module to a particular node. This specific information is stored in the DiagnosticModule class.

Classes from the Rule Handler group communicate with the DiagnosticModuleManager class, which in turn communicates with classes from the Model Handler group to retrieve the diagnostic models that are linked to a certain diagnostic module.

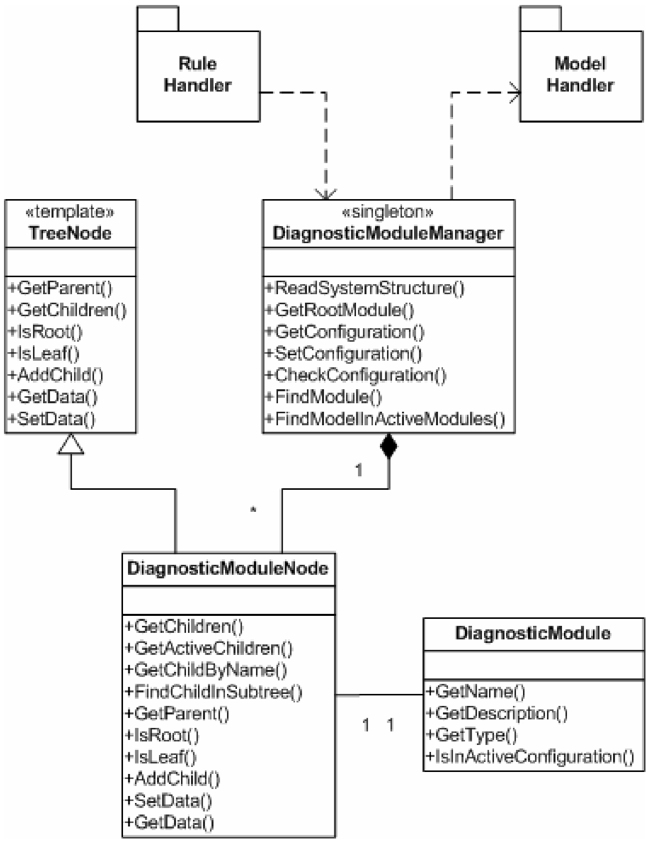


Figure 6.4 Design of Module Handler group of classes

6.2.4 Model Handler design

The model handler group of classes is responsible for handling the models. Its design is shown in Figure 6.5. The DiagnosticModelManager class reads all the models from the files.

The model data is stored in instances of the DiagnosticModel class. A diagnostic model consists of observation points, components and model rules and all this information is stored in instances of the ObservationPoint, DiagnosticComponent en ModelRule classes respectively.

The ObservationManager class is responsible for contacting the DiagnosticObservationServer interfaces and retrieves the observation point data from the software that is under diagnosis.

Finally, the DiagnosticModelChecker class checks the model rules of a diagnostic model and for that task it uses the BooleanExpressionEvaluator class to evaluate the condition part of the model rules.

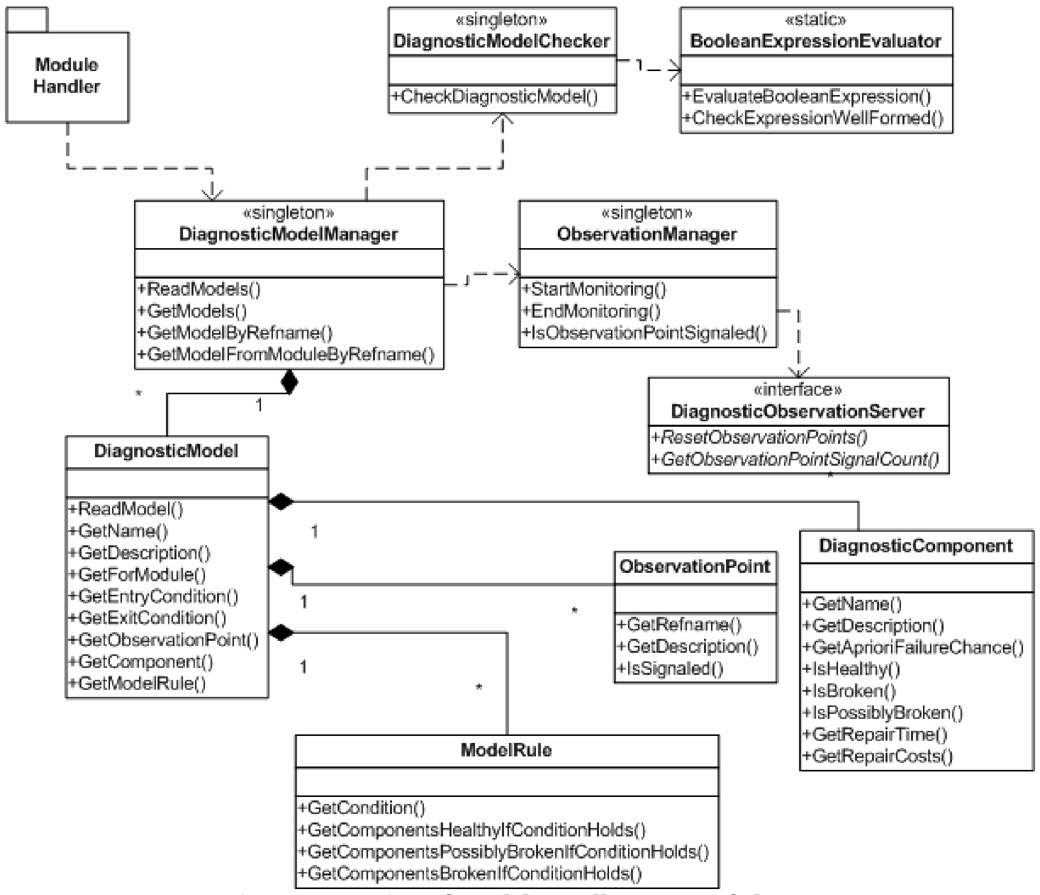


Figure 6.5 Design of Model Handler group of classes

6.2.5 External Handlers

This group contains “helper” classes that are not interrelated, but are used by classes in the other groups. The Logging class is used by the other classes for logging purposes. The goal of the logging is to provide information that can be used to debug the prototype diagnostic solution. The system structure file, model files and rule files are all stored in XML format. An XML Parser is used to read and parse these files.

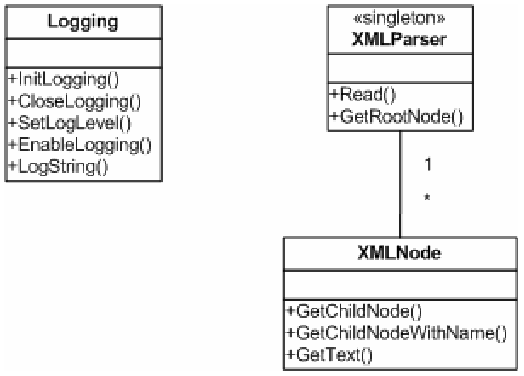


Figure 6.6 Design of External Handlers

6.2.6 Dynamic behavior

In this paragraph some of the dynamic behavior of the diagnostic engine will be described. This dynamic behavior will be described using UML Sequence diagrams. Here the same holds as with the static structure design, UML is chosen because this method is good for describing designs and it is the standard within PMS.

6.2.6.1 Initialization

During the initialization process a number of initializations need to be done. First the logging is initialized and then several files are read. This files that are read contain the system structure, the diagnostic rules and the diagnostic models. This behavior is described in Figure 6.7.

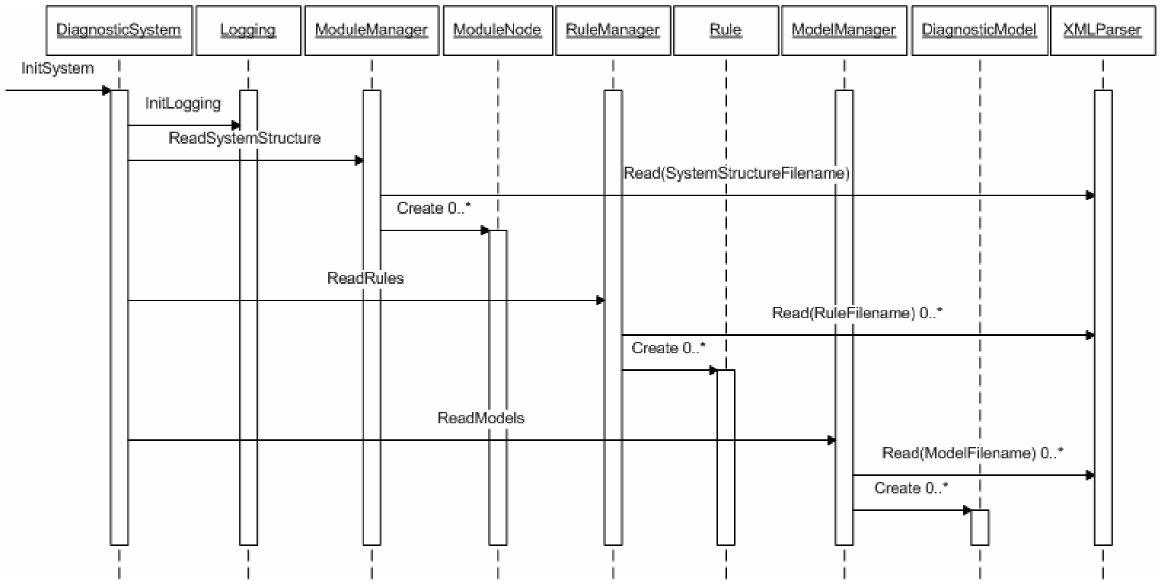


Figure 6.7 Initialization process of the diagnostic engine

6.2.6.2 First part of the checking of a diagnostic rule (PrepareExecuteRule)

The most important function of the diagnostic engine is to diagnose the target. For diagnosing the target a diagnostic rule is selected by the user (for instance the check table movements rule) and this rule is then checked by the diagnostic engine (executed).

The checking of a diagnostic rule is done in two parts. In the first part, the system is prepared, i.e. the right rule is selected, the configuration is checked, it is checked if the rule can be executed in the given configuration and the observation manager starts monitoring the observations points. As the final part of the preparation the user is told to perform an action. This action (for example press button X) is supposed to cause certain behavior in the system and this behavior is then captured by the observation points. The checking of the models and model rules is done in part two.

The UML sequence diagram for the first part of the checking of a diagnostic rule is shown in Figure 6.8.

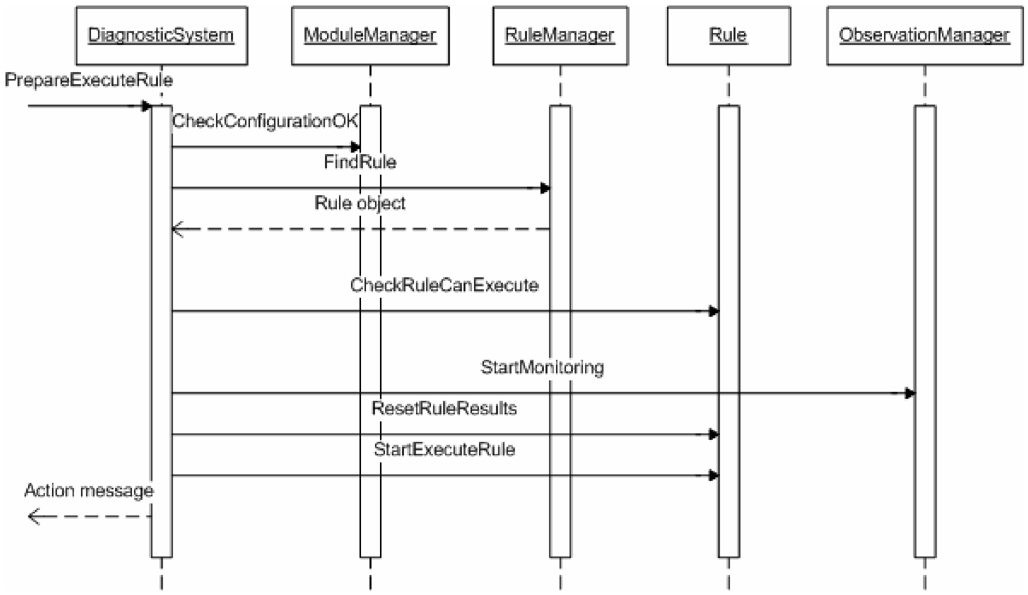


Figure 6.8 The first part of the checking (execution) of a diagnostic rule

6.2.6.3 Second part of the checking of a diagnostic rule (FinishExecuteRule)

In the second part of the checking of a diagnostic rule the complete diagnostic rule is checked step by step. In each step the diagnostic model corresponding to that step is checked, which means that all the model rules in that model must be checked. In turn this means that for each model rule the condition part is evaluated (where a signaled observation point is TRUE and an observation point that was not signaled is FALSE) and if the condition holds then the components are marked Healthy, PossiblyBroken or Broken, depending on what the rule indicates.

When this is done for all model rules for all models for all steps then the results are ranked using the DiagnosticConclusionManager. These results can then be presented to the user via the presentation layer (which is out of scope for this project).

The UML sequence diagram for the second part of the checking of a diagnostic rule is shown in Figure 6.9.

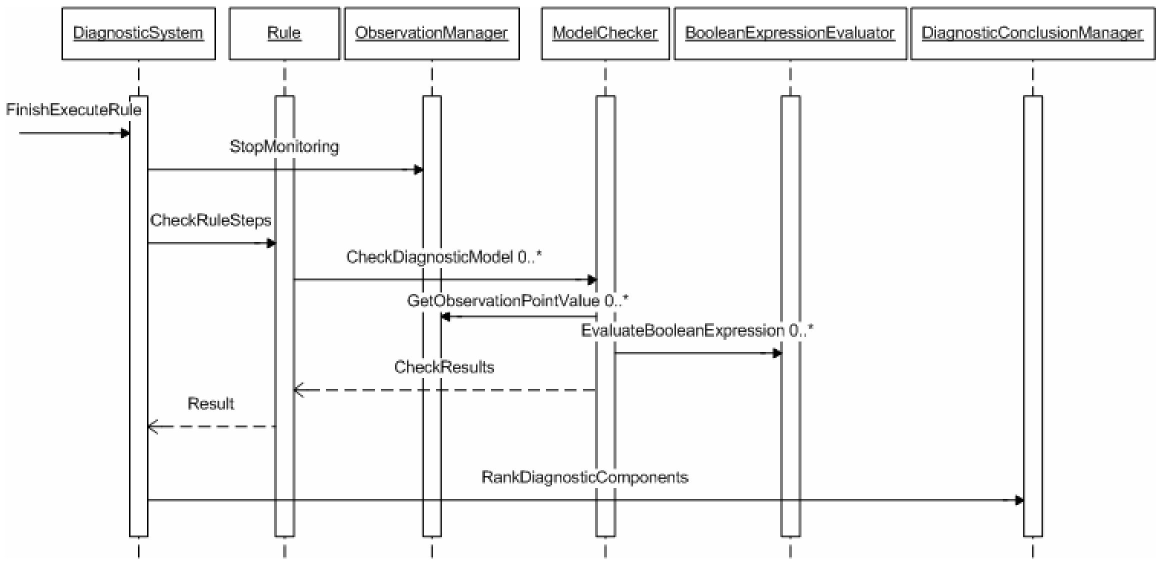

DiagnosticSystem
Rule
ObservationManager
ModelChecker
BooleanExpressionEvaluator
DiagnosticConclusionManager
FinishExecuteRule
StopMonitoring
CheckRuleSteps
CheckDiagnosticModel 0..*
GetObservationPointValue 0..*
EvaluateBooleanExpression 0..*
CheckResults
Result
RankDiagnosticComponents

Figure 6.9 The second part of the checking (execution) of a diagnostic rule

6.3 Observation Server Design

In this paragraph the static structure and dynamic behavior of the observation server will be described.

6.3.1 Static structure

The observation servers are integrated into the target software that is under diagnosis. Its UML static structure design is shown in Figure 6.10.

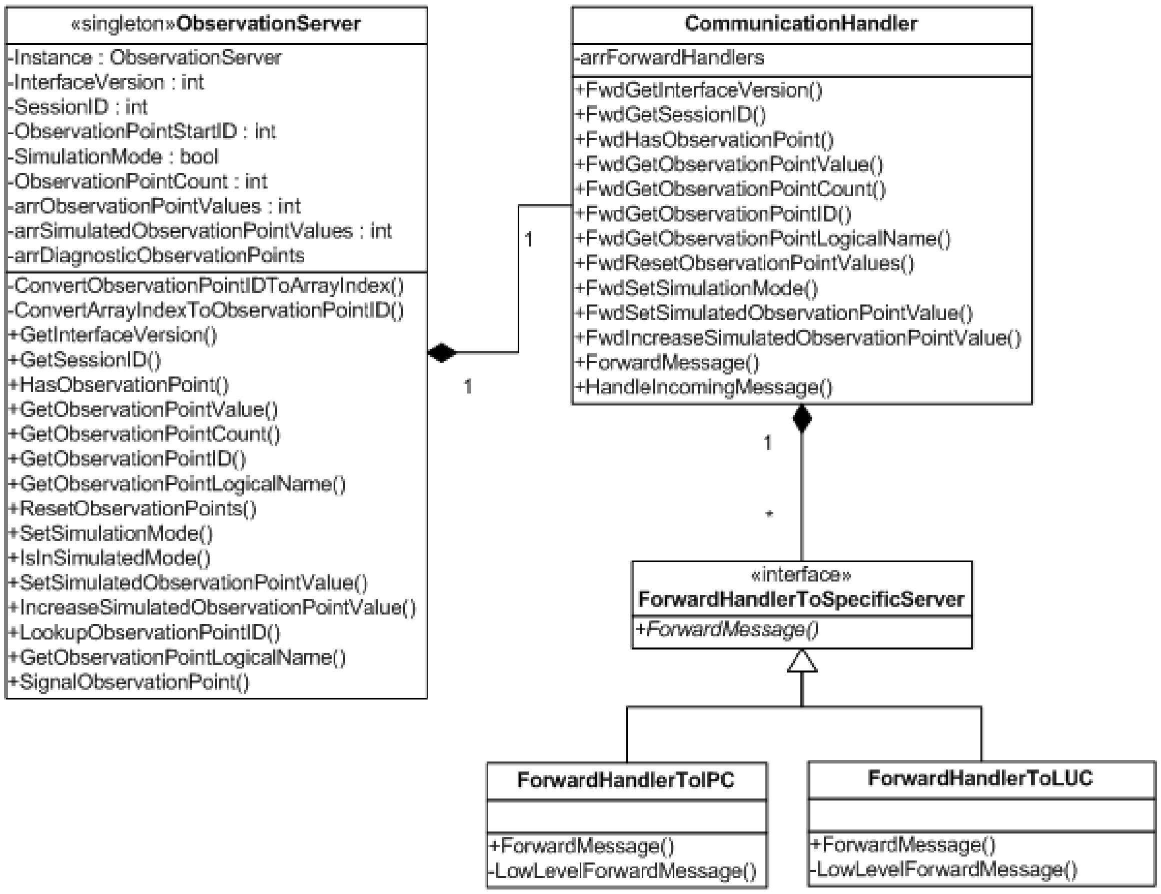

«singleton»ObservationServer
-Instance : ObservationServer
-InterfaceVersion : int
-SessionID : int
-ObservationPointStartID : int
-SimulationMode : bool
-ObservationPointCount : int
-arrObservationPointValues : int
-arrSimulatedObservationPointValues : int
-arrDiagnosticObservationPoints
-ConvertObservationPointIDToArrayIndex()
-ConvertArrayIndexToObservationPointID()
+GetInterfaceVersion()
+GetSessionID()
+HasObservationPoint()
+GetObservationPointValue()
+GetObservationPointCount()
+GetObservationPointID()
+GetObservationPointLogicalName()
+ResetObservationPoints()
+SetSimulationMode()
+IsInSimulatedMode()
+SetSimulatedObservationPointValue()
+IncreaseSimulatedObservationPointValue()
+LookupObservationPointID()
+GetObservationPointLogicalName()
+SignalObservationPoint()
1
1
CommunicationHandler
-arrForwardHandlers
+FwdGetInterfaceVersion()
+FwdGetSessionID()
+FwdHasObservationPoint()
+FwdGetObservationPointValue()
+FwdGetObservationPointCount()
+FwdGetObservationPointID()
+FwdGetObservationPointLogicalName()
+FwdResetObservationPointValues()
+FwdSetSimulationMode()
+FwdSetSimulatedObservationPointValue()
+FwdIncreaseSimulatedObservationPointValue()
+ForwardMessage()
+HandleIncomingMessage()
1
*
«interface»
ForwardHandlerToSpecificServer
+ForwardMessage()
ForwardHandlerToIPC
+ForwardMessage()
-LowLevelForwardMessage()
ForwardHandlerToLUC
+ForwardMessage()
-LowLevelForwardMessage()

Figure 6.10 Design of Observation Server

The observation server manages the observation points. This means that all observation points in the target software “register themselves” at the ObservationServer object by providing a “logical name”. Then each observation point is given an ID and an entry in the array that stores the values of all observation points is assigned. Every time the control flow passes the observation point, the entry in the array is increased by one. This way it can be checked if the observation point is signaled in a certain period of time by comparing the value at the beginning of the period with the value at then end of the period. If those two values differ, then the observation point was signaled.

The ObservationServer class provides functions to search for observation points by logical name, convert the logical name into an ID or an ID into a logical name and for retrieving the value of an observation point. Furthermore it provides simulation functions where behavior can be simulated without the target software actually running.

In paragraph 6.4.4 it will be precisely described why there is a need to have multiple observation servers. For now we will assume that we have multiple observation servers divided over a number of processes which run on a number of different hardware platforms. We will also assume that there is a need for communication between these observation servers (why this is will also be discussed in paragraph 6.4.4).

The CommunicationHandler class arranges this communication. The CommunicationHandler class is responsible for receiving requests from other ObservationServers (running in different processes, possibly on different hardware) and it also takes care of the forwarding of requests to other ObservationServers.

For instance, there is an embedded PC called the Geo IPC which is connected to the Host PC (which runs the diagnostic software). This Geo IPC is connected to the Host PC via Ethernet. Observations need to be gathered at the Geo IPC too and therefore it also has an ObservationServer in its process. To communicate with this observation server a specific ForwardHandlerToIPC class is created implementing the generic interface ForwardHandlerToSpecificServer. The specific implementation takes care of the communication via Ethernet.

In turn, there are embedded real time platforms called LUC’s connected to the Geo IPC. Here observations need to be gathered too, so they also have an ObservationServer in their process. These LUC’s are connected via CAN so a specific ForwardHandlerToLUC class is created that implements the communication via CAN.

Using the generic ForwardHandlerToSpecificServer interface every instance of ObservationServer can forward the request to other ObservationServers without needing knowledge of the underlying communication channel.

6.3.2 Dynamic behavior

In this paragraph the dynamic behavior will be described. As in paragraph 6.2.6, UML Sequence diagrams will be used for this.

6.3.2.1 Registering and signaling of observation point at the observation server

As is discussed in paragraph 6.3.1, the observation points in the code of the target software need to register themselves with the observation server. To do this the code constructs a logical name in the initialization phase and sends this name to the observation server. The observation server then links a unique ID to this name and sends this back. This unique ID is stored locally.

Whenever the control flow then passes the observation point in the code, the observation point needs to be signaled at the observation server. For this the unique ID (that is stored locally) is used and the result is that the value of that observation point at the observation server is increased by one. This process is shown in Figure 6.11.

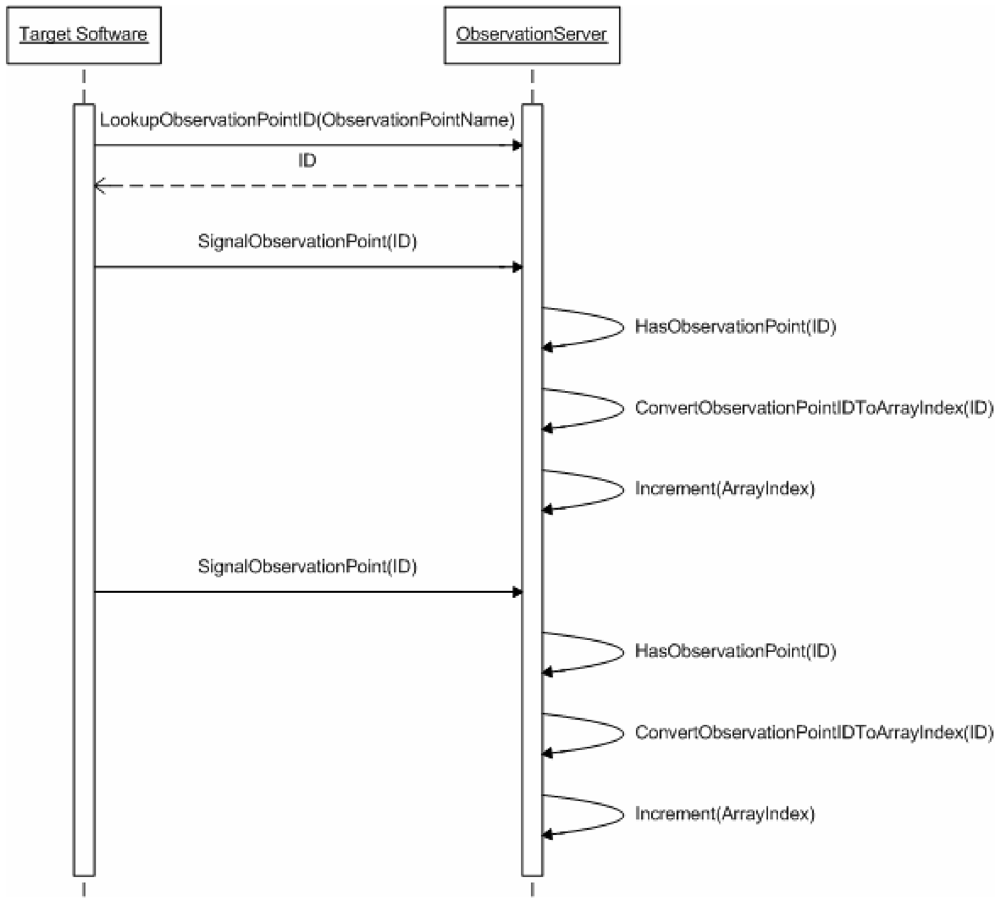

Target Software
ObservationServer
LookupObservationPointID(ObservationPointName)
ID
SignalObservationPoint(ID)
HasObservationPoint(ID)
ConvertObservationPointIDToArrayIndex(ID)
Increment(ArrayIndex)
SignalObservationPoint(ID)
HasObservationPoint(ID)
ConvertObservationPointIDToArrayIndex(ID)
Increment(ArrayIndex)

Figure 6.11 Registering and signaling of observation point at the observation server

6.3.2.2 Forwarding of requests to other observation servers

Another feature that was discussed in paragraph 6.3.1 is the ability of an observation server to forward requests to other observation servers. We have chosen to illustrate this behavior for the GetObservationPointCount function, but the same principle applies to the other functions that can be forwarded.

Whenever a request needs to be forwarded, the function in the ObservationServer class calls the corresponding function of the CommunicationHandler class. This function creates a message according to a certain format and calls the internal ForwardMessage function while passing the message to it. The ForwardMessage function then takes care of the actual forwarding to all other known observation servers.

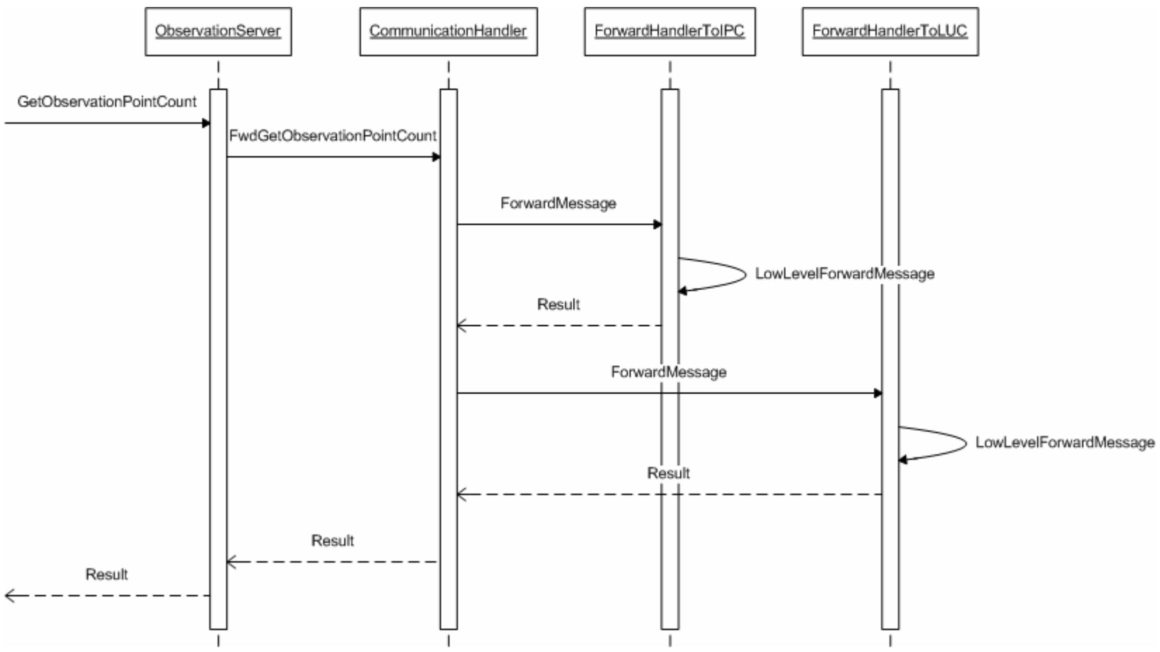


Figure 6.12 Forwarding of requests to other observation servers

6.3.2.3 Checking of a rule by the diagnostic engine

The sequence diagram shown in Figure 6.13 shows how the observation server is involved in the main task of the diagnostic engine, the checking of a diagnostic rule. As is explained in paragraph 6.2.6.2 this checking consists of two parts. For the observation server this means that in the first part the session ID is retrieved and all observation points are reset.

Then the action message is shown to the user (the observation server is not involved in this, but it is shown as an indication which actions happen in part one and which actions happen in part two). After this the session ID is retrieved again. If this session ID is changed, then the target software (including the observation server) has crashed or rebooted while the diagnostic rule was running. This means that the results are unreliable since not all observations might have been seen (those before the crash/reboot are not seen since on crash/reboot all observation points reset again). In this case the execution of the diagnostic rule should be aborted and the complete test should be retried.

When the session ID has not changed then the values of all observation points that are used in the models can be retrieved. Since logical names are used in the models, first a lookup needs to be done to convert the logical name to an ID. After this the value can be retrieved. If the value differs from 0 then the observation point was signaled in the time period of the test, if it is 0, then the observation point was not signaled.

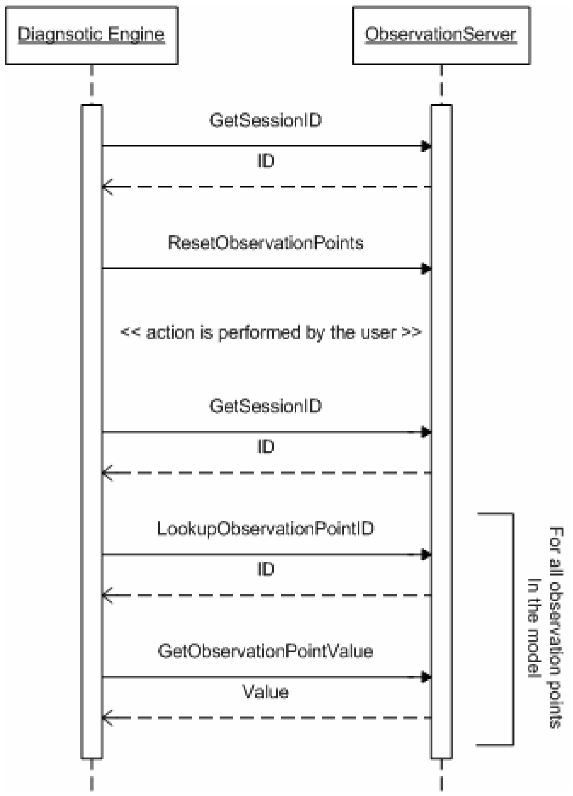


Figure 6.13 Checking of a rule by the diagnostic engine

6.4 Design Decisions

This paragraph will explain and motivate some of the design decisions that have been made during the design phase.

6.4.1 Separate the diagnostic system from the target software that is under diagnosis

One of the experiences in diagnosis at Philips Medical Systems is that once a system is introduced in the field, faults occur at places where the designers of the systems did not think faults would occur. Since the designers were not aware that these faults could happen, there often is no logging at the place of such a fault that clearly indicates what could be the faulty components. In such case, designers would like to extend the logging in such a way that a clear notification is made in the logging which tells the field service engineer which components could be responsible for the error.

The problem here is that the updated software cannot simply be released for public use because the software controlling medical systems is under strict regulations. Before any new software can be released to customers in the field, it must be rigorously tested. The result is that new software releases are only made available two or three times a year.

If the diagnostic system can be designed as a separate system (where also the models and rules are separate from the target software of the system), then it can be updated much more frequently since only the target software that controls the medical system falls under these regulations of strict software testing.

When looking at Figure 6.1 it can be seen how the separation between the target software and the diagnostic system was realized in our method. The target software is represented by the white blocks marked “Target Software Module”. This is the software that is under strict regulations and that cannot be changed. This also means that the observation servers and the observation points cannot be changed between major software releases.

However, the diagnostic engine, the presentation layer, the system structure file, the diagnostic rule files, the diagnostic model files and the file containing the location of the observation servers are all separate from the target software which means that they can be changed between major software releases. This means that new rules can be added to the diagnostic solution, rules can be changed and models can be created or updated according to field experience gathered. The limitation here is that these new models must use existing observation points, no new points can be added to in between major software releases.

6.4.2 Diagnostic conclusion manager algorithm

In paragraph 5.2.3 it was discussed that it is the task of the diagnostic engine to create a ranked list of components based on what component is the best choice for replacement. In that paragraph an example was given that in practice it might not always be cost and time efficient to replace the component that has the highest probability of being broken, but that factors as the costs of a component and the time it takes to replace a component (or to manually check its healthiness) might result that it is better to consider another component (that does not have the highest probability of being broken) for replacement (the power supply vs. fuse example).

The diagnostic conclusion manager contains an algorithm that can create a list of components ranked on the components that is the best choice for replacement using the following information:

- The gradation of a component (definitely healthy, unknown, possibly broken, definitely broken)
- The a priori failure probability of the component
- The costs of the component
- The time it takes to replace the component

This process is briefly described in Figure 6.14.

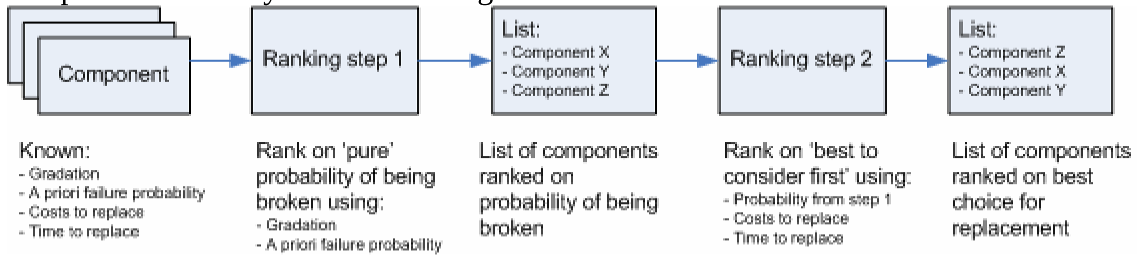

Component
Ranking step 1
List:
- Component X
- Component Y
- Component Z
Ranking step 2
List:
- Component Z
- Component X
- Component Y
Known:
- Gradation
- A priori failure probability
- Costs to replace
- Time to replace
Rank on 'pure' probability of being broken using:
- Gradation
- A priori failure probability
List of components ranked on probability of being broken
Rank on 'best to consider first' using:
- Probability from step 1
- Costs to replace
- Time to replace
List of components ranked on best choice for replacement

Figure 6.14 Process of creating a ranked list of components

The process starts after a diagnostic rule has been executed. This means that a number of diagnostic models have been checked, and that one model indicated that there are or might be faults in a certain module. The result of the checking of this model is a list of components of which their gradation (definitely healthy, unknown, possibly broken or definitely broken) is known. This gradation is given to the component by the model rules. Besides the gradation the a priori failure probability, the costs to replace the component and the time it takes to replace the component are also known. This last information is also contained in the models.

From this information first a list of components is created that is ranked on the pure probability of the component being broken. The gradation and a priori failure probability are used for this. For this ranking a mathematical function is used which we will give the name "ranking function step 1" and it will be precisely described later. After this a second mathematical function is used to create a list of components that is ranked on which component is the best choice for replacement. This function uses the result from the "ranking function step 1" together with the costs and time to replace a component. We will call this second function "ranking function step 2".

Ranking function step 1

We will now give a mathematical definition for the ranking function that is used in step 1. We will first look at what basic conclusion can be drawn from the given information:

- A component marked definitely healthy should never be in the list.
- A component marked definitely broken should always have a higher ranking than a component marked possibly broken (because of the definition of definitely broken).
- A component that has the same marking (definitely or possibly broken) as another component, but with a higher a priori failure probability should have a higher ranking than the component with the lower a priori failure probability.

Considering these conclusions, the ranking function for step 1 is simple. It should first rank the components based on their gradation (where components with a gradation of unknown or definitely healthy are removed from the list), and within the resulting two groups of components it should rank the components based on their a priori failure probability.

We first define *Gradation*("Definitely Healthy") = -2, *Gradation*("Unknown") = -1, *Gradation*("Possibly Broken") = 0 and *Gradation*("Definitely Broken") = 1. Furthermore we define the *component_gradation* as the gradation given to the component by the model rules and *component_failure_probability* as the a priori failure probability given to the component by the model.

Then considering that the a priori failure probability is a number between 0 and 1 the following function can be used to calculate the rank in step 1:

$$\text{Rank} = \mathit{Gradation}(\mathit{component\_gradation}) + \mathit{component\_failure\_probability}$$

The component with the highest score for "Rank" is the component that has the highest probability of being broken.

Ranking function step 2

In step two we consider three factors that determine what component is the best choice to be replaced. We consider the rank from step 1, the costs of a component and the time it takes to replace a component. From these three factors a final score needs to be calculated and then the components are sorted on this score such that the component with the highest score is the best choice of being replaced.

Because not all factors should have an equal weight in the final outcome, we define weights for each of these three factors and calculate a final score using the weights. Using weights the final outcome of this step can be tuned with field experience by adjusting the weights. For the function of step 2 we define the variables as shown in Table 6.1.

Variable	*Description*
P_{broken}	The initial ranked probability (from step 1)
$T_{replace}$	The time it takes to replace the component
$C_{component}$	The costs of a component
W_{prob}	The weight factor of the probability
W_{time}	The weight factor of the time

W_{costs}	The weight factor of the costs

Table 6.1 Variables involved in re-ranking

The new rank can then be calculated using the following formula:

$$\text{Rank} = P_{broken} * W_{prob} + (1 - T_{replace}) * W_{time} + (1 - C_{component}) * W_{costs}$$

Note that the weight should have a value between 0 and 1 and that the values for $T_{replace}$ or $C_{component}$ should be normalized such that they also represent a value between 0 and 1.

The exact values of the weights are not yet known and should be determined together with PMS.

6.4.3 Performance of Observation Servers

When investigating the assignment, it became clear that performance is an issue since the target software has fixed timing constraints. For instance, the main control loop of the software running on the Geo IPC executes every 50 ms. If an iteration of the main control loop takes more than 50 ms a watchdog terminates and restarts the Geo IPC software.

Since there might be observation points that need to be signaled in functions that are called by the main control loop, an efficient solution for the signaling needs to be found.

Therefore it is chosen that each observation point should be kept as simple as possible. An array of integers is chosen to keep track of all observation points. Any signaling of observation points is then registered as the increment of an element of the array. Since integer increment and array lookup operations are both efficient, the performance impact is minimal. Whether an observation point is then signaled can be checked by comparing the integer values at the beginning of the diagnosis and at the end of the diagnosis. If the two values differ, the observation point is signaled.

This method has one flaw however. In the case where an observation point is signaled exactly MAX_INT times, the integer values are the same before and after the diagnosis, and the signaling is not seen. For now, this case is considered extremely unlikely and the drawback is discarded.

6.4.4 Multiple intercommunicating Observation Servers

The software that controls the system of Philips Medical Systems is distributed over a number of PC’s and other hardware platforms. Since every process needs its own observation server (inter-process communication would cost too much performance) this means that there will be multiple observation servers. However, some of these hardware platforms are not directly accessible from the PC that runs the diagnostic engine, and needs to be accessed via a specific other hardware platform. In these cases, an Observation Server should be able to forward an external request (for example, a request to retrieve the value of a certain observation point) to another Observation Server that is not directly accessible.

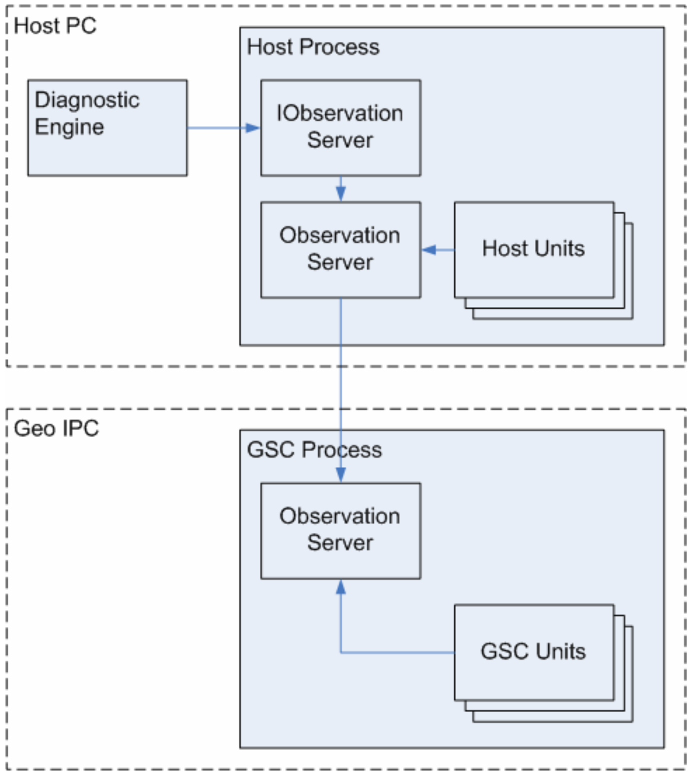


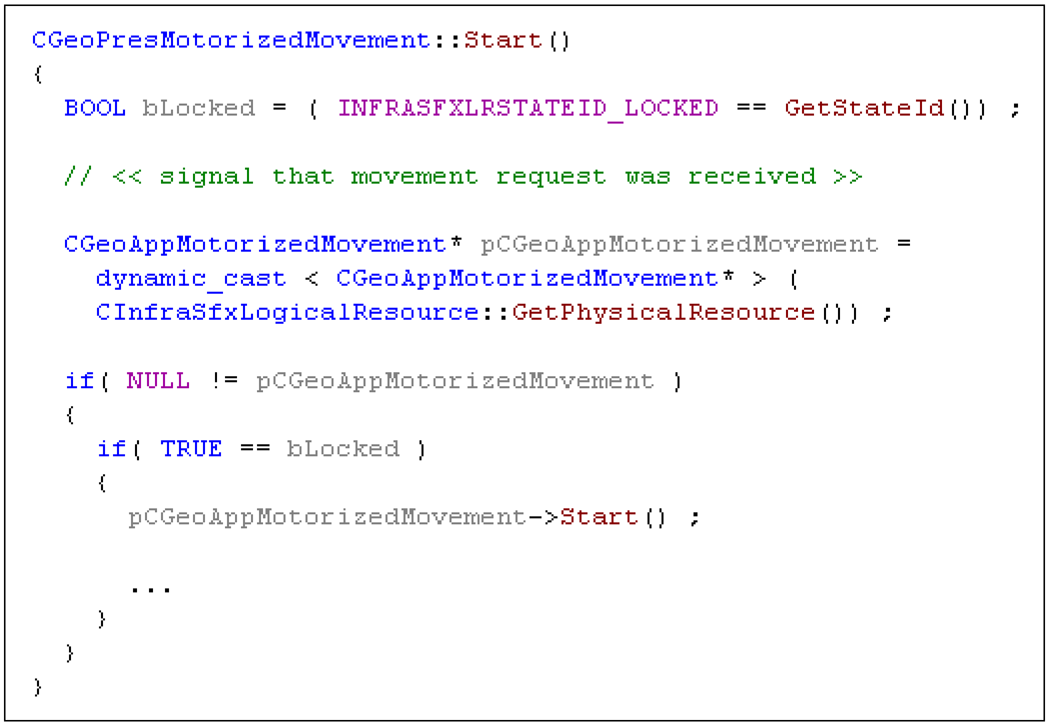
Figure 6.15 Overview of multiple Diagnostic Servers in system

Figure 6.15 shows this for the situation at PMS (although only a small part of the controlling software is shown). The request received by the observation server in the Host Process is forwarded to the Observation Server in the GSC Process and the result is returned. This picture only shows a small part though. In reality, the software on the Geo IPC controls a number of embedded real-time hardware platforms. These platforms also cannot be directly accessed, so the Observation Server in the GSC Process should be able to forward any requests received from the Observation Server in the Host Process to the Observation Servers on these embedded platforms.

6.4.5 Creation of Observation Points

The sequence diagram in paragraph 6.3.2.1 shows that observation points need to register themselves at the observation server. To register the code constructs a logical name which it passes to the observation server. The observation server then returns a unique ID to the code which is then used for signaling the observation point. We will now discuss why it is necessary to first construct a logical name and why it is not possible to just use unique ID's that have been created at compile time.

Consider the fragment of real code shown in Figure 6.16. This function is called when a movement request is received and it has been approved by the host software.



```
CGeoPresMotorizedMovement::Start()
{
  BOOL bLocked = ( INFRASFXLRSTATEID_LOCKED == GetStateId()) ;

  // << signal that movement request was received >>

  CGeoAppMotorizedMovement* pCGeoAppMotorizedMovement =
    dynamic_cast < CGeoAppMotorizedMovement* > (
    CInfraSfxLogicalResource::GetPhysicalResource()) ;

  if( NULL != pCGeoAppMotorizedMovement )
  {
    if( TRUE == bLocked )
    {
      pCGeoAppMotorizedMovement->Start() ;

      ...
    }
  }
}
```

Figure 6.16 Code example of movement starting

At the point of the line that says "signal that movement request was received" we want to add statements such that an observation point is signaled that a movement request was received and approved by the host software. Suppose we have a CObservationServer singleton class with a function SignalObservationPoint() that takes an ID of the observation point that is signaled (this is how the Observation Server is designed). Then we could add the following statement:

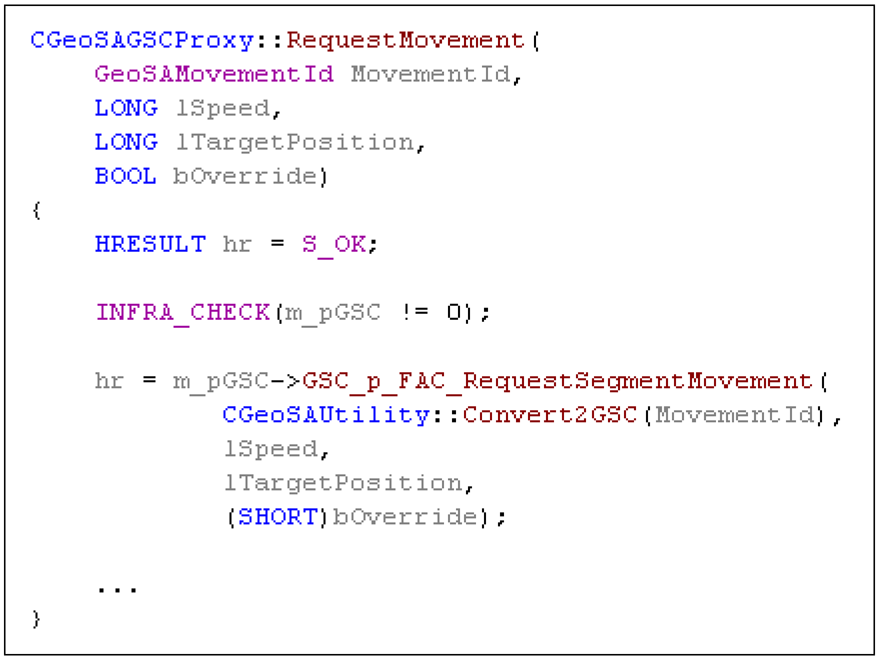
```
CObservationServer::Instance()->
SignalObservationPoint(MOVEMENT_STARTING);
```

If then a movement is started the observation point "`MOVEMENT_STARTING`" is signaled and we can use this observation point in our models. However, we want more detailed information about which movement was started. Now we have a problem because the function shown in Figure 6.16 is called for every movement type. If we would add the following statement:

```
CObservationServer::Instance()->
SignalObservationPoint(TABLE_HEIGHT_MOVEMENT_STARTING);
```

Then the observation point "`TABLE_HEIGHT_MOVEMENT_STARTING`" would indeed be signaled if we would request a table height movement from the input controls. However, if we would request a table lateral movement or an L-Arc movement this observation point would also be signaled.

The problem is that the code as is shown in Figure 6.16 is a so-called "generic code" block which means that it contains code that is executed for all movements (in this project we just consider the code that controls the movements). After further investigation we have found that there are generally two types of generic code. The first type is a generic function in which a parameter of that function indicates which movement the code is for. An example of this type is shown in Figure 6.17.



```
CGeoSAGSCProxy::RequestMovement(
    GeoSAMovementId MovementId,
    LONG lSpeed,
    LONG lTargetPosition,
    BOOL bOverride)
{
    HRESULT hr = S_OK;

    INFRA_CHECK(m_pGSC != 0);

    hr = m_pGSC->GSC_p_FAC_RequestSegmentMovement(
            CGeoSAUtility::Convert2GSC(MovementId),
            lSpeed,
            lTargetPosition,
            (SHORT)bOverride);

    ...
}
```

Figure 6.17 Generic function that uses parameter to specify movement type

Here the parameter "`MovementId`" makes this general movement function specific for one type of movement.

The second type of generic code is a class in which the constructor takes the movement type as a parameter. The generic functions in this class can then use the class member variable of the class to make it specific for one type of movement. The difference with the first type is the scope in which the information about the movement type is available, which is at class level in the latter case and at function level in the first case.

The solution that has been designed is to construct a logical name for the observation point at runtime, using the function parameter or class member variable to make the observation point specific for one type of movement. For instance, code to create a specific observation point for the code shown in Figure 6.16 could be:

```
sprintf(sOPName, "GEOSA_GSCPROXY_REQUEST_MOVEMENT_%s",
            MovementIdToString(MovementID));
```

This statement assumes the existence of a `MovementIdToString` function which can be created if necessary (but in our case such a function already exists, called "Convert").

The result of this statement is that in case of a height movement an observation point called `GEOSA_GSCPROXY_REQUEST_MOVEMENT_TABLE_HEIGHT` is created, and in case of an L-arc propeller movement an observation point called `GEOSA_GSCPROXY_REQUEST_-MOVEMENT_LARC_PROPELLER` is created. Now every movement type has its own observation point which is exactly what we wanted.

One last remark is that string handling (the creation of the name and the conversion of the name to a unique observation point ID at the observation server) is expensive performance wise. Therefore the aim should be to construct the logical name and retrieve the unique observation point ID in the initialization of the software unit (if possible). The retrieved observation point ID can then be stored locally and the statements that signal observation points can then use this locally stored observation point ID. In some cases this is not possible and another performance efficient solution needs to be chosen (for instance a switch statement with all possibilities).

6.4.6 Fixed start situation

The diagnostic engine creates a diagnosis based on comparing the observed behavior with the behavior described in the models. Therefore, the models should be described using a fixed starting situation such that if a machine is in this state and the diagnostic test is run, the outcome should always be the same (while taking the assumption into consideration that intermittent problems are out of scope). Of course, this is the ideal situation and it might not always be possible, but care should be taken to strive that this holds in most cases.

To achieve this, the following three measures are taken:

The system should be restarted before the diagnostic test is run
When a component breaks, the system might not always detect it immediately. Then after a restart of the system it might detect that something is wrong and disable the movement (this happens for quite a lot of problems). Without this measure, the system behavior could be quite different if the diagnostic test is run before the problem is detected by the system than it would be if the diagnostic test is run after the problem is detected by the system. Furthermore this measure might prevent different system behavior from non-persistent settings that are done by the field service engineer in order to find the problem.

The observation points should be reset before the diagnostic test is started
The previous measure already partly covers this measure, but before the diagnostic test is started the observation points should be reset, such that only observation points that are signaled during the test are taken into account.

The diagnostic test should precisely describe which actions need to be taken in order to execute that test
Since the diagnostic system is currently designed as a passive system, which only observes system behavior, actions need to be performed to ensure that control paths are taken through the software that otherwise remain idle (the controlling of the movements

and the hardware involved with that task). The actions and the order in which they need to be taken need to be precisely described and followed by the service engineer. This way, the system behaves as was modeled.

These three measures should take care of a fixed start situation. Any deviations that might still be possible should be incorporated into the models. This means that they must be robust enough to handle this.

7 Validation

In this chapter the validation of the diagnostic solution and its prototype implementation will be described. First the method of validation will be described and then the results are presented. An interpretation of these results will be given and a short conclusion is given based on these results. The main conclusion is presented in the next chapter.

7.1 Validation plan

Validation of the diagnostic solution can be done in a number of ways. Though this number of ways is limited with the knowledge that the prototype solution is not officially approved software. This means that validation can only be done internally on test systems and that it is not allowed to test it on real systems in the field.

The downside of this is that the diagnostic solution can not be tested on real fault situations. Instead, faults have to be injected in a test system and then the diagnostic solution can be used to see if it detects the fault.

Together with PMS it has been decided that a number of test cases will be made where each test case contains one fault. Then the diagnostic solution should be used to see if it can find the fault. For the validation, two groups of test cases are created.

For the demonstration of the prototype the AD7 table has been chosen. This choice was made together with experts of PMS because the table is relatively new, there is not yet much knowledge about it amongst field service engineers, it is mechanically more complex than other tables and also quite important, the available test systems are all equipped with an AD7 table.

Note that for the validation the a priori failure probabilities have not been taken into account. The reason for this is that these probabilities are based upon statistics about which FRU’s have a higher chance of failing. But in our test situation, each FRU will fail once (we will simulate the failure of each FRU in a different test) meaning that using a priori failure probabilities would negatively influence the results and that it would not be realistic since the a priori failure probabilities do not apply when faults are injected in the system. The result of this decision is that components can only be ranked in one of the four gradations and that no ranking within these gradations is possible.

7.1.1 Test cases group 1

For the first group of test cases, an overview will be made of the different FRU’s that are involved in the part of the system that is covered by the prototype. A test will then be made for each FRU in which it is simulated as being broken. Of course, an FRU can break in many ways, but for the validation we will chose to simulate each component as completely electrically broken (so no signals come in or out of the FRU). The reason for this is that it is the easiest way to simulate. More complicated errors are much harder to simulate and in some cases it is even impossible (without actually breaking the FRU).

The different test cases will be explained now:

0. Everything OK
The first test case will be the case where the whole system is OK. The diagnostic solution should detect that all (involved) modules are working correctly.

1. Broken UI module
A broken UI module is not able to send any movement commands to the controlling software. The diagnostic solution should be able to detect that the input module is broken since it does not receive any input.

2. Broken CAN cable from UI module to TTCB
The UI module is connected to the Table Top Connection Box (TTCB) via a cable. The CAN signal goes through this cable. If the CAN lines are broken, then the controlling software does not receive any input and this should be detected by the diagnostic solution.

3. Broken enable_move line from UI module to TTCB
One of the safety mechanisms is that besides a software signal also a hardware signal is used to enable movements. When this line is broken, the controlling software receives movement commands but detects that the enable_move line stays false.

4. Broken network card in Host PC
The Host PC contains a network card for the communication with the IPC. This network card can be broken and when this happens there is no communication possible between the Host PC and the IPC.

5. Broken Ethernet cable from network card in host PC to switch
The cable from the Host PC to the switch is part of the Ethernet communication channel between the Host PC and the IPC. It could break and when this happens there is no communication possible between the Host PC and the IPC.

6. Broken Ethernet switch
The Ethernet switch is part of the Ethernet communication channel between the Host PC and the IPC. It could break and when this happens there is no communication possible between the Host PC and the IPC.

7. Broken Ethernet cable from switch to IPC
The Ethernet cable from the switch to the IPC is part of the Ethernet communication channel between the Host PC and the IPC. It could break and when this happens there is no communication possible between the Host PC and the IPC.

8. Broken IPC
The IPC can break down in various ways. Many ways will prevent it from starting up which should be detected by the Host PC since no communication is possible to the IPC.

9. Broken CAN cable from IPC to SIB
There is a CAN cable from the IPC to the System Interface Board (SIB) through which CAN signals pass. This cable can break and this will result in an unavailable CAN network.

10. Broken IO cable from IPC to SIB
There is also an IO cable from the IPC to the SIB. Various IO signals go through this can cable and if it breaks, no movements are possible.

11. Broken Smart Drive for height and tilt movement
Each motorized movement (height, tilt, lateral, longitudinal, cradle) contains a device that directly controls the movement. It sends signals to the motor and the brake and it receives input from the potentiometer. This device is called a smart drive. When it is broken, the movements will not work.

12. Broken potentiometer for height and tilt movement
Each motorized movement (height, tilt, lateral, longitudinal, cradle) contains a potentiometer which is used to detect movements. Since the rotating axis of the potentiometer is attached to the moving part, the increase or decrease in resistance of the potentiometer can be used to calculate to movement in millimeters or degrees. When this device is broken, the Smart Drive no longer has feedback if the table is moving.

13. Broken motor for height and tilt movement
Each motorized movement (height, tilt, lateral, longitudinal, cradle) contains a motor that controls the movement. The voltage, current and also the frequency and the amplitude of the current are directly regulated by the Smart Drive. When the motor does not work, no movement is possible.

14. Broken brake for height and tilt movement
Each motorized movement (height, tilt, lateral, longitudinal, cradle) contains a brake that will prevent movements if no movement request is done. When the brake is broken no movement is possible (since the brakes are designed in such a way that they will be on if the internal electrics of the brake are broken).

15. Broken emergency stop cable
At various places in the room there are emergency stop switches available (at the walls, at the input modules, at the table). These are all part of one electrical loop and if this is interrupted at any place, the emergency stop procedure is activated. Whenever an emergency stop cable is broken somewhere, the emergency stop will always be activated.

7.1.2 Test cases group 2

For the second group of test cases, a number of possible problems have been thought up by experts of PMS. Most of these are not detected by the POST or by the logging. All of the test cases in this group are shortly explained:

16. Broken AD7 tilt sensor
The AD7 table has a tilt sensor which is not a separate FRU but for demonstration purpose it is interesting to check whether it is possible to detect if this component is broken.

17. Loose potentiometer
With a loose potentiometer is meant a potentiometer of which the rotating axis is no longer mechanically connected to the moving part of the table. This results in a movement not being detected by the potentiometer.

18. Movement denied by 3D model
A situation that might occur that is closely related to the previous one is when the table is close to another part of the system and a movement is denied by the software because the 3D model predicts a collision. Sometimes this is not entirely obvious, and it could be mistakenly be interpreted as a hardware fault. The diagnostic system should recognize this and indicate that nothing is wrong with the hardware.

19. Motor over current
If a hardware panel is bent, or some cable tree moves out of position this might cause a situation where the motor current goes over a certain threshold (because of the increased friction). This should be recognized by the diagnostic system.

20. IPC crash during test
It could happen that due to a software bug the IPC crashes during a test. This should be detected by the diagnostic system and it should not mistakenly mark parts as broken in such a situation.

21. No power to input module
22. No power to IPC
23. No power to AD7 table
24. No power to height motor
The power net is not incorporated into the diagnostic models. However, we will test faults with the power net anyway to see how the diagnostic solution handles faults that are outside of its scope.

25. Emergency stop pressed
When the emergency circuit is activated, all movement is disabled. Although this gives a clear user message, the diagnostic solution should detect that the hardware is working correctly and that the emergency stop is pressed.

26. Broken tilt button on UI module
A single button of the UI module could be broken. This should be detected by the diagnostic solution.

27. Broken Synqnet loop

Finally all drives in the AD7 table are connected via a ring network called Synqnet. If a connection anywhere in this ring is broken, it should be detected by the diagnostic solution.

7.2 Validation results

To present the validation results, first it needs to be defined when a test is passed and when a test is failed. A test is passed if the following three rules hold (unless specified otherwise by the test description):

- *The right module is detected as being suspicious or failed*
  The prototype solution has been made on a part of the system that contains five modules (Host PC, Network, Embedded PC, Input and AD7). The module that contains the faulty component should be correctly identified as being suspicious or failed.

- *The faulty component is diagnosed as either possibly broken or definitely broken*
  The component that is simulated broken should be diagnosed as either possibly broken or definitely broken.

- *There are no components listed as possibly broken or definitely broken that could have never been broken in that situation (within the limitations of the target software)*
  The list of fault candidates should be as short as possible, i.e. it should not contain any components that could have been diagnosed as definitely healthy in that situation.

Now the results of the validation will be presented. Each test case will be presented separately and an explanation of the results will be given.

7.2.1 Test cases of group 1

These are the results of the tests of group 1.

Test case 0: Everything OK

Outcome

```
Host PC: healthy
Network: healthy
Embedded PC: healthy
Input: healthy
AD7: healthy
```

Result Passed

Remarks Every diagnostic module is marked as healthy so the results are OK.

Test case 1: Broken UI module

Outcome

```
Host PC: healthy
Network: healthy
Embedded PC: suspicious
```

```
Input: failed
AD7: dependency failed

Possibly broken:
•  IO cable from Geo-IPC to adapter card
•  Adapter card
•  UI Module
•  Cable from UI Module
•  Cable from TTCB to TBCB
•  Cable from TBCB to SIB
•  System interface board
```

Result Passed

Remarks The input module is correctly identified as the faulty module. In this module, no distinction can be made between the UI module itself being broken or any of the cables involved being broken. Therefore they are all listed.

Because the enable_move signal is also not active, the adapter card and IO cable are also listed as possible broken components. The model could be improved here though, as it could be modeled that if there are no input signals detected, then it is logical that the enable_move signal is not detected. (This addition would also clear the suspicious tag for the embedded pc module).

Test case 2: Broken CAN cable from UI module to TTCB

Outcome Same as test #1

Result Passed

Remarks Same as test #1

Test case 3: Broken enable_move line from UI module to TTCB

Outcome

```
Host PC: healthy
Network: healthy
Embedded PC: failed
Input: dependency failed
AD7: dependency failed

Definitely broken:
•  GSC Software
```

Result Failed

Remarks To test this, the software simulates that the enable_move line is always false. However, the logging indicates that the enable_move line became true after all. This indicates that the observation point used is closer to the input hardware than the place where the simulation forces enable_move to be false. Therefore, the model detects that the enable_move observation point was signaled, while the GSC Software does not detect that enable_move becomes true since the simulation forces enable_move (at a higher level) to be false. Therefore the GSC Software blocks the movement. This means that the model actually correctly identifies the GSC Software as the cause of the disabled movement and this test needs to be redone using another method to simulate the broken enable_move line.

Test case 4: Broken network card in host PC

Outcome

```
Host PC: healthy
Network: failed
Embedded PC: dependency failed
Input: dependency failed
AD7: dependency failed

Definitely broken:
•  Network connection

Possibly broken:
•  Host PC network card
•  Ethernet cable from Host PC
•  Geo ethernet switch
•  Ethernet cable
•  Geo-IPC
```

Result Passed

Remarks The components that are involved in the networking are all listed as possibly broken and since currently there is no way of making a further distinction between these components this is the best possible result. However, it is noted that the switch involved also connects other parts of the system and if in these other parts observation points and servers are placed, a further distinction might be possible.

Secondly, the model has incorporated a safety feature that detects if the network connection was lost during the test. If this happens the test is marked as unreliable. This is the “network connection” component that is marked broken in this test. The model could however be improved by using the observation points to make a difference between the state in which there never was a connection during the test, and the state in which there was a connection during the test but this connection was lost. If the model would be improved this way, then the “network connection” component would also disappear from the list.

Test case 5: Broken Ethernet cable from network card in host PC to switch

Outcome The same as test #6

Result	Passed
Remarks	The same as test #6

Test case 6: Broken Ethernet switch

Outcome	The same as test #6
Result	Passed
Remarks	The same as test #6

Test case 7: Broken Ethernet cable from switch to IPC

Outcome	The same as test #6
Result	Passed
Remarks	The same as test #6

Test case 8: Broken IPC (no power to IPC)

Outcome	The same as test #6
Result	Passed
Remarks	The same as test #6

Test case 9: Broken CAN cable from IPC to SIB

Outcome

```
Host PC: healthy
Network: healthy
Embedded PC: healthy
Input: healthy
AD7: healthy
```

Result Failed

Remarks For the test it was assumed that this cable contained the enable_move line. This assumption proved wrong, since movements did work while the cable was disconnected. Further investigation pointed out that this cable is only responsible for the connection from the IPC to the lateral and frontal stand (which are out of scope for this prototype). Therefore, everything within the tested scope was indeed functioning ok, so the outcome is actually correct.

Test case 10: Broken IO cable from IPC to SIB for tilt movement

Outcome	The same as test #12.1
Result	Failed

Remarks One of the lines of the IO cable is the enable_move line. During modeling it was assumed that whenever this signal was seen enabled on the Geo IPC, this cable was healthy (since the signal needed to travel through that cable and partly broken components are out of scope). However, after investigation it appears that this signal is also high when the cable is completely removed from its connector. Therefore the model needs to be adapted and the rules that indicate whether the IO cable is healthy or possibly broken should be revised.

Table components tests

For the following eight tests, first a couple remarks need to be made. Most of these faults are detected by the POST of the Embedded PC. This means that the movements are already blocked at the Host PC and the control flow stops there. The diagnostic server currently has no means of making a better distinction between these problems (and should therefore refer to the post in these cases). Each test is divided into an x.1 and x.2 variant. The x.1 is for the height drive, the x.2 is for the tilt drive.

Test case 11.1: Broken smart drive for height movement

Outcome

```
Host PC: healthy
Network: healthy
Embedded PC: suspicious
Input: healthy
AD7: suspicious

Possibly broken:
•  XMP card
•  Smart drive for the height
•  Potmeter assembly for the height
•  Motor assembly for the height
•  Drive assembly for the height
•  Smart drive for the tilt
•  Potmeter assembly for the tilt
•  Motor assembly for the tilt
•  Cable set assembly for the tilt
•  Actuator assembly for the tilt
•  Smart drive for the cradle
•  Drive assembly for the cradle
•  Cable set assembly for the cradle
•  Tilt level indicator
```

Result Passed

Remarks This error is detected by the post, and all table movements are disabled. Therefore, the diagnostic solution is only able to indicate that there is a problem with the table (or with the XMP card controlling it). The diagnostic solution should refer to the POST results in this case. Moving the diagnostic system from passive to active (allowing it to perform actions such as calling certain functions) might result in a better diagnostic resolution, but referring to the POST results should give as much information (though it is an extra step for the service engineer).

Test case 11.2: Broken smart drive for tilt movement

Outcome The same as test #12.1

Result Passed

Remarks The same as test #12.1

Test case 12.1: Broken potentiometer for height movement

Outcome

```
Host PC: healthy
Network: healthy
Embedded PC: healthy
Input: healthy
AD7: suspicious

Possibly broken:
•  Smart drive for the height
•  Potmeter assembly for the height
•  Motor assembly for the height
•  Drive assembly for the height
```

Result Passed

Remarks This error is detected by the post, and the height movement is disabled. The components that are involved in the height movement are listed as possible fault candidates.

Test case 12.2: Broken potentiometer for tilt movement

Outcome

```
Host PC: healthy
Network: healthy
Embedded PC: healthy
Input: healthy
AD7: suspicious

Possibly broken:
•  Smart drive for the tilt
•  Potmeter assembly for the tilt
•  Motor assembly for the tilt
•  Cable set assembly for the tilt
•  Actuator assembly for the tilt
•  Tilt level indicator
```

Result Passed

Remarks The same comment as in #13.1 also holds here but then for the tilt movement instead of the height. An additional comment can be made that the tilt level indicator is listed as possibly broken. This can be solved by modifying the model such that it incorporates that the tilt level indicated can never work if the movement did not work.

Test case 13.1: Broken motor for height movement

Outcome	The same as test #12.1
Result	Passed
Remarks	The same as test #12.1

Test case 13.2: Broken motor for tilt movement

Outcome

```
Session ID changed during test, test unreliable
```

Result	N/A
Remarks	The GSC Software crashed during this test, which was detected by the session ID change and therefore the test is discarded.

Test case 14.1: Broken brake for height movement

Outcome

```
Session ID changed during test, test unreliable
```

Result	N/A
Remarks	The GSC Software crashed during this test, which was detected by the session ID change and therefore the test is discarded.

Test case 14.2: Broken brake for tilt movement

Outcome

```
Session ID changed during test, test unreliable
```

Result	N/A
Remarks	The GSC Software crashed during this test, which was detected by the session ID change and therefore the test is discarded.

Test case 15: Broken emergency stop cable

Outcome

```
Host PC: healthy
Network: healthy
Embedded PC: suspicious
Input: suspicious
AD7: suspicious

Broken:
•  Emergency stop circuit

Possibly broken
•  XMP card
```

- `UI Module`
- `Cable from UI Module`
- `Cable from TTCB to TBCB`
- `Cable from TBCB to SIB`
- `System interface board`
- `Smart drive for the height`
- `Potmeter assembly for the height`
- `Motor assembly for the height`
- `Drive assembly for the height`
- `Smart drive for the tilt`
- `Potmeter assembly for the tilt`
- `Motor assembly for the tilt`
- `Cable set assembly for the tilt`
- `Actuator assembly for the tilt`
- `Smart drive for the cradle`
- `Drive assembly for the cradle`
- `Cable set assembly for the cradle`
- `Tilt level indicator`

Result	Passed
Remarks	The emergency stop circuit is correctly identified as being responsible for disabling movements. The model could be improved however, if it were added to the model that it is normal that all movements are disabled when the emergency stop circuit is activated, then all the components under possibly broken would be removed from the list.

Summary

The result of the validation of the first test group is presented in Table 7.1. An interpretation of these results is given below. An explanation of when a test is passed and when a test is failed is given in the beginning of paragraph 7.2.

Tests	*Passed*	*Failed*	*N/A*
20	14	3	3

Table 7.1 Validation results for test group 1

For each test an explanation is given in the “Remarks” section of that test why it passed or failed. The summary is given here.

Group 1 contains 24 test cases. From these 3 tests were given the result N/A because the IPC software crashed during this test. When the IPC software crashes, the observation points are reset. This means that the diagnostic engine can no longer derive whether the observation points in the IPC were signaled or not. This situation is detected by the diagnostic engine and when this happens the test is abandoned.

For the failed tests it needs to be remarked that test #3 failed because the error was not simulated correctly. Test number #9 and #10 failed because of modeling errors. Test #3 actually passed when it was retried later and simulated correctly. Further investigation is necessary to check how the modeling errors for the tests #9 and #10 can be resolved.

7.2.2 Group 2

These are the results of the tests of group 2.

Test case 16: Broken AD7 tilt sensor

Outcome

```
Host PC: healthy
Network: healthy
Embedded PC: healthy
Input: healthy
AD7: suspicious

Possibly broken:
•  Tilt level indicator
```

Result Passed

Remarks The broken AD7 tilt sensor is correctly identified as (the only candidate) being possibly broken.

Test case 17: Loose potentiometer for height movement

Outcome

```
Host PC: healthy
Network: healthy
Embedded PC: healthy
Input: healthy
AD7: suspicious

Possibly broken:
•  Potmeter assembly for the height
```

Result Passed

Remarks The loose potentiometer for the height movement is correctly identified as (the only candidate) being possibly broken.

Test case 18: Potentiometer with a lot of noise

Outcome Not tested

Result N/A

Remarks This could not be simulated and therefore the test was discarded.

Test case 19: Wrong calibration

Outcome	Not tested
Result	N/A
Remarks	This test was not possible since the calibration procedure of a movement expects its parameters to be in a certain range. When the parameters are too far out of range (which is what we wanted to simulate), the values are not accepted. Therefore it is not possible to calibrate the machine that it is rigorously out of calibration.

Test case 20: Movement restricted by 3D model

Outcome

```
Host PC: healthy
Network: healthy
Embedded PC: healthy
Input: healthy
AD7: suspicious

Possibly broken:
•  Tilt level indicator
```

Result Failed

Remarks The outcome of this test is debatable. The tilt level indicator should not be listed as possibly broken. This can be fixed by extending the model in such a way that if no tilt movement is possible, it can never be said if the tilt level indicator is healthy or not.

If the model was extended this way, all parts in the table would be diagnosed as unknown, in which case the diagnostic engine concludes that the table is healthy (this is a design decision). In that case, the results would be correct.

Part of the problem is that the software does not completely block the movement; it allows small movements (under reduced performance / override). Therefore the diagnostic engine does not detect that movements are blocked (since they are not), but the movements are too small to be certain that there was movement.

Test case 21: Motor over current for height movement

Outcome

```
Host PC: healthy
Network: healthy
Embedded PC: healthy
Input: healthy
AD7: failed

Broken:
•  Height drive over current

Possibly broken:
•  Smart drive for the height
•  Potmeter assembly for the height
•  Motor assembly for the height
•  Drive assembly for the height
```

Result Passed

Remarks This specific error is correctly detected. All components that are involved in the movement are listed as possibly broken.

Test case 22: IPC crash during test

Outcome

```
Session ID changed during test, test unreliable
```

Result Passed

Remarks The crash is successfully detected, and the test results are marked as being unreliable. If this happens multiple times, the diagnostic engine should refer to the event viewer and if that shows no errors or warnings a software error should be reported back.

Test case 23: Wrong configuration on Host PC

Outcome Not tested

Result N/A

Remarks This test was not possible since the Geo IPC detects that the configuration of the Host PC is different from the actual configuration and therefore it does not start.

Power tests

The power lines and connections to the different parts were not considered in the models. Therefore, the following four tests will give wrong results, but this is a test to see how the diagnostic engine behaves when something that was not modeled is encountered (and if those results are acceptable).

For all of the four cases below holds that it would be good to extend the model to incorporate the power design, and to add observation points at places where it is possible to measure if the power input is ok there.

Test case 24: No power to input module

Outcome

```
Host PC: healthy
Network: healthy
Embedded PC: suspicious
Input: failed
AD7: dependency failed

Possibly broken:
•  IO cable from Geo-IPC to adapter card
```

- `Adapter card`
- `UI Module`
- `Cable from UI Module`
- `Cable from TTCB to TBCB`
- `Cable from TBCB to SIB`
- `System interface board`

Result Passed

Remarks The power goes through the CAN cable to the UI module, so this test is the same as the test in which the cable between the UI module and TTCB is broken (#2). The results are ok if indeed the power is cut off due to a broken cable or defect UI module, but if the power is cut off at a much earlier stage, this is not detected and the results would be wrong.

Test case 25: No power to IPC

Outcome

```
Host PC: healthy
Network: failed
Embedded PC: dependency failed
Input: dependency failed
AD7: dependency failed

Definitely broken:
```

- `Network connection`

```
Possibly broken:
```

- `Host PC network card`
- `Ethernet cable from Host PC`
- `Geo ethernet switch`
- `Ethernet cable`
- `Geo-IPC`

Result Passed

Remarks This is the same test as the broken IPC, since a broken IPC was simulated by cutting the power off (#10). If the power to the IPC is cut off due to a broken power supply in the IPC then the fault is identified. If the power is cut off earlier, then the results would be wrong and the fault is not detected.

Test case 26: No power to AD7

Outcome

```
Host PC: healthy
Network: healthy
Embedded PC: suspicious
Input: healthy
AD7: suspicious

Possibly broken:
```

- `XMP card`

- `Smart drive for the height`
- `Potmeter assembly for the height`
- `Motor assembly for the height`
- `Drive assembly for the height`
- `Smart drive for the tilt`
- `Potmeter assembly for the tilt`
- `Motor assembly for the tilt`
- `Cable set assembly for the tilt`
- `Actuator assembly for the tilt`
- `Smart drive for the cradle`
- `Drive assembly for the cradle`
- `Cable set assembly for the cradle`
- `Tilt level indicator`

Result Passed

Remarks This error is detected by the POST and all table movements are disabled. The diagnostic engine can therefore not test what is wrong since the signals are stopped at the Host PC. It has to be investigated whether it is possible to detect the power failure using observation points.

Test case 27: No power to height motor of AD7

Outcome

```
Host PC: healthy
Network: healthy
Embedded PC: suspicious
Input: healthy
AD7: suspicious

Possibly broken:
```

- `XMP card`
- `Smart drive for the height`
- `Potmeter assembly for the height`
- `Motor assembly for the height`
- `Drive assembly for the height`
- `Smart drive for the tilt`
- `Potmeter assembly for the tilt`
- `Motor assembly for the tilt`
- `Cable set assembly for the tilt`
- `Actuator assembly for the tilt`
- `Smart drive for the cradle`
- `Drive assembly for the cradle`
- `Cable set assembly for the cradle`
- `Tilt level indicator`

Result Passed

Remarks This is the same test as a smartdrive being broken (#14.1). In this case, all movements are disabled and the diagnostic engine should refer to the POST results. The error can be detected there.

Test case 28: Emergency stop pressed

Outcome

```
Host PC: healthy
Network: healthy
Embedded PC: suspicious
Input: suspicious
AD7: suspicious

Broken:
•  Emergency stop circuit

Possibly broken
•  XMP card
•  UI Module
•  Cable from UI Module
•  Cable from TTCB to TBCB
•  Cable from TBCB to SIB
•  System interface board
•  Smart drive for the height
•  Potmeter assembly for the height
•  Motor assembly for the height
•  Drive assembly for the height
•  Smart drive for the tilt
•  Potmeter assembly for the tilt
•  Motor assembly for the tilt
•  Cable set assembly for the tilt
•  Actuator assembly for the tilt
•  Smart drive for the cradle
•  Drive assembly for the cradle
•  Cable set assembly for the cradle
•  Tilt level indicator
```

Result Failed

Remarks It cannot be detected whether the emergency circuit is broken or whether an emergency stop button is pressed. Therefore in the software design it is said that before each test the system should be restarted. This also clears the emergency stop state which is only re-entered if the emergency stop circuit is broken or the emergency stop button is pressed again. Since the description of the test does not state that the emergency stop button should be pressed, the diagnostic engine concludes that the emergency stop circuit is broken.

Test case 29: Broken tilt button on UI module

Outcome

```
Host PC: healthy
Network: healthy
Embedded PC: healthy
Input: failed
AD7: dependency failed
```

```
Broken:
•  UI module
```

Result Passed

Remarks This error is correctly identified.

Test case 30: Broken Synqnet loop

Outcome

```
Host PC: healthy
Network: healthy
Embedded PC: suspicious
Input: healthy
AD7: suspicious

Possibly broken:
•  XMP card
•  Smart drive for the height
•  Potmeter assembly for the height
•  Motor assembly for the height
•  Drive assembly for the height
•  Smart drive for the tilt
•  Potmeter assembly for the tilt
•  Motor assembly for the tilt
•  Cable set assembly for the tilt
•  Actuator assembly for the tilt
•  Smart drive for the cradle
•  Drive assembly for the cradle
•  Cable set assembly for the cradle
•  Tilt level indicator
```

Result Passed

Remarks The POST detects this error and all movements are disabled. The diagnostic engine should refer to the POST results where this fault can be detected.

Summary

The result of the validation of the second test group is presented in Table 7.2. An interpretation of these results is given below. An explanation of when a test is passed and when a test is failed is given in the beginning of paragraph 7.2.

Tests	*Passed*	*Failed*	*N/A*
15	10	2	3

Table 7.2 Validation results for test group 2

For each test an explanation is given in the "Remarks" section of that test why it passed or failed. The summary is given here.

Group 2 contains 15 test cases. From these 3 tests were given the result N/A because the test could not be executed. The explanation why is given in the “Remarks” section of the given tests.

For the 2 tests that failed it holds that test #20 might be resolved if extra observations are added. Test #28 was given the classification failed after discussion with experts from PMS. This classification is debatable and the reason why is further explained in the “Remarks” section of that test.

7.3 Validation conclusion

The goal of the validation was to investigate whether the method has brought improvement in the way diagnosis is done. To be able to draw this conclusion, first it needs to be known what the current status of fault diagnosis is. Unfortunately, this information is not available. Often, a FSE replaces a number of components at the same time. When this happens, it cannot be traced what the actual fault was. Also the actual time needed to come to a diagnosis or the total time of repair is not directly available. Especially in cases where more than one FSE needs to visit a broken system before the fault is found it is not known how much the total time of repair was.

Therefore another way needs to be found to draw a conclusion from the results of the validation. Since the idea that fault diagnosis can be improved came from internal discussions amongst people from PMS, we have chosen to present the prototype and the results of the validation to these people and ask them for their opinion about whether this prototype can improve fault diagnosis. The survey that was used for this is attached as appendix B. People from the Service Innovation department, Field Monitoring team, the Software Development department and the Test department were asked to fill in this survey. In total eight persons were asked of which five returned answers to the survey questions and in some cases some general remarks.

7.3.1 Survey results

Here the survey results will be presented. The results are grouped the same way as is done in the survey.

The way the tests are executed
All five persons that returned the survey indicated that the way tests are selected is clear. A remark is made that the test description does not describe how the test is executed. An extended description would solve this.

The presentation of the diagnosis results
The presentation of the diagnosis results was found clear by all. The coloring was seen as a good way to quickly determine where the fault is and what components are candidates for further investigation. Remarks were made here that it might not always be clear to

which module a component belongs. For instance a network card may be part of the Network module, but also of the Host PC module.

The results of the validation
For the results of the validation the opinions are spread. Two persons indicated that the number of failed test cases is acceptable; one person answered that he couldn't judge that and two persons indicated that the results are not acceptable. One of the latter mentioned that with the remarks the result is acceptable, and the other indicated that all tests should pass since all tests represent scenarios that can occur and that the FSE should always be able to rely on the diagnosis.

Another remark that was made by two persons is that the results look promising for the Geometry subsystem, but that further investigation is needed before a conclusion can be drawn whether this method can be used throughout the whole system.

The usability of this method to improve fault diagnosis
For the usability of this method to improve fault diagnosis, four of the five persons indicated that the results are promising and that on the limited scope on which the prototype was built, it shows good results. Two persons mentioned that there are currently many tools available to diagnose a part of the system, but that it is not clear for the FSE which one he should choose. This method could help the FSE decide which tool he could use for further investigation (if necessary). One person also mentioned that most of those other diagnostic capabilities are not useful enough since they generate a lot of unclear messages and logging.

Two persons also mentioned that it should be avoided that a lot of components get the gradation "Possibly Broken" as this might lead to even more non-broken components that are replaced.

Four of the five persons indicate that this method is promising and that the work done is good as a basis, but that now a prototype with a larger scope should be built to investigate if the method can be scaled up to the entire system. Furthermore a business study is necessary to do a cost-benefit analysis whether the gained benefits outweigh the work needed to implement this method. Finally, one person mentioned that a pilot should be started where this method should be used to find faults on a real system in the field.

7.3.2 Survey conclusion

From the survey it can be concluded that four of the five persons are positive about the usability of the prototype. The fifth person indicated that in his opinion it does not provide much extra diagnostic capability compared to the POST and the logging.

Furthermore, two points of improvement have been found necessary by some of the people that filled in the survey. The first point of improvement should be to improve the models such that all tests pass. Two of the five persons mentioned that none of the tests should fail. As is already mentioned in the remarks section of the failed tests, it should be possible to extend the models such that some of the tests that now fail will pass. On some other tests further investigation is necessary before it can be concluded that those test can pass by adapting the models.

The other point of improvement should be on the resolution of the diagnosis. Two persons mention that there should not be too many components marked as possibly broken because this might lead to more replacement of non-broken hardware. The cause of the low resolution in certain test cases lies in the architecture of the controlling software.

The software that controls the C/V X-Ray system has been designed such that when the POST detects that something is wrong in for instance the height movement, it disables the complete height movement at the level of the Host PC. This means that any controlling signals concerning the height movement are blocked at the Host PC. Since our method uses these signals as observable information, the diagnostic engine is limited in its diagnostic capabilities by this behavior of the controlling software (since no observable information can be gathered after the Host PC).

To solve this issue, three possible improvements have been suggested where each of these might solve the problem. The first solution is to change the behavior of the controlling software in such a way that it will no longer block signals and that signals will always be sent to the hardware. Such behavior might be implemented in a separate operating mode called a “diagnostic mode”. This suggestion is further described in paragraph 9.4.1.

The second solution that is suggested is to add the capability to the models that statements might be derived from observations. Currently, only the healthiness of components is derived from observations but this might be extended in a way such that also statements might be derived from the observations. Such a statement could then be “Table height movement disabled, check the Table POST results” for example. This way no changes are necessary in the controlling software. However, this suggestion does not improve the diagnostic capabilities of the diagnostic solution; it just refers to other diagnostic methods that are available in the system. This suggestion is further described in paragraph 9.1.2.

The last solution that is proposed to solve this issue is to place observation points in the code where the POST results are retrieved. This way the diagnostic solution can use the diagnostic capabilities of the POST and use the derived conclusion of the POST as observable information in the models. This suggestion also does not improve the diagnostic capabilities of the diagnostic solution directly, but rather makes use of the results of other diagnostic methods in the system. This suggestion is further described in paragraph 9.3.2.

7.3.3 Comparison to earlier work

As is already stated earlier in this thesis, there has been another Master's project on improving fault diagnosis at Philips Medical Systems. This project was carried out a few months before the beginning of this Master's project and was carried out by W.M. Lindhoud. His thesis can be found in [8]. It would be valuable for the validation to compare the outcomes of both projects. However a direct comparison is not possible. In his project he concentrated on one specific movement of the system. This selected subsystem only contained 5 components and everything up to there is assumed to be functioning correctly. This means that input is not considered and also the way the Host PC or the Geo IPC influence the behavior of the system are considered out of scope. His thesis does not state if and how his method can be scaled up to the entire system, while the method that is described in this thesis has been designed to be able to diagnose the entire control chain from the input to the movement of the table. Because the scopes of the two projects are very different, a comparison is not possible.

8 Conclusion

The goal of this project was to design a method that improves fault diagnosis for the C/V X-Ray system of PMS. Unfortunately, this cannot be directly concluded. As is also described in paragraph 7.3, to conclude that the designed method can improve fault diagnosis it first needs to be quantified what the current status of fault diagnosis is and then it needs to be quantified how fault diagnosis is improved using the designed method. A comparison can then be made on the factors that are important for fault diagnosis, such as accuracy, resolution, completeness, variability/configurability, extendibility, representation facilities, automation, development costs and maintenance costs. However, such a comparison is not possible on the exact statistics, since the current status of fault diagnosis can not be precisely quantified (which was also concluded in [8]).

Since an exact statistical comparison is not possible, a comparison will be made based on the results of the validation and on the data that has been collected from interviews with people from PMS. The method presented in [8] will also be taken into the comparison.

	Accuracy	Resolution	Completeness	Variability	Extendibility	Representation	Automation	Development costs	Maintenance costs
Current practice of FSE	+	+	0	+	0	0	--	0	0
Method presented in this thesis	+	+	0	+	0	+	+	0	+
Method presented in [8] (MBD)	++	+	+	0	0	+	+	-	0

Table 8.1 Comparison of this method against current practice and MBD

The current practice of low to medium experienced field service engineers scores only moderate on completeness since, using the current method, he is only able to detect a certain percentage of the faults (estimated at 60%-70%). It scores good on accuracy and resolution, since when the right test is selected (or the right fault procedure is followed) this test will usually pinpoint the right component (or a small number of components) that could contain the fault. On variability it scores good since for every different version of the system there are test available. The field service framework (the interface that is used by the field service engineer to find faults) automatically filters the tests or data that are applicable for the current configuration. Extendibility scores moderate, since for every extension to the system it needs to be reviewed what impact that extension has to the field service framework. Representation also scores moderate since the representation only gives a textual description, which is not always clear to the FSE. The major drawback of the current practice is that the field service framework is huge, and it contains many different tests and logging data but it does not contain a general overview of where the fault could be. It is very unclear for the FSE where he should start the diagnosis. Also the development and maintenance costs are moderate since all parts of the system contain code for field service. Furthermore, the field service framework itself is also very large. Therefore development and maintenance requires quite some effort.

The method presented in this thesis has comparable accuracy, resolution and completeness to the current methods of fault diagnosis. However, the major improvement is in automation. The diagnostic capabilities in the current method are not available in one diagnostic test, but are separated over a very large number of diagnostic methods (functional tests, POST's, logging). It is up to the FSE to select the right functional test, or to view the right POST results or the right part of the logging to find out what the fault could be. If he does not select the right part, he will not find anything at all. The method presented in this thesis provides one diagnostic engine with a small number of diagnostic rules that together can diagnose the whole system (and thus reach the same level of accuracy, resolution and completeness with a much smaller test time). The method presented here also has built-in facilities to deal with different configurations (variability) and scores also the same on extendibility (extensions should be reviewed to see what impact they have on the models and model rules). On representation it scores good, since it gives an overview of the system (divided into modules) and lists per module what components could be faulty (ordered by probability of being broken). It scores moderate on development costs, since this is a new method and if it is going to be used, it has to be built up from the beginning, which costs resources. The resources used for this however, are less than with MBD as presented in [8]. Since for using MBD, full behavioral and structural modules need to be made for the entire system (and since the software influences the behavior of the system, the software needs to be modeled too).

The method presented in [8] scores better on accuracy and completeness than both the method presented in this thesis and the current practice, since full behavioral and structural models contain much more information. Therefore a better diagnosis is possible by the diagnostic engine. The reason that it scores good instead of very good on completeness is because a model is still an abstraction from the reality and thus is not exactly the same as the original system and therefore might still contain false negatives. In the work presented in [8] it is not discussed how that method can cope with different configurations of the system, therefore it scores moderate there. On representation it scores the same as the method presented here since the way the results are presented is the same (ranked list of fault candidates). The tool used in [8] still requires observations to be inputted manually, but this could be automated and therefore it also scores good on automation. Finally, the major drawback of MBD is that building full scale models that capture the behavior and structure of the entire system is not feasible. This is also the reason why PMS chose not to continue with MBD.

It can be concluded that MBD scores better on accuracy and completeness than the method presented in this thesis. However, scaling MBD up to the entire system is not (yet) feasible and therefore another method has to be found. The method presented here is feasible on a much larger part of the system than that which is chosen in [8]. The method presented here also gives the same accuracy and completeness as the current practice of fault diagnosis. As major improvement, this method is almost fully automated, where the current practice requires a lot of user effort (which results in an increased time of diagnosis). As a last point it has been designed to handle the different configurations of the system with minimal user effort. Therefore one diagnostic system can be used to diagnose all different configurations of the system. This is also an advantage over MBD.

Another important aspect of the conclusion is the usability of this work for PMS. Therefore a survey has been held amongst people of different departments within PMS. The prototype and the validation results of this prototype have been presented to people from the field service department, from the service innovation department, from the test and integration department and from the development department. Then a survey was held amongst them in which they were asked to give their opinion on whether this method can improve fault diagnosis at PMS.

The results and the conclusion drawn from this survey can be found in paragraph 7.3.1 and 7.3.2. The general conclusion that is drawn is that the results are promising as a basis, but that further investigation is necessary to conclude that this method can be used to improve fault diagnosis on the entire system.

9 Future work and recommendations

In this chapter a number of recommendations will be described. These recommendations will be about ideas that might be worth investigating if implementing them would lead to better diagnostic capabilities. The recommendations have been divided into four paragraphs, the first paragraph discusses changes with regard to the method, the second paragraph with regard to the models and the third paragraph will discuss changes with regard to observation points. The last paragraph contains future work and recommendations that could not be categorized in any of the other three paragraphs.

The recommendations that will be described here are all with the current situation taken as starting point. This means that not all combinations of recommendations might be possible, so care must be taken when combining several recommendations.

9.1 Diagnostic Method

Here the future work and recommendations regarding the diagnostic method itself will be described.

9.1.1 Combining results of different diagnostic rules

Currently the system is diagnosed using one diagnostic rule. The result from this is a diagnosis where components are given a gradation of Healthy, Unknown, Possibly Broken or Broken. In the current prototype, there is no correlation between the diagnosis results of two rules. If in one rule a component has been diagnosed as definitely healthy then it is possible that in a second rule it is diagnosed as possibly broken.

Since the method only looks at persistent errors it can be stated that the healthiness of components does not change between test if nothing is replaced or repaired. Therefore it might be possible to profit from this knowledge by combining the diagnosis results from different diagnostic rules.

9.1.2 Allow statements to be derived

In some cases unexpected behavior is caused by something other than a hardware fault. Examples of these cases are when the configuration is wrong or when for instance a movement is blocked by the 3D model in the software. Those cases were considered out of scope for this project, but adding diagnostic capabilities to also detect these cases would be valuable for Field Service Engineers. To add these cases an extension to the method is necessary because these cases do not involve components. Therefore it should be investigated if the models can be extended such that it is also possible to derive conclusions about “statements”.

9.1.3 Allow actions to be done to generate observable information

Currently the method is passively monitoring signals that pass through the system. Extra observable information could be gathered if the diagnostic system is allowed to perform certain actions (call functions) such that extra observable information can be gathered. This does however raise questions about the safety and therefore it must be investigated first to see whether it could be allowed in a safe way. It should also be investigated if the extra observable information that could be gathered this way is useful to get a better accuracy or resolution from the diagnosis.

9.2 Diagnostic Models

Here the future work and recommendations regarding the diagnostic models will be described.

9.2.1 Allowing "virtual" observation points

The model rules consist of an observation part and a conclusion part. The observation part of the model rules consist of observation points and logical operations. The observation part of these model rules can get quite complex when a lot of observation points are used. When this happens, the observation part becomes difficult to read which makes it hard to maintain or extend the models. It also makes it easier to make mistakes.

Therefore we propose adding some "syntactical sugar" to the model rules where "virtual observation" points can be defined to replace part of the condition. An example of this is presented below. Suppose the following model rule exists:

```
Host_PC_movement_request_received AND Host_PC_movement_is_operational
          AND Host_PC_movement_request_forwarded AND (NOT
                  Geo_IPC_movement_request_received)
                                 =>
                 PossiblyBroken("Network components")
```

Then a virtual observation point called "`Host_PC_is_functioning`" can be defined as being a synonym for:

```
Host_PC_movement_request_received AND Host_PC_movement_is_operational
              AND Host_PC_movement_request_forwarded
```

The model rule would then become:

```
  Host_PC_is_functioning AND (NOT Geo_IPC_movement_request_received)
                                 =>
                 PossiblyBroken("Network components")
```

The impact might seem minimal in this example, but in the actual models there are model rules that contain more than ten observation points and for these model rules this syntactical sugar would really make the model rules more readable, making it easier to understand, modify and extend them.

9.2.2 Introducing a minimal time span for tests

Currently, when the diagnosis is started all observation points are reset. Then the user is asked to perform an action and after that the user can click that the test is finished. The period between the reset of the observation points and the click that the test is finished is called the diagnostic period and only during this time observations are gathered from the systems.

This might lead to cases where different observable behavior is captured when the user is performing the action really quick or when the user is performing the action at a slower speed. As an example, take the case where a message is sent over a network. As long as no confirmation of the message is received, the sending is retried for a fixed number of times. After this fixed number of retries, the sending of the message is given up and an observation point is signaled. Because time is involved in the sending of the message and waiting for the confirmation it might be the case that the diagnostic period is over before the maximal number of retries is reached. This means that in that case the observation point that indicates that the sending of the message has been given up is not signaled. However, when the diagnostic period is longer because the action is performed at a slower speed, the observation point is signaled. This difference in outcomes is not wanted because the physical state of the machine is the same in both cases (and thus the diagnosis result should be the same).

This unwanted behavior can be avoided if for the diagnostic period a minimal time span is defined. This could mean that for each diagnostic model a minimal time span should be defined that needs to be waited before the next model is checked.

9.2.3 Adding time or precedence to observations in the model rules

The model rules currently allow that conclusions about the healthiness of components are drawn using static observations that are gathered from the system. The observations that are gathered are whether or not an observation point is signaled. However, it might give extra information if also time information could be used or precedence between observation points. For example, model rules could specify that a certain observation point should be signaled before another observation point is signaled. Further investigation is necessary to see whether this could lead to better diagnostic conclusions.

9.2.4 Investigate tooling that can assist building models

Model rules can become quite complex when a lot of observations are used. This makes it easy to make mistakes when creating the model rules. Obviously, mistakes are not wanted because faults in the model rules might lead to faults in the diagnosis. Therefore it might be useful to investigate whether it is possible to create tooling that can assist in building model rules. Such tooling should then reduce the number of faults that are made when creating the models.

9.2.5 Investigate tooling that can validate models

In previous suggestions we have stated that it is realistic that faults are made when creating the diagnostic models. This might lead to incorrect diagnostic conclusions by the diagnostic engine. Therefore it might be worth to investigate if it is possible to create tooling to validate models. This tool might then contain a number of test cases where observation points are simulated and that the outcome of the diagnostic engine is compared to the given correct outcome. This tool could then be used to test models after they have been changed to check whether they are still correct for known situations.

9.3 Observation Points

Here the future work and recommendations regarding the observation points will be described.

9.3.1 Adding manual observation points

The observable behavior that can be used for the diagnosis is limited to information that can be traced from the software. This results in a decrease in diagnostic resolution in places where only hardware is available. For instance, the input module is connected by a cable to a hardware box and from there a cable goes through an interface board to the Host PC. This is all hardware so the current method cannot distinguish between a broken cable, a broken hardware board or a broken input module. By adding “manual” observation points, i.e. observation points where the user or FSE is asked to perform a measurement or to see whether an LED is on or not, extra observable information can be gathered from places where the software cannot get observable information from.

9.3.2 Include startup observations in diagnosis

When the diagnosis is started, all observation points are reset. This is done to get a fixed starting situation as is explained in paragraph 6.4.6. This means that any observable information that is gathered before the observation points are reset, is lost. This also includes any tests, such as the POST, which are only executed at the startup of the system. Since the POST contains valuable information that could improve the diagnosis, it might be worth to investigate whether it is possible to create a fixed starting situation by, for example, restarting the system and omit the resetting of observation points at the beginning of the diagnosis. This means that observable information gathered during startup can also be taken into the diagnosis.

9.4 Architecture of controlling software

This paragraph describes future work and recommendations about the architecture of the controlling software.

9.4.1 Introducing a "diagnostic mode" in the controlling software

One of the outcomes of the validation has been that the diagnostic capabilities of the diagnostic solution are seriously limited by the controlling software when it disables some movements. The controlling software disables movements when, for example, the POST detects that something is wrong with a movement and the movement is then disabled at the level of the Host PC. This means that all signals for that movement stop at the Host PC and that no observable information for that movement can be captured in the modules after the Host PC.

My suggestion is to introduce different behavior in the controlling software such that movements are no longer blocked at the Host PC level (and signals are thus no longer stopped at the Host PC level). Such behavior could be introduced in a separate functioning mode that could be called the "diagnostic mode". It does need to be investigated however what impact introducing this mode has on the controlling software.

9.4.2 Improve the logging in the controlling software

From discussions with FSE's it has become clear that the logging and the information in the event viewer does not contain understandable information for the beginning and medium experienced FSE. Improving the logging such that it contains more meaningful information (which is already being done at the moment for some subsystems) would help the FSE to quicker determine what the problem in the system is.

10 References

[1] M. Kurvers, *Project Plan*, Non-public document, Philips Medical Systems, 2007

[2] M. Kurvers, *Requirements Specification*, Non-public document, Philips Medical Systems, 2007

[3] S. Dash, V. Venkatasubramanian, *Challenges in the industrial applications of fault diagnostic systems*, Technical report, Purdue University, 2000

[4] J. de Kleer, B. Williams, *Diagnosing Multiple Faults,* Artificial Intelligence, Vol. 32, pp 97-130, April 1987

[5] V. Venkatasubramanian, R. Rengaswamy, K. Yin, S. Kavuri, *A review of process fault detection and diagnosis Part I: Quantitative model-based methods,* Computers and Chemical Engineering, Vol. 27, Nr. 3, pp 293-311, March 2003

[6] V. Venkatasubramanian, R. Rengaswamy, K. Yin, S. Kavuri, *A review of process fault detection and diagnosis Part II: Qualitative models and search strategies,* Computers and Chemical Engineering, Vol. 27, Nr. 3, pp 313-326, March 2003

[7] V. Venkatasubramanian, R. Rengaswamy, K. Yin, S. Kavuri, *A review of process fault detection and diagnosis Part III: Process history based methods,* Computers and Chemical Engineering, Vol. 27, Nr. 3, pp 327-346, March 2003

[8] W. Lindhoud, *Automated Fault Diagnosis at Philips Medical Systems,* Master's thesis, Delft University of Technology, June 2006

[9] R. Davis, W. Hamscher, *Model-based reasoning: Troubleshooting,* Exploring Artificial intelligence, H. Shrobe, E. Morgan, Kaufmann Publishers, San Francisco, CA, 1988

[10] B. Peischl, F. Wotawa, *Model-Based Diagnosis or Reasoning from First Principles,* IEEE Intelligent Systems, Vol. 18, Nr. 3, pp 32-37, May 2003

[11] D. Poole, *Representing Diagnosis Knowledge,* Annals of Mathematics and Artificial Intelligence, Vol. 11, Nr. 1-4, pp 33-50, 1994

[12] L. Console, D. Theseider Dupre, P. Torasso, *On the relationship between abduction and deduction,* Journal of Logic and Computation, Vol. 1, Nr. 5, pp 661-690, 1991

[13] R. Davis, *Diagnostic Reasoning Based on Structure and Behavior,* Artificial Intelligence, Vol. 24, Nr. 1-3, pp 347-410, December 1984

[14] B. Kaiser, P. Liggesmeyer, O. Mäckel, *A New Component Concept for Fault Trees,* Proceedings of the 8th Australian Workshop on Safety Critical Systems and Software, Vol. 33, pp 37-46, 2003

[15] M. Maurya, R. Rengaswamy, V. Venkatasubramanian, *Application of signed digraphs-based analysis for fault diagnosis of chemical process flowsheets,* Engineering Applications of Artificial Intelligence, Vol. 17, Nr. 5, pp 501-518, August 2004

[16] A. Lin*, A Hybrid Approach to Fault Diagnosis in Network and System Management,* HP Technical Report, HPL-98-20, 1998

[17] I. Rish, M. Brodie, S. Ma, *Efficient fault diagnosis using probing,* in proceedings of AAAI Spring Symposium on "Information Refinement and Revision for Decision Making: Modeling for Diagnostics, Prognostics, and Prediction", pp 16-23, March 2002

[18] P.H. Ibargüengoytia, L.E. Sucar, E. Morales, *A probabilistic Model Approach for Fault Diagnosis,* in proceedings of Mexican International Conference on Artificial Intelligence, pp 687-698, April 2000

[19] J. de Kleer, *Using crude probability estimates to guide diagnosis,* Artificial Intelligence, Vol. 45, Nr. 3, pp 381-391, October 1999

[20] A. van Gemund et al, *Lydia Project*, http://fdir.org/lydia/

[21] K. Czarnecki, U. Eisenecker, *Generative Programming – Methods, Tools and Applications*, Addison-Wesley, June 2000

Appendix A

Proof 1: Assumption that all components are working with given observations leads to a contradiction, hence one of the assumptions must be wrong.

1.	*“The power supply is working”*	assumption
2.	*“The switch is working”*	assumption
3.	*“The light bulb is working”*	assumption
4.	¬*“The light is on”*	observation
5.	“*the switch is on*”	observation
6.	*“The power supply is working”* → *“The system is powered”*	rule 1
7.	*“The switch is working”* → ((*“The system is powered”* ∧ *“The switch is on”*) ↔ *“The light bulb is powered”*)	rule 2
8.	*“The light bulb is working”* → (*“The light bulb is powered”* ↔ *“The light is on”*)	rule 3
9.	*“The system is powered”*	6 & 1
10.	(*“The system is powered”* ∧ *“The switch is on”*) ↔ *“The light bulb is powered”*	7 & 2
11.	*“The system is powered”* ∧ *“The switch is on*	9 & 5
12.	*“The light bulb is powered”*	10 & 11
13.	*“The light bulb is powered”* ↔ *“The light is on”*	8 & 3
14.	*“The light is on”*	13 & 12
15.	False	14 & 4

Proof 2: Removing the assumption that the power supply is working still gives a contradiction, hence the power supply is not a fault candidate.

~~1.~~	~~*“The power supply is working”*~~	~~assumption~~
2.	*“The switch is working”*	assumption
3.	*“The light bulb is working”*	assumption
4.	*“The 2nd light bulb is working”*	assumption
5.	¬*“The light is on”*	observation
6.	“*the switch is on*”	observation
7.	“*the 2nd light is on*”	observation
8.	*“The power supply is working”* → *“The system is powered”*	rule 1
9.	*“The switch is working”* → ((*“The system is powered”* ∧ *“The switch is on”*) ↔ *“The light bulb is powered”*)	rule 2
10.	*“The light bulb is working”* → (*“The light bulb is powered”* ↔ *“The light is on”*)	rule 3
11.	*“The 2nd light bulb is working”* → (*“The system is powered”* ↔ *“the 2nd light is on”*)	rule 4
12.	*“The system is powered”* ↔ *“the 2nd light is on”*	11 & 4
13.	*“The system is powered”*	12 & 7
14.	(*“The system is powered”* ∧	9 & 2

	“The switch is on”) ↔ *“The light bulb is powered”*	
15.	*“The system is powered”* ∧ *“The switch is on*	13 & 6
16.	*“The light bulb is powered”*	14 & 15
17.	*“The light bulb is powered”* ↔ *“The light is on”*	10 & 3
18.	*“The light is on”*	17 & 16
19.	False	18 & 5

Reasoning 3: Different way of modeling system as in Proof 2. Now removing the power supply no longer gives a contradiction, hence the power supply is a fault candidate.

~~1.~~	~~*“The power supply is working”*~~	~~assumption~~
2.	*“The switch is working”*	assumption
3.	*“The light bulb is working”*	assumption
4.	*“The 2nd light bulb is working”*	assumption
5.	¬*“The light is on”*	observation
6.	“*the switch is on*”	observation
7.	“*the 2nd light is on*”	observation
8.	*“The power supply is working”* → *“The system is powered”*	rule 1
9.	*“The switch is working”* → ((*“The system is powered”* ∧ *“The switch is on”*) ↔ *“The light bulb is powered”*)	rule 2
10.	*“The light bulb is working”* → (*“The light bulb is powered”* ↔ *“The light is on”*)	rule 3
11.	*“The power supply is working”* → *“The 2nd light bulb is powered”*	rule 4
12.	*“The 2nd light bulb is working”* → (*“The 2nd light bulb is powered”* ↔ *“the 2nd light is on”*)	rule 5
13.	(*“The system is powered”* ∧ *“The switch is on”*) ↔ *“The light bulb is powered”*	9 & 2
14.	*“The light bulb is powered”* ↔ *“The light is on”*	10 & 3
15.	*“The 2nd light bulb is powered”* ↔ *“the 2nd light is on”*	12 & 4
16.	*“The 2nd light bulb is powered”*	15 & 7
17.		
18.		
19.		

Now from this point on, to get a contradiction we need the *“The light is on”* to be true. This is true if and only if *“the light bulb is powered”* is true. For that to be true, we need *“The system is powered”* to be true which (in this model) we can only get by the assumption that *“The power supply is working”*.

This immediately gives an indication why, after removing the assumption that *“the power supply is working”*, we do not get a contradiction in the model of reasoning 3, but we do get a contradiction in the model of proof 2. In the model of proof 2 the assumption that *“The power supply is working”* leads to the conclusion that the whole system is powered (including light bulb 1 and light bulb 2). In the model of reasoning 3 the assumption that *“The power supply is working”* leads to two separate conclusions, namely that *“The* *system is powered”* and that *“The 2nd light bulb is powered”*. This effectively removes the “link” between the power nets of the first and second light bulb, leading to a model in which it is no longer possible to conclude that the whole power net must be working if one of the two light bulbs is powered. (This means that the model of proof 2 is probably closer to the reality than the model of reasoning 3).

Appendix B

Survey improvement of fault diagnosis with new method

Introduction

For my graduation project at Philips Medical Systems I've created a method for improving fault diagnosis at PMS. The goal of the method was to help beginning and medium experienced engineers to quicker narrow down the search area for a fault and to reduce the amount of non-faulty hardware that is replaced. Currently a prototype implementation has been built on a selected part of the whole system and a verification of this prototype is done. More information about the verification can be found in the documents "Verification Plan" and 'Verification Results". A quick guide to give an idea how the prototype diagnostic tool works can be found in the document "User guide for diagnostic method prototype".

Purpose of survey

To finish the project a conclusion needs to be drawn whether this method has succeeded in improving fault diagnosis in the ways described in the introduction. It is hard to quantify this conclusion based upon figures. The reason for this is that no exact figures are available for the time a FSE spends on average on the problems described in the test cases that were used to test the prototype on. It is also not known exactly how much components would have been replaced by the FSE without using this tool and with using this tool. This prototype cannot be verified on real problems in the field because the software is not officially approved (as is needed by regulations of the FDA). Therefore it has been decided that a survey will be done in which people of PMS give their opinion of the usability of the prototype and if it satisfies the goals that were set out. These opinions and the arguments that have lead to these opinions will be used in the conclusion whether this method can improve fault diagnosis at PMS.

Survey questions

The target audience for this diagnostic tool is the beginning or medium experienced Field Service Engineer. All questions are asked with the target audience in mind. It would be very appreciated if the answers are motivated (can be done generally per section at the "remarks" section).

The way the tests are executed

- Is the selection of the tests clear?
- Is the action text clear?

Remarks:

The presentation of the diagnosis results

- Do the blocks that represent the different parts or modules of the system give a good indication in which part of the system the fault is?

- Is the listing of components per block clear and useful in determining what could be the faulty component?

Remarks:

The results of the verification
- Is the number of failed tests acceptable?
- Are the current results of the verification acceptable to create a full scale diagnostic application based on this method?

Remarks:

The usability of this method to improve fault diagnosis
- Can this method help to reduce the number of non faulty hardware that is replaced?
- Can this method help to quicker locate a fault in a system?
- Is this method a useful addition to the already available diagnostic methods (POST, logging) in that it adds extra diagnostic capabilities?
- Is this approach suitable for use in the system of PMS?

Remarks: