RADBOUD UNIVERSITY NIJMEGEN

FACULTY OF SCIENCE

A pragmatic approach to active failure diagnosis of discrete-event systems

AN APPLICATION IN THE FIELD OF MACHINE CONTROL APPLICATIONS

MASTER THESIS SOFTWARE SCIENCE

Author:
Niels OKKER

Supervisor Radboud:
prof. dr. Jozef HOOMAN

Supervisor ESI (TNO):
dr. Jacques VERRIET

Second reader Radboud:
dr. Daniel STRÜBER

15th February 2021

A pragmatic approach to active failure diagnosis of discrete-event systems

An application in the field of machine control applications

Niels Okker

Abstract

Industrial systems have to support an ever-increasing number of possible configurations. Machine control applications that determine the behaviour of these systems become increasingly complex because they need to deal with these configurations. The complexity of these systems and their machine control applications make it difficult to diagnose failures manually. In this thesis, we present a practical approach that detects and diagnoses failures in these systems. The method uses domain knowledge, such as machine control application specifications, to create a model that can diagnose failures in a discrete-event system.

Our method actively diagnoses a system based on the concepts of partial observability of discrete-event systems and system state estimation using a *diagnoser* model. The method detects failures during system operation. After the occurrence of a failure, an *active diagnoser* stops the system and starts active diagnosis. The active diagnoser repeatedly actuates the system and listens to the system's sensor values to exclude potential faults as potential root-cause. Based on the resulting system output sequence, the diagnoser reduces the number of potential root-causes, ideally to obtain a single root-cause. We use a prototype to demonstrate that these active diagnosers can diagnose industrial systems after detecting failures.

Contents

Acknowledgements

During the creation of this thesis, I learned a lot about conducting scientific research, anomaly detection/diagnosis and machine control applications. I have received a great deal of support, which helped me write this thesis.

I would like to thank my ESI (TNO) supervisor, Jacques Verriet, for his insights, guidance, and support. I learned a lot during our fruitful discussions. I would like to thank my Radboud supervisor, Jozef Hooman, for his advice and feedback during all research stages. I want to thank Daniel Strüber for being the second reader and providing feedback.

Furthermore, I would like to thank everyone involved in the Machinaide project from TNO and Cordis Automation. They provided technical support and gave helpful feedback. Finally, I would like to thank my family for their support and love.

The research is carried out as part of the ITEA3 18030 MACHINAIDE project under the responsibility of ESI (TNO) with Cordis Automation as carrying partner. The MACHINAIDE research is supported by the Netherlands Organisation for Applied Scientific Research TNO and Netherlands Ministry of Economic Affairs.

1 Introduction

Systems in the high-tech industry have to deal with an ever-increasing number of potential configurations due to the market's demand [1]. To deal with this demand, machine control applications controlling these industrial systems become increasingly complex [2]. New techniques such as digital twins can analyse the behaviour of these machines. One application of digital twins is to detect anomalous behaviour of these industrial systems. The digital twin gathers data from the machine and uses models to determine if the shown behaviour is abnormal. A next step is to determine the cause of abnormal behaviour. The root cause of abnormal behaviour, such as a broken component, should be dealt with promptly to prevent potential unplanned production interruption, as this results in cost increase [3]. Due to the machines' complexity, it is a time-consuming job to determine the root cause of an anomaly. This thesis focuses on methods that can generically diagnose detected anomalies in industrial systems controlled by machine control applications.

The rest of this chapter introduces the thesis. Section 1.1 discusses the context of the thesis. Sections 1.2 and 1.3 describe the root cause analysis problem and the objective. Section 1.4 formulates the research questions. Section 1.5 describes the approach and finally, a reading guide in Section 1.6 shows the structure of the remaining chapters of this thesis.

1.1 Context

This thesis is carried out as part of the Machinaide project. The Machinaide project is a European cooperation of four countries and 19 partners. Together with Additive Industries, Lely, TU/e, KE-works and Cordis Automation, ESI (TNO) forms the Dutch part of the project. The project's goal is to improve industrial machines and involved processes by applying and improving knowledge-based services [4]. The usage of digital twins of machine control applications is such a knowledge-based service.

1.2 Problem statement

Cordis Suite, developed by Cordis Automation, is a toolset for developing machine control applications [5]. It lets users specify their Machine Control Applications (MCAs) using a UML subset and generate code for various Programmable Logic Controllers (PLCs). It is critical to determine if systems controlled by these applications function as expected and act upon detected anomalies to prevent potential unplanned production interruptions.

MCA specifications, such as those created with Cordis Suite, are currently not used to diagnose anomalies. Cordis MCA specifications contain class diagrams that model the hierarchy of the various components and contain behavioural diagrams, such as state machine diagrams, that model the behaviour. In general, MCA specifications determine the machine's behaviour. It is unknown how an MCA specification can help interpret the machine's behaviour.

An MCA specification gives information on the system behaviour; it explains what the actuators of a system should do based on the sensor values received. Interpreting the machine's behaviour means to try to reason which fault in the system could explain the shown behaviour.

1.3 Objective

This thesis aims to define (semi-)automated methods that help diagnose anomalies in industrial systems controlled by machine control applications. The methods should use readily available domain knowledge, e.g., the machine control application specification and the execution data (such as state machine transitions and sensor data). There are various kinds of anomalies, and as many approaches to diagnosing them. Therefore, the objective is to deal with a part of the problem stated above. The main focus is on anomalies caused by hardware faults, such as malfunctioning or broken components, since this type of failure is common in industry and an active research topic [2]. While software bugs are also a potential root cause, we assume that the MCA specification models are correct and the code generation to PLC code is bug-free.

1.4 Research questions

The thesis will answer the following research questions to reach the project goal:

Q1 What are existing diagnostic methods for the diagnosis of anomalies in industrial systems?

Q2 Which quality criteria determine how well diagnostic methods fit in the context of machine control applications?

Q3 Which method for diagnosing detected anomalies of industrial systems found by answering Research question Q1 is most fitting according to the criteria of Research question Q2?

Q4 How can the method found by answering Research question Q3 be refined to use machine control application specifications and data from systems in the field?

Q5 Which diagnostic information of the abnormal behaviour should a method present to system developers?

Q6 Does the refined method of anomaly diagnosis found by answering Research question Q4 satisfy the criteria identified in Research question Q2?

The answer to the first research question provides an overview of existing methods of failure diagnosis that are generic and knowledge-driven. The answer to the second research question determines which criteria are relevant to finding the method that fits the context best. The answer to the third research question uses the criteria of Research question Q2 to compare the methods identified by answering Research question Q1. The answer to the fourth question looks for a way to refine the method identified by Research question Q3 and apply it to systems controlled by machine control applications. The answer to the fifth question determines how the diagnosis of the refined method should be presented to a system developer. The answer to the final question validates if the refined method is a good fit to the quality criteria identified by answering Research question Q2.

Together the research questions should provide sufficient information to answer the main research question: **How can detected anomalies in systems controlled by machine control applications be diagnosed in a generic and knowledge-driven way?**

1.5 Approach

A first step towards the objective is to find methods that, based on domain-knowledge and an MCA specification:

1. Generate a model to diagnose found anomalies,

2. Use the model to diagnose failures in a system,

3. Present the diagnosis to a system engineer.

The second step is to determine a set of criteria to compare these methods. Based on the criteria, the third step should compare the found methods to find the most suitable technique for MCA-controlled systems by applying them to the same example. The fourth step is to determine if we can improve the best-fitting technique by making the technique specific to MCA-controlled systems.

The final step is to implement a prototype and apply it to a real use case to validate the method in practice. Lely or Additive Industries, both partners in the Machinaide project, are candidates for providing such a use case. An alternative is to use Cordis Suite's toolset to create a similar machine control application for the sole purpose of validating the found method.

1.6 Reading guide

This section gives an overview of each chapter. Chapter 2 shows the related work. It outlines various methods to detect and diagnose failures in industrial systems. Chapter 3 explains the preliminaries of the thesis. It introduces discrete-event systems, the discrete-event system specification toolset of Cordis Suite and discrete-event system diagnostics. Chapter 4 draws a comparison between Bayesian networks and discrete-event system diagnostics based on a set of criteria. Chapter 5 introduces a pragmatic approach based on discrete-event system diagnostics. Chapter 6 describes a prototype used to demonstrate the proposed method. Chapter 7 summarises the answers to the research questions, explains the thesis' main contributions, discusses the results and suggests future work.

2 Related work

This section gives an overview of the approaches to detecting and diagnosing anomalies found in the literature. The chapter is structured based on the topics of the related literature. Section 2.1 defines several concepts used throughout the related work chapter and the rest of the thesis. Section 2.2 describes review papers that compare several methods of anomaly diagnosis. Section 2.3 discusses techniques using Bayesian networks. Sections 2.4 and 2.5 present papers using decision trees and discrete-event system models to diagnose anomalies.

The anomaly diagnosis methods described in this chapter answer Research question Q1: 'What are existing diagnostic methods for the diagnosis of anomalies in industrial systems?'

2.1 Definition of concepts

Before discussing the related work, this section defines the concepts used throughout the remainder of the chapter and thesis. The definitions are based on Avižienis *et al.* [6], but some are fitted to the domain of machine control applications. Definitions are shown in **Bold** font.

Definition 2.1.1 (System). A **system** is an entity that communicates with other entities, i.e., other systems such as software, hardware, humans and the physical world. The other entities are also called the **environment** of the system. The interface or common frontier between the system and its environment is called the **system boundary**. What a system is intended to do is called its **function**. The **behaviour** of the system is what it does to perform its function.

Definition 2.1.2 (System structure). The **structure** of a system enables the system to generate behaviour. A system is composed of a set of **components**, where each component is another system. Components that cannot be split down into smaller components are called **atomic**.

Definition 2.1.3 (System state). The **system state** is the combination of all relevant information about the system at a point in time. The **external state** of a system is the observable part of the system state as perceived by users. The **internal state** is the part of the system state that users cannot observe. The **service** delivered by a system is its behaviour as perceived by its users. The service is a sequence of external system states.

In this thesis' context, we assume that a system is a cyber-physical system [7] consisting of several components. Such systems have varying functions, such as manufacturing products. Most of the components are physical systems that combine sensors and actuators to achieve a function. For example, a lift combines a cylinder with sensors that determine the cylinder's position. The function of the lift is to move objects vertically. A **controller** is a software component that commands the physical components based on their sensor measurements and human input. System engineers can design controllers, also called machine control applications. The controller's function is to give the physical components their desired behaviour and let them work together. An important observation is that the controller cannot observe the other components' entire state; it can only see their external state. Furthermore, the controller cannot observe the entire state of the physical world either.

Definition 2.1.4 (Failure). A system delivers **correct service** when its service implements the system function. A **service failure**, abbreviated to **failure**, is an event that occurs when the delivered function deviates from correct service. Since the service is a sequence of external

states, a failure means that some states in the sequence deviate from the correct service state. This state deviation is called an **error**. An error is a part of the system state that may lead to a failure. It might also be possible for the error to be part of the internal state. A **fault** is the cause of an error, also called a **root-cause**. An **anomaly** is a synonym for an error; it is a deviation from the system's normal state.

Definition 2.1.5 (**Diagnosis**)**.** In the context of this thesis, a detected anomaly or failure is the starting point. The goal is to determine the underlying fault that caused the deviation in system behaviour, making the system no longer perform its function. **Failure diagnosis** is the activity of determining the underlying fault(s) that through an error resulted in a failure. **Root cause analysis** is a synonym for failure diagnosis.

Example 2.1.1. As an example, a conveyor belt actuator may break down. The broken conveyor belt is a fault in the physical system. When the conveyor belt gets a command to start, this leads to an error; it does not start. When this error is observable from outside the system, this error leads to a failure; the system no longer performs its function.

2.2 Review papers

Solé *et al.* [8] review different methods and models to analyse the root cause of anomalies. They describe the typical approach of creating a model and then inferring potential root causes by adding observations to the model. Next, they review the models based on their complexity, how they can be generated and their advantages and disadvantages regarding the inference of the root cause based on observations. The paper is a good starting point to determine the potential approaches of diagnosis anomalies. The paper describes various deterministic models such as decision trees and probabilistic models such as Bayesian networks. The methods that are semi-automatic and use domain knowledge are most relevant in our context and are discussed in the following sections.

2.3 Bayesian networks

A Bayesian network is model that describes a set of variables and their conditional probabilities [9]. Each node in the graph corresponds to a random variable that depends on other random variables. Edges describe this dependency. Bayesian networks find the root cause of anomalies by describing faults and their effect as nodes, and their causal relations as edges. Based on observations on the system from a service engineer, the probability of fault-nodes will change, and the model can diagnose the most probable root-cause. Bayesian networks are a Directed Acyclic Graph (DAG) because a node cannot depend on itself.

Typically, the random variables used in Bayesian networks are discrete and finite. In some cases, the potential values of variables are not a finite set, i.e., they are continuous variables. The inference of Bayesian networks that use continuous variables are computationally expensive [10]. There are various ways to prevent such problems. Some methods discretise the continuous variables by dividing the range of the continuous variable [9]. In literature there are various approaches and frameworks that allow the usage of continuous variables [11] [10] [12].

In the following paragraphs we will describe the different subtypes of Bayesian networks the are used in the diagnosis context. Then, we will discuss how different approaches use these subtypes of Bayesian networks, varying from specifying Bayesian network based on physical causality to more abstract types of networks. Then we will discuss how Bayesian networks can be created from fault trees, since Bayesian networks can be more flexible in some scenarios. Finally we will show a paper that applies Bayesian network to diagnosing software bugs, since this might be relevant for machine control applications.

Types of networks Various types of Bayesian networks exist to cope with shortcomings of the basic Bayesian network. The required acyclic structure of Bayesian networks is such a shortcoming. By definition, DAG's do not contain any loops. In practice, the modelling of dependencies of a system does result in loops. Think about a typical machine with a control loop. Actuators influence the world. Consequently, sensors read this change and then influence the actuators. In case the root cause of a phenomenon in the real world should be diagnosed, this causes an issue with basic Bayesian networks.

Dynamic Bayesian networks can deal with loops by modelling a domain over time. A Bayesian network is created for each time slice [13]. Dynamic Bayesian networks model dependencies over time as edges between time slices. In industrial systems, this could mean that sensors directly influence the world in one time slice and actuators influence what the sensor reads in the next time slice. Some approaches use dynamic Bayesian networks to reason about the most probable cause in the past for an observation made in the present [14].

Physical components modelled in Bayesian networks There are various ways to model a domain into a Bayesian network to perform anomaly detection and diagnosis.

Barbini *et al.* [2] use Bayesian networks to reason about the health of components of industrial systems. Based on the input and output variables of components, and the causal relations between these variables, the used method determines the components' health. The networks consist of components, each with input variables, output variables and health nodes. The output variables depend on the component's health and input variables, while the input variables depend on the output variables of other components. The output variables contain the domain knowledge of what faulty behaviour is. By adding observations, the Bayesian network infers the health of nodes.

More fundamental work by de Kleer and Kurien [15], was the first to introduce this model-based approach with components as a basis. One notable assumption of the component-based approach is that it bases the causal relation between components on their physical connection. One component influences the another component by providing power, water or any other physical resource. In machine control applications, the control logic can also be the causal relation between components; based on the state of components, the software logic within the machine control application decides other components' behaviour.

Borth and von Hasseln [16] describe an approach to model complex industrial systems using Bayesian networks by using the system specification. According to the authors, data-driven approaches often lack the required data and are therefore an impractical approach, while manual creation of Bayesian network is often infeasible. Instead, they propose an approach that uses the fact that often a lot of the same components are used in an industrial system. By creating small components based on requirements, these can be assembled by connecting the input and output of components. If time plays a role, dynamic Bayesian networks are the proposed solution.

Provan [17] shows the commonality of various temporal and atemporal models, including Bayesian networks. The shown example of a Bayesian network is similar to the models used by Barbini *et al.* [2]. The paper uses a water tank as an example. In the shown network, the amount of water is conditionally dependent on the input flow. The amount of water is a dependency of the output valve of the water tank. The method adds additional nodes for potential faults in the system. One such node is leakage of the tank. The method can infer if the tank leaks based on the input flow, the amount of water in the tank and the output flow. The paper introduces a model that can generate both temporal and atemporal models.

Causal dependencies of variables in a Bayesian network Other approaches abstract away from the physical structure and only capture the causal relationship between variables within a system. Cai *et al.* [18] use this more data-driven approach. Based on historical data, it is possible to determine what sensor values indicate an anomaly. System engineers can use the

model to diagnose the root cause when the anomaly is known. The paper describes which types of Bayesian networks exist within the data-driven approach.

Weidl *et al.* [19] apply Bayesian networks to an industrial process. The component-based Bayesian network is called an Object-Oriented Bayesian Network (OOBN) in the paper. This type of Bayesian networks can reason about faults on different levels. OOBN use low-level networks as a single node in more high-level networks. The object-oriented network is a hierarchical description of a system. The models typically consist of three layers, the cause, the effect and the failure. Based on the failure, and observed effects, the original cause can be diagnosed. The final step proposed by the paper shows the expert how to correct the diagnosed faults.

Bayesian network generation from fault trees Perreault *et al.* [20] use fault trees as a basis to create Continuous-Time Bayesian Networks (CTBN). While the usage of continuous-time is out of scope for machine control applications, the generation from a fault tree is an appealing approach. For the Bayesian network generation, the method assumes that a fault tree exists that describes causes and effects. The first step in generating the network is to prune the fault tree. The step removes all redundant-nodes. The next step ensures that the tree does not contain duplicate nodes. In the last step, the method converts the fault tree into a Bayesian network. The variables of the fault tree become a node in the Bayesian network. The method removes the logic gates from the fault tree. The variables connected by a logic gate, now directly get an edge in the Bayesian network. Next, Perreault *et al.* [20] describe how to parametrise the Bayesian network for the various logic gates.

Mascaro *et al.* [14] describe Bayesian networks that do not use the physical layout. Their method uses historical vessel information to determine if the behaviour of a vessel is abnormal. It uses both a static Bayesian network and a dynamic Bayesian network. A similar model could be created for machine control applications. In the paper, the model is not used to diagnose the root cause of the anomaly.

Software in Bayesian networks While diagnosing anomalies caused by software bugs is out-of-scope, some methods diagnose root causes of software bugs in Bayesian networks. Yu *et al.* [21] generate Bayesian networks from source code. The method adds the results of software tests as observations to the model. The observations eventually lead to the suspiciousness of lines of code. If the code generation of machine control application seems to be a primary cause of anomalies, the method is an inspiring approach. The state machine diagrams created in an MCA specification are a potential substitute for source code. The method could then diagnose the correctness of models created in the MCA specification. The method should then look at suspicious transitions or states instead of lines of code.

2.4 Decision trees

A decision tree is a deterministic model that, when used in the context of failure diagnosis, shows the process of determining the fault, based on the observations made [22]. This process starts at a single root node. The nodes give a choice down to the leaves. Each node is a function that determines the next node down the tree. The leaves describe which fault is most probable based on the choices made. The nodes in the tree can either be Boolean functions or discrete functions [23]. Boolean function give a choice between two cases, while discrete functions give a choice between multiple cases. Decision trees used to classify a root cause are often called fault trees [24].

The generation of a decision tree often involves machine learning, such as using artificial neural networks or support vector machines [25]. A popular approach to learning such a decision tree is a top-down induction approach by Wahl and Sheppard [24]. The top-down approach tries

to partition the data, step by step, by splitting them on a variable. The greedy approach iteratively tries to find a variable, that gives the largest information gain at each decision node down to the root nodes.

In the following paragraphs we will discuss the found diagnostic methods using decision trees in literature. In the first paragraph we will discuss methods that use domain knowledge. In the following two paragraphs we will discuss literature that show how Bayesian networks can be translated into decision trees and vice versa. Both models have their advantages and based on the specific case, one of the two models could be preferable. The advantages of both methods are discussed next. Finally we show a relevant paper that uses a reasoning framework based on decision trees.

Generating decision trees using domain knowledge Various approaches integrate domain knowledge into the learning process, such as using ontologies [26] [27]. Iqbal *et al.* [28] add feature importance as domain knowledge to the learner. They apply the resulting framework on a bank loan screening problem. The 20 attributes that determine if someone gets a loan, each get a weighting by an expert. Next, the method learns the model, based on the data and expert weighting. A similar approach could diagnose detected anomalies if an expect could determine essential variables beforehand.

Abdelhalim *et al.* [29] use a rule-based approach for the generation of decision trees. Based on a rule-set (most likely generated from data), the method generates a decision tree that uses attribute effectiveness to determine which choice should be made first in the tree. The method places the variable that is most influential in determining the right class of anomalies first. If variables are equal in terms of attribute effectiveness, several additional steps describe how to determine the best choice in the tree first. The rule-based approach's idea is that the rules are more straightforward to adapt than a created decision tree when the data changes.

Generating decision trees from Bayesian networks Wahl and Sheppard [24] discuss several approaches that generate decision trees from Bayesian networks. The first approach creates a data set from the Bayesian network. For each fault in the Bayesian network, samples are created based on the Bayesian network's probability tables. Next, the method trains the decision tree using the induced data set. This approach is called induction by forward sampling. Two other approaches try to select variables in the Bayesian network based on other measurements such as KL-Divergence and MEU. The two final approaches are novel inventions of Wahl and Sheppard [24] and are heuristics based on the Bayesian network's dependency structure.

Generating Bayesian networks from decision trees The other way around, learning a Bayesian network from a decision tree is also discussed in the literature. Strasser and Sheppard [30] created bipartite Bayesian networks, which are networks that only contain root cause and observable effects from a fault tree. Strasser and Sheppard [30] describe an algorithm that first adds the leaves (the faults) as nodes to the Bayesian network. Next for each test in the decision tree, the method adds a node to the tree. The method adds an edge to the Bayesian network for each connection between a test and a fault. If Cordis Suite clients use decision trees as a method to determine faults, this method can create Bayesian networks from that information. It is interesting to create Bayesian networks because they have the advantage that there is no required order to test the system's properties. If the first test in a decision tree is not possible to conduct, then it is impossible to use a decision tree. The lack of choice is not a problem with Bayesian networks.

Comparing Bayesian networks and decision trees Bayat *et al.* [31] compare learning a Bayesian network against learning a decision tree from a medical data set. The authors use the models to diagnose which patients get on a transplant waiting list first. While the medical

data set is not comparable to industrial systems, the comparison between the two approaches gives insight into their usage. Bayesian networks give a clear insight into the causal relationship between variables; they give an overview of all variables and their relation. The decision tree is more convenient to interpret by experts. The conclusion of the comparison between Bayesian networks and decision trees is that the models are complementary.

Reasoning framework based on decision trees Sleuters *et al.* [32] describe an analysis approach for large scale internet-of-things (IoT) systems. The method uses a domain model consisting of eight domain-specific languages to describe the desired behaviour of a system. The created model, generated from the DSLs, serves as a digital twin. The approach feeds the data generated from the physical IoT-system to the digital twin. The physical system and digital twin run in parallel. Since both systems receive the same sensor input, the method can compare their behaviour. The approach is applied to an elaborate lighting system to compare the digital twin to the actual lighting behaviour. Using a comparator that processed both systems' behaviour, differences in behaviour, i.e., anomalies, can be detected. A reasoning framework, most comparable with a decision tree, is used for root-cause analysis.

2.5 Discrete-event system diagnostics

Discrete-event system diagnostics is an approach to detect and diagnose anomalies in systems that typically fall in the category of discrete-event systems (DESs). While the name discrete-event system diagnostics is rather generic, this is a specific approach different from Bayesian networks and decision trees. The latter two can also diagnose other types of systems. Systems fall in the discrete-event category when they have a discrete state space, i.e., have a finite number of possible (system) states, and switch states in an event-driven way. Computer networks are typical systems that fall in the DES category [33]. Continuous-variable systems can also be modelled as a DES when the system is abstracted to a discrete level [34].

DES diagnostic models, not to be confused with models describing discrete-event systems in general, model the entire system as one or more automata, with the specific purpose to diagnose failures in a system. We will call the models used for diagnostics **DES diagnostic models** to prevent confusion with a model describing a system's behaviour. The DES diagnostic model approach assumes that a controller, a software component, is connected to a set of physical components with sensors and actuators. The physical components together are called the *plant*. The controller can observe the plant based on the sensor values it receives. The controller gives action commands to the plant, based on the sensor values obtained from the plant.

Not necessarily all plant properties are observable for a controller; a broken actuator or sensor is not always trivially detectable by the controller. Only if sensors can detect a broken component, the controller knows. The general idea behind DES-model diagnostics is determining the faults, assumed to be unobservable events, from observable events.

Diagnostics using a model including faulty behaviour Sampath *et al.* [33] use finite state machines that contain both the desired behaviour and failure behaviour of a system under consideration. A controller automaton and several physical component automata form the full system automaton. The component models describe the physical components' behaviour and contain unobservable transitions, such as faults. The controller model describes the control logic between physical components and is assumed to be fault-free. As the controller cannot know in which state the system is due to the unobservable behaviour, the approach creates a diagnoser from the full system model [34]. The diagnoser estimates the system state and can detect and diagnose failures.

Other authors improved the original model of Sampath *et al.* [35] in various ways. Some approaches use a distributed modelling structure, to prevent the computation complexity of a

monolithic model [34]. Others include timing information in the model [36]. Zhao *et al.* [37] describe a minimum diagnosis approach, which focuses on more probable diagnoses. The method uses only minimal fault sets and ignores supersets that are improbable to happen. This heuristic avoids the complexity of the original model.

Diagnostics using a fault-free model DES diagnostic model approaches using a model that includes the faulty behaviour can only detect and localise modelled faults. A fault-free model does not have this limitation. The principle behind the fault-free model approach is to compare the system under consideration against the fault-free model. If the system shows behaviour that is not part of the fault-free model, a fault occurred in the system [36].

Roth *et al.* [36] describe an approach using a fault-free model to diagnose a system. The used automaton is called a Non-Deterministic Autonomous Automaton with Output (NDAAO). The method learns the automaton by observing event sequences, called *paths*, of the system during a failure-free run. An evaluator that compares the behaviour system against the model uses the NDAAO as the fault-free model. During operation, the system feeds data to the evaluator. The evaluator tries to reproduce the output. If the model cannot represent the output sequence, the evaluator detects a failure. The model's alphabet consists of input-output vectors (I/O-vectors); where I/O-vectors consist of sensor values and controller outputs (action commands). The I/O-vectors are also the basis for failure diagnosis. Based on which sensor-values or controller outputs are unexpected, it gives an insight which component could contain a fault.

Moreira and Lesage [38] continue the work of Roth *et al.* [36] but use a new model. The new model is more effective than the previous model because it prevents excessive language that is possible in NDAAO's. During learning, each transition in the model also remembers to which learned paths it belongs. When checking a system's behaviour, it is not only checked if the observed path can be reproduced in the model, but it is also checked if paths associated with a transition are coherent with the observed path. The approach is similar because it also requires one first to identify the fault-free model and afterwards use I/O-vectors in the system to diagnose the fault.

A master thesis recently written at TNO describes a method that internally uses fault-free models. Jansen [39] uses runtime verification to determine abnormal behaviour. The method lets domain experts describe rules over a particular scope, such as that an event should occur every ten seconds. The method uses state machines internally to determine violations of the rules. The method also uses MCA specifications as a data source. Therefore we can use information about the extraction of data from MCA specifications in this thesis.

Supervisory control synthesis Supervisory Control Synthesis (SCS) is related to the diagnosis using DES diagnostic models. In general, SCS is a formal method to create controllers for complex systems. The controllers restrict the system's possible behaviour. The idea is to take the models of all physical components of the system and compose them to obtain the plant's uncontrolled behaviour. SCS algorithms synthesise controllers based on the potential behaviour of the plant and the requirements. By design, the algorithm also creates a controller that is non-blocking, controllable and maximally permissive [40].

Reijnen *et al.* [40] and Reijnen *et al.* [41] apply supervisory control synthesis to a waterway lock and a movable bridge, respectively. Fault-free DES-model approaches could apply the synthesised controllers as a fault-free model.

There also exist supervisory controllers that cope with faults in the plant. Reijnen *et al.* [42] describe an approach that creates fault-tolerant supervisory controllers. The approach first synthesises a supervisory controller based on the requirements and component models when the system operates nominally. When the supervisory controller operates as expected, the next step is to incorporate the component models' faulty behaviour. A diagnoser, based on the system model including faults, determines the faults of the system. The faults detected by the diagnoser

are a special kind of sensor inputs to the controller. As preparation for the fault-tolerant controller, the requirements should differentiate between normal behaviour and behaviour after the occurrence of a fault. Likewise, plant models should differentiate the behaviour of the plant, based on the occurrence of faults. With the changed plant models and requirements, the algorithm creates a supervisory controller that changes its behaviour after a fault, to satisfy the requirements corresponding to the fault.

Although the approaches above can already cope with large systems, similar to DES diagnostic models, there are methods to improve supervisory control synthesis's scalability. Goorden *et al.* [43] prevent the creation of the entire system model, by determining which requirements apply to which part of the plant's components. The method uses a tree structure with the single components on the lowest layer. Requirements that apply to multiple components are on layers above. The synchronous composition of all models in the tree together forms the controller.

Modular DES diagnostics models A modular or component-based DES diagnostic model approach uses the system's structure to reduce the computational complexity of diagnosing a system. These approaches use local diagnosers for each component. The local diagnosers send their verdict to a central coordinator. The approach fits MCA applications well because MCA specification models are typically component-based (both logical and physical components). Debouk *et al.* [44] describe an approach that uses component models that include faulty, unobservable transitions, similar to Sampath *et al.* [33]. The approach creates a diagnoser for each component that contains fault transitions. The local diagnoser sends a notice to the coordinator after it finds a fault. The coordinator declares that a fault has happened if any local diagnoser declares that a fault occurred; the coordinator forwards the message. The primary assumption to diagnose the system is that the local models regularly take transitions. If a fault occurs in a local physical component, but the local diagnoser never executes transitions, it cannot determine if faults occur.

Cabral *et al.* [45] use the same structure but use different local diagnosers. The method uses fault-free Binary Petri-nets to diagnose faults in components. The local Petri-nets directly connect to a global fault state. When a local diagnoser detects a fault, it fires a transition from that local diagnoser to the global fault state (similar to the coordinator in Debouk *et al.* [44]). The Petri-nets start in the initial system state. Transitions in the model lead to multiple states based on the potential states of the original model. If the system executes an event that is not possible in a fault-free model's potential state, then that state is removed from the list of potential states. This way, if there are no potential states left in the local model, a fault must have occurred. The local diagnoser, therefore, executes the transition to the global fault state.

Active diagnosis The diagnostics considered in the previous paragraphs is passive diagnosis. An external entity monitors the behaviour of the system without changing the behaviour of the system. Active diagnosis changes the behaviour of the system to detect and diagnose faults. Sampath *et al.* [46] describe a method to synthesise controllers that adhere to a set of requirements and make the controlled systems diagnosable. A proposed two-step algorithm, using supervisory control synthesis, restricts the actions of the controller such that the system always remains diagnosable.

Chanthery and Pencolé [47] find the approach of Sampath *et al.* [46] too restrictive in terms of system behaviour. Instead, they interpret the diagnosis problem as a planning problem. Their approach tries to find all admissible test plans to determine a fault has occurred. The approach uses an active diagnoser that observes the system similar to a passive diagnoser and can determine when a fault has occurred with certainty. Additionally, it determines for each potential fault, which path will prove that a fault has occurred or not. Whether a different path should be taken and which path is taken by the active diagnoser is determined by which criteria are most important to the system engineers. Examples of criteria are how critical the potential fault is,

the similarity between the original mission plan and the proposed path, and the cost of specific actions.

2.6 Conclusion

This chapter gave an overview of the existing anomaly diagnosis methods, supplemented with related anomaly detection methods. The chapter answered Research question Q1: 'What are existing diagnostic methods for the diagnosis of anomalies in industrial systems?'. There is a wide variety of methods to create a diagnostics model, often having data-driven aspects. The primary domain knowledge-driven approaches create Bayesian networks, DES diagnostics models or decision trees.

3 Preliminaries

This chapter describes the information needed to read the remaining chapters. Section 3.1 gives an introduction to discrete-event systems and the control of this type of systems. Section 3.2 gives an overview of Cordis Suite and Section 3.3 introduces discrete-event system diagnosis.

3.1 Discrete-event systems and controllers

A Discrete-Event System (DES) is characterised by its discrete-state space and event-driven behaviour [33]. As described in Section 2.1, the **system state** - abbreviated to **state** - is the relevant information of a system at one point in time. The state of discrete-event systems changes because of events. The state space of a DES is a discrete set, meaning that each state is distinctly different from every other state. In other words, each state is 'isolated' [48].

Example 3.1.1. For example, a lift between two levels can be up, down or in between positions, i.e. the state space is $\{\text{UP}, \text{DOWN}, \text{INBETWEEN}\}$.

Definition 3.1.1 (Event and state transition)**.** An **event** is something occurring instantaneously, that changes the state of the system [49]. Such a state change is called a **state transition** caused by the event.

Sometimes a specific action is bound to an event, such as the press of a button. Other events are generated by the system because a variable has reached a threshold, think of the change of temperature in a system or because the lift in Example 3.1.1 reaches a position. After presenting a definition for both events and states, we present the following definition of discrete-event systems:

Definition 3.1.2 (Discrete-event system)**.** A system is a discrete-event system when it has two properties [49]:

1. The state space is a discrete set
2. All state transitions are caused by events

Comparison to other systems Compared to DESs, other types of systems differ in the kind of states and the way in which system states change. As an example, continuous-state systems describe their states by vectors of real numbers which change gradually. Informally, this means that for every state, there is another state that is infinitely close. The states in a continuous-time or discrete-time system change as the time changes, while discrete-event systems are not time-driven. For additional information on DESs in general, Cassandras and Lafortune [49] give an overview of DESs and the difference with other system types.

Structure of a DES As Sampath *et al.* [33] describe, a DES typically consist of two layers of controllers. A high-level controller commands low-level equipment controllers. Figure 3.1 illustrates this architecture. The low-level controllers use sensors and actuators of the physical system to interact with this system. The physical system is also called the **plant**. Each low-level controller is an interface for a physical component in the system, such as a lift. The high-level controller, also called a **supervisor** or **supervisory controller**, is tasked with the control and coordination of the low-level components.

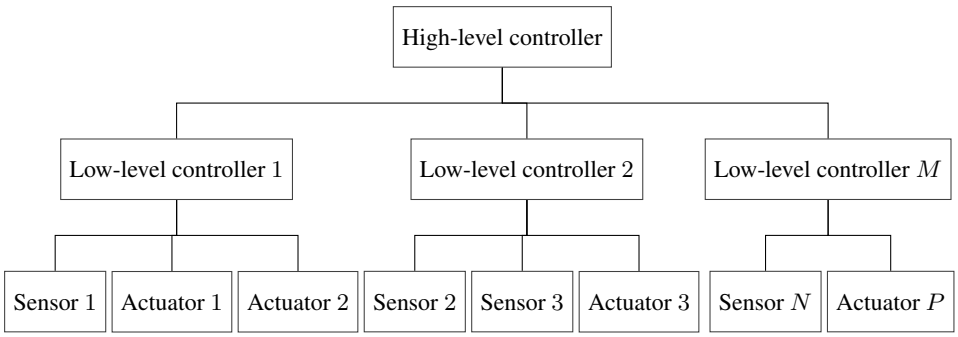


Figure 3.1: Typical structure of a discrete-event system

3.1.1 Controller

In this thesis, we assume that the supervisory controller is the 'brain' of the system. The low-level controllers are a way for the supervisory controller to communicate with the rest of the system using sensors and actuators. Therefore, in this report, the supervisory controller is the part of interest and will be abbreviated to controller, while the low-level controllers will be implicitly available.

A **Machine Control Application** is another word for a supervisory controller; it is the software application run on hardware. In practice, machine control applications run on Programmable Logic Controllers (PLCs) and micro-controllers. So-called Soft PLCs implement traditional PLC functionality through software on personal computers, often running a specific operating system for embedded applications [50]. We should note that PLC-controlled systems are only a type of DES, the methods we propose are applicable to all DESs.

Scan cycles One notable characteristic of PLCs is the way they handle the sensor and actuators. PLCs operate in *scan cycles*. Every scan cycle, a PLC first reads its inputs, then executes its instructions as described in its machine control application and finally updates its outputs [51]. Figure 3.2 shows the scan cycle of a PLC. Typically the time between scan cycles is fixed. The cycle time depends on the specific PLC and PLC manufacturer.

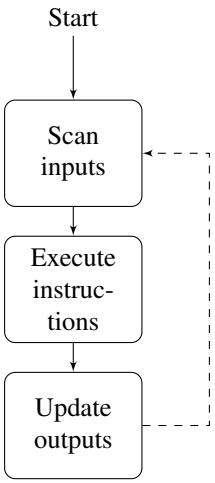


Figure 3.2: Scan cycle of a PLC

3.2 Cordis Suite

Cordis Suite is a toolset created by Cordis Automation as a means to develop and support machine control applications for industrial systems. Instead of writing code for these systems, Cordis Suite provides a low-code development platform. As an effect, system designers, the end-users of Cordis Suite, need no extensive programming knowledge. They only have to be expert on their domain and understand UML diagrams to specify machine control applications. The platform lets system developers describe their machine control application in terms of class diagrams, state machine diagrams and activity diagrams. The following summary of the functionality is based on Cordis Suite documentation [52].

Cordis Suite consists of four components, as shown in Figure 3.3. Cordis Modeler lets users describe the controller in terms of UML diagrams and provides ways to check the models' syntax. Cordis Modeler can translate the combination of diagrams into code for different kinds of programmable logic controllers. Cordis Modeler can also generate diagrams representing the current state in state machine models in the Cordis Dashboard. Figure 3.3 shows how the generated code compiles into a controller for the various types of PLCs.

Another component is the Machine Control Server (MCS) which retrieves the events from different types of PLCs and sends them to the Cordis Dashboard. The Cordis Dashboard shows the system's observable variables and the current states in the various state machine diagrams. It also sends commands to the system. The Machine Control Server also contains the functionality to create plugins. These plugins subscribe to events of controllers and send commands to controllers connected to the MCS. Examples of plugins are specific human-machine interfaces for a customer or digital twin data retrieval and collection. An additional component for simulating a machine is the Cordis Gateway. As Figure 3.3 shows, the Cordis Gateway acts as a proxy between the controller and external simulations.

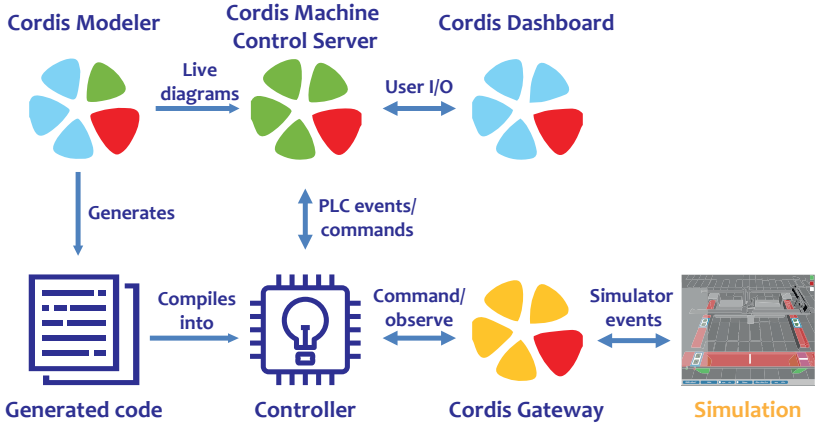


Figure 3.3: Component overview of Cordis Suite [52]

3.2.1 Cordis Modeler models

There are three types of UML diagrams used in Cordis Modeler to define a machine control application: class diagrams, state machine diagrams and activity diagrams. We will focus on class diagrams and state machine diagrams since the state machine diagrams are more commonly used in Cordis Modeler and can describe activity diagrams' functionality. Both behavioural models are equally expressive in the Cordis modeler context. This subsection introduces the models used in Cordis Suite, which are a restricted version of the corresponding UML diagram.

Class diagram A class diagram in Cordis Modeler describes the hierarchy of components in the system. The 'classes' represent the physical components in the field and more abstract components to bundle physical components that perform a task together. The class diagrams in Cordis Modeler are restricted to be directed acyclic graphs, with the entire machine's class as the root node. There are various types of relation edges possible between the different components, such as composition, aggregation and usage. The models require a strict hierarchy and no relation can form loops.

The diagram expresses the cardinality of each component on the edges. The cardinality describes the number of components in a relation. It is possible to define a set of error statuses, commands, constants, settings, input and output signals, and variables for each class. We will discuss each of these properties in the state machine diagrams paragraph below. Figure 3.4 shows a simplified example of a class diagram as described in Cordis Modeler. A heating machine consists of one blower and one heating element. The heating machine has a target temperature as setting and commands for enabling and disabling the machine. The composition edges between the components show that a heating machine cannot function without either the blower or the heating element.

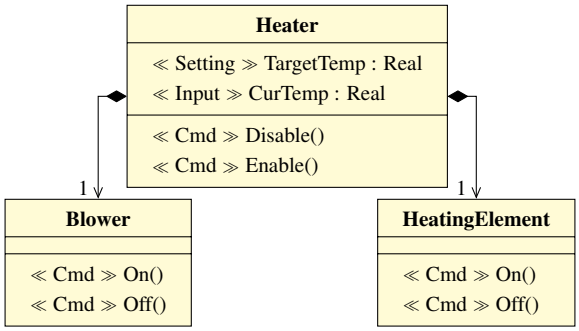


Figure 3.4: Example of a class diagram

State machine diagrams The state machines used in Cordis Modeler describe (a part of) the behaviour of a component. For each component in the class diagram, a state machine diagram can define behaviour. The state machine diagrams of a class can only access classes that are lower in the class diagram tree. The state machine diagrams consist of a set of states and the transitions between these states. Cordis Modeler offers features to improve usability such as sub-states, but we do not consider this in this chapter.

Most interesting are the state transitions. The transitions in Cordis Modeler contain guards, which are Boolean expression. When a guard evaluates to true, the state transition is enabled. Guards becoming true are considered events in Cordis state machine models. Guards consist of functions of properties. Properties describe a part of the state of components. Typically, functions on properties retrieve the value of such a property. Guards compare the properties with other properties or fixed values. Table 3.1 describes the properties available in Cordis Modeler.

Please note that the properties described in Table 3.1, except variables and settings, can also be used by components higher up in the class diagram, e.g., a state machine of the heater of the example in Figure 3.4 can also use properties of the blower on its transitions.

During operation of the controller, in each scan cycle, the current state in each state machine diagram is 'activated'. The activated diagram executes actions described in the current state and checks for enabled transitions. There are two types of actions for each state, entry statements and continuous statement. Entry statements are executed once, when the state becomes active. The continuous statements get executed each scan cycle.

Property	Explanation
Settings	A setting is a variable, specifically meant to be adjusted by system operators. Typically these are variables such as the target temperature or speed of a certain actuator.
Commands	Commands can be issued by components higher up in the class diagram, but can also be issued by a system operator.
Constants	Constants are variables determined beforehand and cannot change during system execution.
Variables	Variables are internal values that typically do bookkeeping within the model.
Inputs and outputs	Inputs and outputs are used to communicate with the hardware. Sensors are inputs, and actuators are outputs.
Input and output signals	Input and output signals are used to communicate between components inside the controller. Signals do not communicate with the hardware, but only between state machines of various components.
Observers	Inputs, outputs and input-/output-signals are internal properties by default. Only when a property is an observer, external clients will have access to these properties. Cordis Dashboard is an example of an external client.
States	The current state, sub-state or state of components lower in the class diagram.

Table 3.1: Properties in Cordis Modeler

In the case of an enabled transition, the current state in the state machine diagram gets changed according to the transition. The execution of each scan cycle starts with the state machine diagrams of the root component and descends to its children's state machine diagrams. In practice, this means that it is not a *pure* discrete-event system, since it contains discrete-time behaviour. Still, while only checked every cycle, events determine the system behaviour. Besides, any computer system operates in cycles and hence a proper discrete-event system does not exist.

Example 3.2.1. Figure 3.5 shows a state machine diagram for the HEATER in Figure 3.4. The diagram has three states, with the initial state being DISABLED. Giving the command ENABLE results in a state transition to the COOLING state. If the temperature is below a target temperature, the state machine will switch to HEATING. If the temperature is equal or above the target, or the command to disable the heater is given, the state machine will change to COOLING. COOLING has a transition to DISABLED, fired when the DISABLE command is given.

The guards contain the functions CMDCHCK, which checks if the command given as argument was activated and CMDSET which initiates a command. The functions INP and SETT retrieve the value of an input signal and a setting respectively. Cordis Suite offers various other functions for all types of properties.

3.2.2 MerryGoRound

The MerryGoRound is an example system of Cordis Automation used to teach users about Cordis Suite and to test Cordis Suite functionality. The MerryGoRound is a fictitious system designed to indefinitely move trays between components that together form a loop. The trays

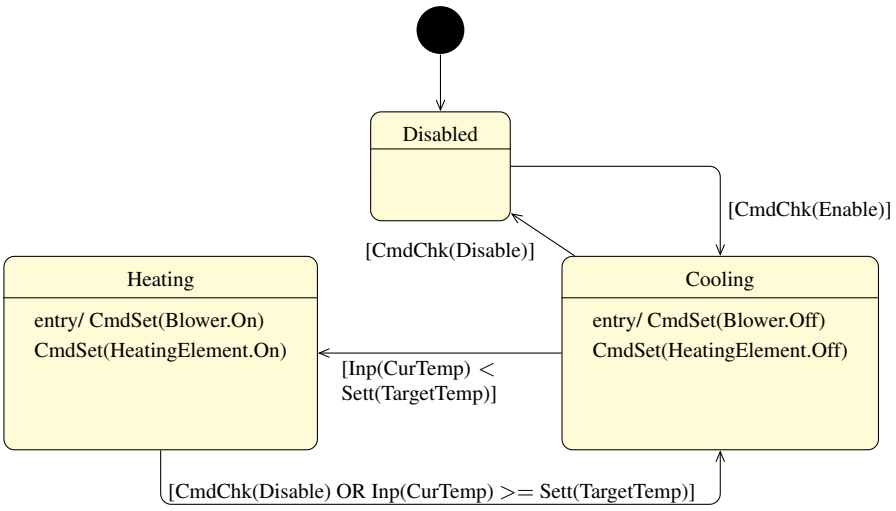


Figure 3.5: Example of a state machine diagram

move between multiple components to eventually return to their starting point, where the same route repeats. Each component has sensors and actuators. The machine can move trays clockwise and counter-clockwise. Figure 3.6 gives an overview of the MerryGoRound. When the system starts in the counter-clockwise configuration, a gantry moves trays from the back of the MerryGoRound (an arbitrary starting point) to the left conveyor belt. The left conveyor belt transports the trays to a lift. A lift (shown in green) moves the trays down onto the front conveyor belt. Next, the front conveyor belt moves trays to the right lift. The right lift moves the trays onto the right conveyor belt, which transports the trays to the right gantry. The right gantry puts the trays back in the starting position.

Cordis Modeler generates the controller of the MerryGoRound. The controller specification describes the behaviour of each component by a set of state machine diagrams. Figure 3.7 shows a part of the class diagram describing the components' hierarchy.

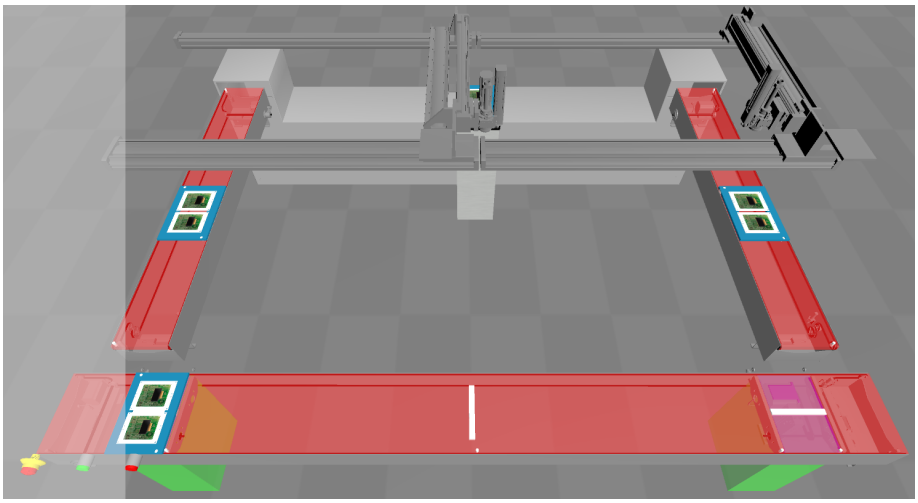


Figure 3.6: Image of the MerryGoRound simulation

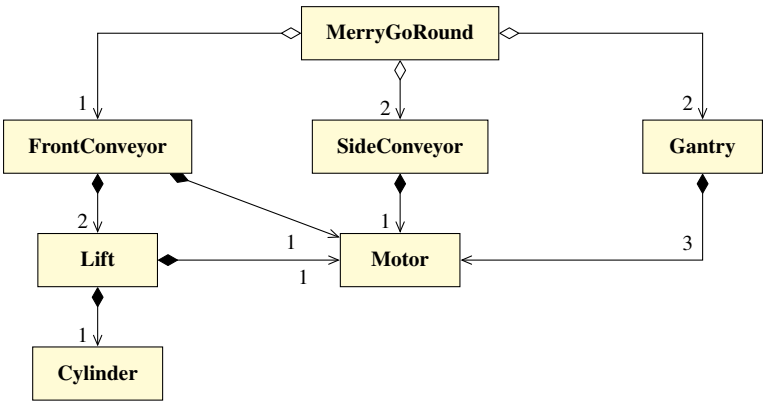


Figure 3.7: Abstract version of MerryGoRound class diagram

3.3 Discrete-event system diagnosis

Sampath *et al.* [35] introduced a formal framework for the diagnosis and diagnosability of discrete-event systems. Their work provides the basis for most of the other works discussing diagnosability and diagnosis of DESs. Throughout this thesis, models for the diagnosis of DESs will be described as **DES diagnostic models**, to clarify that the models can diagnose systems. This clarification is needed because the models are very similar to models that describe the behaviour of the discrete-event systems. While the previously described machine control application models describe the desired part of the behaviour of the system and describe what the actuators should do based on the received sensor values, the DES diagnostic models are more generic and describe all possible state of a discrete-event system.

For the introduction to the framework in this section, we use our own definitions of DES diagnostics based on the original paper of Sampath *et al.* [35] and the review work of Zaytoon and Lafortune [34]. Before introducing DES diagnostics, this section will introduce several concepts and formulas that are used to diagnose DESs, but are not specific to DES diagnostics.

3.3.1 Finite state machine

A Finite State Machine (FSM) describes the states and state transitions of a discrete-event system. An FSM is represented by a four-tuple as defined by Equation 3.1, where X are the states, Σ is the set of events, $\delta : X \times \Sigma \hookrightarrow X$ is the (partial) transition function and $x_0 \in \Sigma$ is the initial state [53].

$$G = (X, \Sigma, \delta, x_0) \tag{3.1}$$

Definition 3.3.1 (Active event set)**.** The transition function determines the potential events possible at each state. For an FSM $G = (X, \Sigma, \delta, x_0)$, the subset of events that could occur in state $x \in X$ is called the active event set of that state and is defined by Equation 3.2.

$$e(x) = \{\sigma \mid \sigma \in \Sigma \wedge \delta(x, \sigma) \text{ is defined}\} \tag{3.2}$$

Each FSM describes a language, i.e, a set of sequences of events. We will denote the language of an FSM G with $L(G)$. We will extend the transition function for an FSM to define it for a sequence of events, if that sequence of events is part of the language generated by the FSM. We will call this the extended transition function of an FSM and use it do show the resulting state of an FSM after a sequence of events.

Definition 3.3.2 (Extended transition function)**.** Equation 3.3 defines the extended transition function as a recursive definition for an FSM $G = (X, \Sigma, \delta, x_0)$ where $x \in X$ is a state, $s \in \Sigma^*$ is a sequence of events, $\sigma \in \Sigma$ is a single event and $s\sigma \in L((X, \Sigma, \delta, x))$, i.e, in the language of G, with x as initial state. Equation 3.4 shows the base case of the recursive definition, an empty sequence of events ϵ results in staying the current state x.

$$\delta^*(x, s\sigma) = \delta(\delta^*(x, s), \sigma) \tag{3.3}$$

$$\delta^*(x, \epsilon) = x \tag{3.4}$$

3.3.2 Partial observability

Controllers of discrete-event systems can only observe events through sensors. From the perspective of the controller, the commands given to a system and the received sensor values from a system are the only observable events of a system. Other system events cannot be seen by the controller and are deemed to be unobservable. The function that determines what is visible to the controller is called the projection $P(.)$. Figure 3.8 shows the idea of partial observability of the controller; it can only see the projection $P(.)$ of the system behaviour. In this example, E_1, E_3 and E_t are visible to the controller, while E_2 is not.

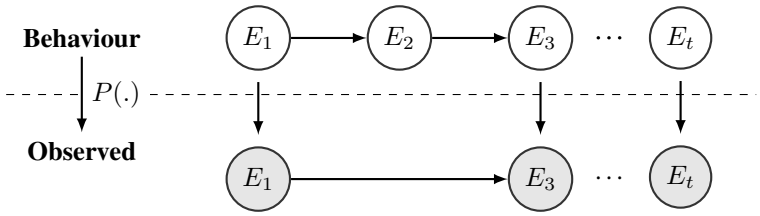


Figure 3.8: The controller can only see observable events

Definition 3.3.3 (Observable and unobservable events)**.** For an FSM $G = (X, \Sigma, \delta, x_0)$, we can define what is observable from the perspective of the controller. The event set is split into observable and unobservable events $\Sigma = \Sigma_o \cup \Sigma_{uo}$, where Σ_o are observable events and Σ_{uo} are the unobservable events.

For instance, observable events are sensor values received by the controller and commands given by the controller.

Because of partial observability, it is also impossible to know the exact system state from the perspective of the controller. When the controller knows the current state and observes an observable event, several unobservable events could precede the observable event.

Definition 3.3.4 (Reach)**.** For an FSM $G = (X, \Sigma, \delta, x_0)$, the reach $S(x, \sigma)$ of state $x \in X$ and observable event $\sigma \in \Sigma_o$ gives the set of possible states of the system after the occurrence of the observable event σ and the sequence of preceding unobservable events s. The reach $S(x, \sigma)$ is defined in Equation 3.5.

$$S(x, \sigma) = \{\delta^*(x, s\sigma) \text{ where } s \in \Sigma_{uo}^*\} \tag{3.5}$$

In practice, some of the events in a system are undesired. A *fault event* causes an undesired deviation of the system. After such a fault, the system does not have the desired *normal behaviour*, but has *faulty behaviour*. Likewise, if the current state is reached without faults, it is called a *normal state*, while a state reached after the occurrence of a fault is called a *faulty state*. An additional distinction on the event set can be made.

Definition 3.3.5 (Faults)**.** For an FSM $G = (X, \Sigma, \delta, x_0)$, the state space X and the event set Σ account for the normal and faulty behaviour of the system. Fault events in the system are assumed to be unobservable, because otherwise they are trivial to detect. Hence, let the fault set, the set of fault events of a system, be $\Sigma_f \subseteq \Sigma_{uo}$.

3.3.3 Synchronous composition

A DES consisting of several components can be described in a modular way, by defining an FSM for each component. The complete system model can be generated from component models by synchronous composition [54]. For two FSM's, $G_1 = (X_1, \Sigma_1, \delta_1, x_{01})$ and $G_2 = (X_2, \Sigma_2, \delta_2, x_{02})$, the synchronous composition $G = G_1 \times G_2$, resulting in $G = (X, \Sigma, \delta, x_0)$ is defined by Equations 3.6 to 3.9. We will denote the synchronous composition of multiple models with $\prod$.

Equation 3.9 describes the transition function δ. A distinction is made based on the active event set of the original states of the individual models. If event σ is possible in both states x_1 and x_2, then the composed model contains a transition with the event. The transition describes that both of the original models execute the transition with the shared event. Please note that this is the only way that a shared event can be fired. If an event σ is active in x_1, but not in x_2 and the event is used elsewhere in G_2, i.e., $\sigma \in \Sigma_2$, then there is no transition in the composed model. Furthermore, if an event exists in G_1 or G_2 only, then only that model executes a transition.

$$\Sigma = \Sigma_1 \cup \Sigma_2 \tag{3.6}$$

$$X = X_1 \times X_2 \tag{3.7}$$

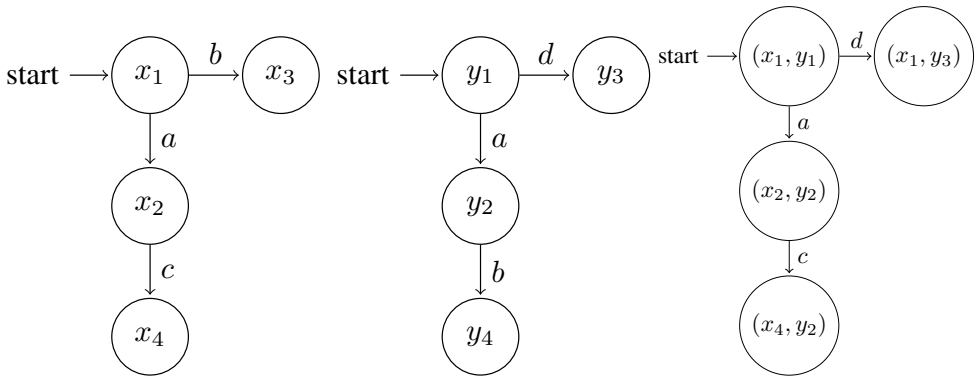
$$x_0 = (x_{01} \times x_{02}) \tag{3.8}$$

$$\delta((x_1, x_2), \sigma) = \begin{cases} (\delta_1(x_1, \sigma), \delta_2(x_2, \sigma)) & \text{if } \sigma \in e_1(x_1) \cap e_2(x_2) \\ (\delta_1(x_1, \sigma), x_2) & \text{if } \sigma \in e_1(x_1) - \Sigma_2 \\ (x_1, \delta_2(x_2, \sigma)) & \text{if } \sigma \in e_2(x_2) - \Sigma_1 \\ \text{undefined otherwise} \end{cases} \tag{3.9}$$

Example 3.3.1. To illustrate the process of synchronous composition, an example is shown in Figure 3.9. The two left models, Figures 3.9a and 3.9b, are composed into the right model, Figure 3.9c. The transition with event a is a shared transition. x_1 and y_1 and share a as an active event. In the composed model, this result in a transition to state (x_2, y_2). Event d is possible from y_1 and is not present in G_1, hence a transition is added from (x_1, y_1) to (x_1, y_3), only the state of G_2 changes. Event b cannot be executed in (x_1, y_1) because both models share the event and b is not in the active set of y_1. This gives an example for each case in Equation 3.9.

3.3.4 Overview of DES diagnostics

The goal of the DES diagnostics approach is to create a diagnoser; a model that can detect failures and can diagnose which fault caused the failure based on the observable behaviour. The diagnoser is generated from a system model. The diagnoser runs along the system and uses the observable events. The diagnoser deals with the uncertainty of unobservable event by keeping track of all possible states of the system (including normal and faulty states) after each observable event. After an observable event, the diagnoser determines whether unobservable events could have happened just before the observable event (by checking a diagnosis model). Based on the observable event and the potential unobservable events, it determines the next potential states of the system (again, based on the diagnosis model). A DES diagnoser is created by the following three steps:



(a) Example G_1 (b) Example G_2 (c) Composition of G_1 and G_2

Figure 3.9: Illustration of synchronous composition

1. For each component i, create a diagnosis model $G_i = (X_i, \Sigma_i, \delta_i, x_{0i})$. These are models that contain faults.

2. Generate the complete system diagnosis model by creating the synchronous composition of the component diagnosis models. I.e., for component models $G_i, i = 1 \ldots N$, create G = (X, Σ, δ, x_0) with $G = \prod_{i=1}^{N} G_i$.

3. Generate a diagnoser from the system model created in Step 2, which we will explain in Section 3.3.5.

3.3.5 Creating the diagnoser from the system model

The final step is to create a diagnoser from the system diagnosis model. A diagnoser for system diagnosis model $G = (X, \Sigma, \delta, x_0)$ is an FSM denoted by $G_d = (Q_d, \Sigma_d, \delta_d, q_{0d})$. A state $q_d \in Q_d$ in the diagnoser has the form $q_d = \{(x_1, \ell_1), \ldots, (x_n, \ell_n)\}$ where $n \in \mathbb{N}$, $n \geq 1$ and $x_i \in X$ and the label ℓ_i is a set giving information about faults.

Definition 3.3.6 (Fault label)**.** A label ℓ_i can be $\emptyset$, meaning that the system state is the result of normal system behaviour or a set of faults $\{F_1, F_2, \ldots, F_k\}$ where $k \in \mathbb{N}$, $k \geq 1$ and $F_i \subseteq \Sigma_f$ for $1 \leq i \leq k$, if the described faults resulted in the system state.

Each state in the diagnoser represents the potential system states, based on the observable behaviour. The diagnoser assumes that the system starts in a normal state, with a corresponding $\emptyset$ label. Hence, the initial state of a diagnoser is $\{(x_0, \emptyset)\}$.

Definition 3.3.7 (Diagnoser calculation)**.** A diagnoser $G_d = (Q_d, \Sigma_d, \delta_d, q_{0d})$ is recursively calculated from the initial state. Given a state q_1 of the diagnoser and an observable event σ, the next state q_2 can be calculated with the following two steps:

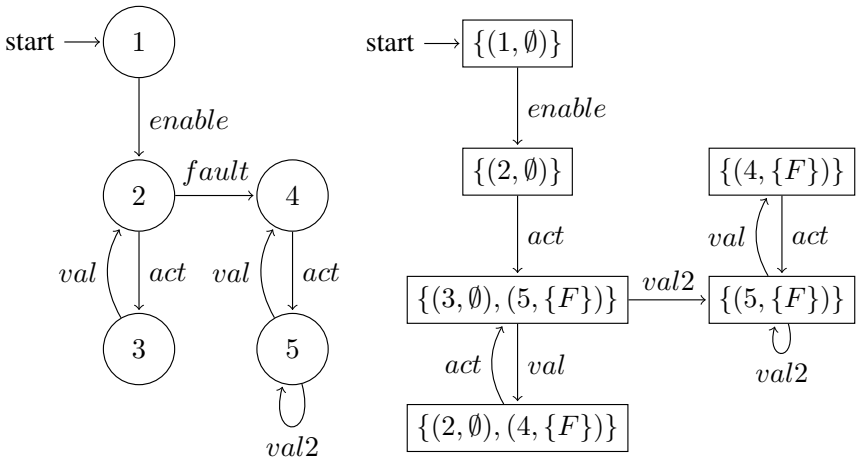
1. For each system state x in the diagnoser state q_1, calculate the reach $S(x, \sigma)$, based on the observable event σ and the possible unobservable events preceding the observable event.

2. For a new state $x' \in S(x, \sigma)$, where s is the sequence of unobservable events leading to x', that is $\delta^*(x, s\sigma) = x'$, the label propagates based on the current fault label and the potential faults found on transitions. Equation 3.10 defines how a label ℓ of (x, ℓ) in q_1 propagates into the new label ℓ' of (x', ℓ') in q_2.

$$\ell' = \ell \cup \{f \mid f \text{ occurs in } s\} \tag{3.10}$$

Example 3.3.2. Figure 3.10a shows an example system model. The only unobservable event in the model is the fault event $fault$, which we will describe as F in the diagnoser. Figure 3.10b is the computed diagnoser for the system model example. The system model and the generated model are based on an example of Zaytoon and Lafortune [34], but are altered to adhere to the original definition of Sampath *et al.* [33].

As described in the creation procedure of the diagnoser, the diagnoser starts in the initial state with the label $\emptyset$. The only possible observable event in the system model is $enable$. Since we assume that an unobservable event can only precede an observable event and there is no event preceding $enable$, the next state of the diagnoser is $\{(2, \emptyset)\}$. From state 2, the command act can be given. In this case, the observable event can be preceded by the unobservable event $fault$. Therefore the reach is $\{3, 5\}$, resulting in state $\{(3, \emptyset)\}, (5, \{F\})\}$, meaning that the system could be either in state 3 or 5. The label of 5 is F, since a fault must have occurred to be in that state. The remainder of the model is calculated in the same way.



(a) Example system model

(b) Diagnoser for fault $fault$

Figure 3.10: Example of a system model and its diagnoser

3.3.6 Diagnosability

If the diagnoser runs in parallel with the system, the diagnoser can determine if faults occurred by checking its state's labels. If the diagnoser reaches a state with only tuples carrying a label with faults, then a fault occurred with certainty. A state of the diagnoser without empty labels is called an *F-certain state*. More specifically, if each label contains the same fault F_i, then the diagnoser state is *F_i-certain*. In the example of Figure 3.10, the states $\{(4, \{F\})\}$ and $\{(5, \{F\})\}$ are F-certain. When $\emptyset$ is one of the labels in the diagnoser state, it is impossible to determine that a fault has occurred. A distinction can be made between *normal states* with only empty labels and *F-uncertain states*, which are states of the diagnoser that contain both fault and empty labels. The diagnoser states $\{(1, \emptyset)\}$ an $\{(2, \emptyset)\}$ are examples of normal states, while $\{(3, \emptyset), (5, \{F\})\}$ and $\{(2, \emptyset), (4, \{F\})\}$ are examples of F-uncertain states.

A system is diagnosable if a fault in the system leads to an F-certain state within a finite number of transitions. If there are loops of F-uncertain states in the diagnoser, the system may remain in this loop forever. These loops are called *F-indeterminate cycles*. When the system gets in such a cycle after a fault, the diagnoser might not detect the fault in a finite number of transitions. A system's diagnosability can be determined by checking the absence of F-indeterminate cycles in the diagnoser model. Sampath *et al.* [33] propose regular loop-cycle finding algorithms to check for the existence of F-indeterminate cycles.

With this definition of diagnosability, the system of Figure 3.10a is not diagnosable, since it contains an F-indeterminate cycle between states $\{(3, \emptyset), (5, \{F\})\}$ and $\{(2, \emptyset), (4, \{F\})\}$. The non-diagnosability of this system is also rather intuitive. As long as the system does not produce the event $val2$, it is impossible to know if the fault occurred.

4 Comparison of applicable methods

Chapter 2 gives an overview of existing methods for diagnosing anomalies. The found methods revolve around three models: Bayesian networks, decision trees and DES diagnostic models. To test if these models are a good fit for systems controlled by machine control applications and satisfy the basic requirements, we created a proof of concept for each model. In each proof of concept, we created a model using an existing tool and worked out an example scenario. We tried to tweak the model using the available literature on each model.

We will compare the models that satisfy the basic requirements. The basic requirements for a method, as described in Section 1.3, are being semi-automatic, knowledge-driven, and be able to use machine control application specifications in combination with system run-time information. We created a proof of concept for Bayesian networks and DES diagnostic models, which satisfy the basic requirements. The found methods using decision trees violate the basic requirements. The most promising method using decision trees found in the literature, the rule-based decision tree generation of Abdelhalim *et al.* [29], in practice still requires a data-driven first step to determine the rules. Throughout the proof of concept we noticed that it did not satisfy the basic requirements. A system expert could also determine the rules, but then the method is out-of-scope due to not being semi-automatic, since neither the generation of the model nor the diagnosis using the model is semi-automatic. Appendix A.1 describes the proof of concept using decision trees and shows that it does not satisfy the basic requirements.

This chapter compares the two methods found that do fulfil the basic requirements. The remainder of this chapter presents a list of evaluation criteria in Section 4.1, presents the diagnosis application of both Bayesian networks and DES diagnostic models in Sections 4.2 and 4.3, compares the two methods based on the list of criteria in Section 4.4 and presents concluding remarks in Section 4.5. The used example system is the MerryGoRound, as described in Chapter 3.

This chapter answers Research question Q2: 'Which quality criteria determine how well diagnostic methods fit in the context of machine control applications?' and Research question Q3: 'Which method for diagnosing detected anomalies of industrial systems found by answering Research question Q1 is most fitting according to the criteria of Research question Q2?'

4.1 Quality criteria

This section describes a set of criteria that help to compare the diagnostics approaches and select the most suitable method for diagnosing anomalies in the context of systems controlled by machine control applications. The criteria in this section are based on the process failure diagnosis criteria used in Venkatasubramanian *et al.* [55]. Some of the process failure diagnosis criteria focus on the detection of anomalies. These are out of scope, as our goal is to diagnose already detected anomalies. The remaining criteria, that do focus on the diagnosis of failures, are tailored to MCA-controlled systems.

Effort of modelling Typically the generation and usage of a model are two steps in a diagnostics process. The steps should be usable for system experts with as little effort as possible. This criterion focuses on the first step, the generation of the model. It should be as automated as possible based on the domain knowledge present in the system specification and machine control application specification.

Composability System developers of MCA-controlled systems reuse components in various other components. For example, an electric motor is a component in various larger components such as lifts and conveyor belts. As the faults in these sub-components are comparable and propagate to the parent component, the diagnosis models corresponding to the sub-components should also be reusable and composable. Methods should create a system model as the composition of various component models.

Expressiveness The expressiveness of the model used in approaches is vital for the applicability of the approach. With expressiveness we mean the practical expressiveness, i.e, the modeling constructs of a model should be at a good abstraction level. It means that various systems can be modelled and diagnosed. Machine control applications are used in a vast number of domains, varying from bridges to manufacturing systems. Models should be generic and should be able to express the various systems.

Model adaptability As systems change over time, so should the diagnosis model of the system. Ideally, an approach should handle changes in the system without much manual effort.

Model understandability If methods require manual work for the generation of a model, this work should be understandable for a designer and require minimal effort [55]. As a designer should have a good understanding to make a correct model, the model should be understandable for the typical designer who normally creates machine control applications.

Diagnosis explainability Part of the usability is the diagnosis explainability, also called explanation facility [55]. The minimum requirement of the diagnosis of a method is to find the root cause of a detected anomaly or failure, i.e, the component that is broken. In that sense, the approach explains what the underlying problem of the anomaly is. The exactness of the diagnosis is also relevant, i.e, the more specific the fault is, the better. For example, pointing to a small components gives more information than pointing to an overarching component as broken component. Preferably, the model also shows the causal path from cause(s) to effect(s). This information on why the fault resulted in a failure, helps to give insight into the problem. This insight gives confidence that the approach made the right diagnosis.

Scalability The impact of the system size on the diagnosis model size and required computational power is an essential factor. The goal is to find the root cause of the problem quickly to prevent costly downtime. Real-time approaches usually focus on less expensive algorithms in term of computational complexity, but often come with high memory consumption [55]. Both the computational complexity and memory consumption should be reasonable.

Effectiveness The second step of diagnosis uses the created model to diagnose anomalies in the system. The approach's effectiveness is its power to diagnose failures based on the observations given, i.e., its capability to identify the correct fault.

Tool support A less critical criterion for an initial decision, but an essential criterion for implementing an approach are the available tools and their maturity. Approaches with mature tools can be applied more easily, potentially after some modification. We will assess the maturity of tools by experimenting with the tools and checking their usability and robustness. While it is possible to create a proof of concept of tools as part of this project, it is out of scope to create mature tools.

4.2 Bayesian networks

The approach of Barbini *et al.* [2] seems to be the most practical and scalable approach that focuses on the diagnosis of a similar type machines and similar types of failures. Furthermore, the example used to introduce the approach is most similar to an industrial system controlled by a machine control application. Therefore, the first proof of concept uses their approach and tools to create a Bayesian network. The main difference in the domain is that the Bayesian networks created by their domain-specific language assume a physical connection between components. A component can only work because it is provided with a resource (such as energy) by another component. In MCA-controlled systems, however, the controller handles the behaviour of the components. Software logic, instead of a physical connection, determines the connection between components. The method used in the proof of concept extends the approach of Barbini *et al.* [2] with software logic components as intermediate links between physical components.

Modelling a lift component Figure 4.1 shows a lift component. As described by the approach of Barbini *et al.* [2], the model describes the causal relationship between input and output signals of components. Each node represents a random variable described by a conditional probability table (not shown in the figure). The edges in the model describe the dependencies between the random variables. The model diagnoses the lift's health, based on the current and next position, and its current state. In the example, the lift is in the LOW state, as represented by the lift's INPOSITION. In the next state, as represented by the lift's OUTPOSITION, it will be INBETWEEN. Since the lift's state is DISABLED, this is faulty behaviour.

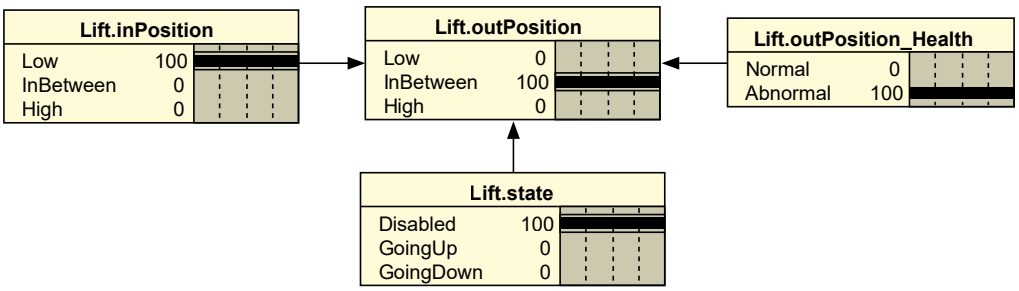


Figure 4.1: Bayesian network for a lift

Modelling multiple components The causal graph of Figure 4.2 shows the influence of the components' behaviour on other components in a part of the MerryGoRound. The model describes the causal relation between the left lift, the right lift, and the front conveyor belt. The model shows that the OUTPOSITION of the lifts and conveyor is the INPOSITION in the next time slice. The causal graph shows the lift of Figure 4.1 twice, one for both lifts in the MerryGoRound, but with an additional edge from OUTPOS to INPOS to show that the current position will be the position of the lift in the next time slice. The logic component in the middle, describes how the sensor values of all components determine the new state of the components. As can be seen in the diagram, this means that there are loops in the Bayesian network. The loops invalidate the shown Bayesian network as a static Bayesian network. Hence, the approach can only model the system as a dynamic Bayesian network.

The approach has the advantage that the model includes the desired behaviour of the components. By knowing the fault-free behaviour of the system, the rest of the behaviour is automatically faulty. The model can diagnose a failure by inserting the actual behaviour as pinned-down probabilities to the Bayesian network. After changing the probabilities of observable nodes, the probabilities of all nodes change due to the dependencies in the network and the probabilities of

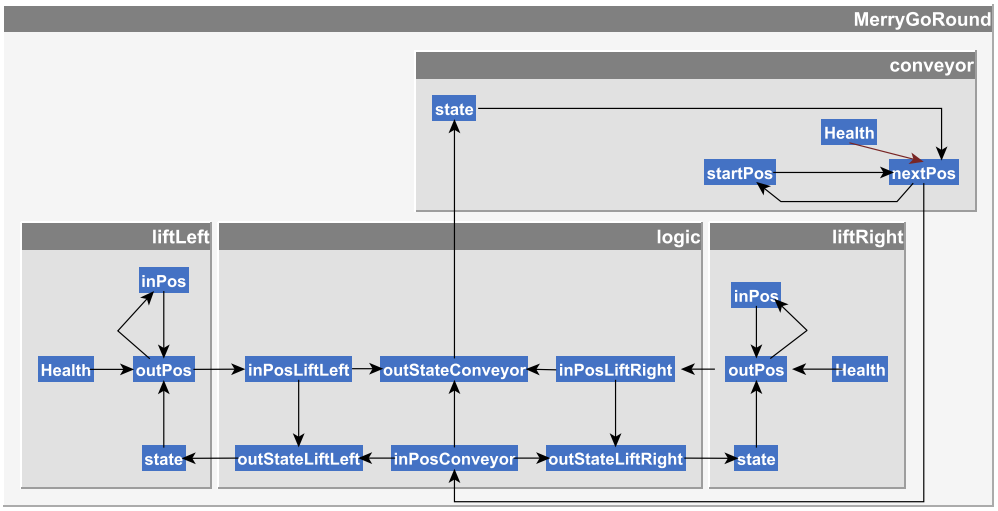


Figure 4.2: MerryGoRound physical (Dynamic) Bayesian network

the model's health nodes are updated. Specific faults of components can also be modelled. In the extended model, instead of the health node, each fault is modelled as a node that depends on the current position, next position and current state. The lift in Figure 4.1 is diagnosed with a circuit-error in the extended model, because the lift moves up while being disabled, typically caused by a short circuit.

Bayesian networks using capability modelling and physical modelling Personal communication with a TNO colleague gave insight into the usage of capability modelling in addition to the physical modelling used in the Bayesian networks approach of Barbini *et al.* [2]. Capability nodes describe observable and required functionality of the system for normal behaviour. Using capability nodes in a Bayesian network prevents a dynamic Bayesian network's complexity by giving the abstraction to prevent loops because they provide the abstraction of time. Instead of describing the impact of the lift's current position on the next position, one can express that the lift can move up and down.

Figure 4.3 shows the result of applying the insight onto the original model. The network consists of three layers of nodes. The upper layer describes the capability of the entire system. In this case, this is the capability to move a tray from start to finish (e.g., moving the tray a single round). The tray has to move along a sequence of segments to fulfil the system capability. The middle layer describes these segment capabilities and their required capabilities. The bottom layer captures the required components and their causal relation. The capability nodes are deterministic; all required components or required capabilities of the system have to function for the node to be 100 per cent capable. The deterministic nodes are shown in a darker colour.

Generating such a model requires several forms of information. This method requires the set of components that could be defective, the system's capabilities that should also be observable and the relations between the capabilities and required physical components. Finally, the probabilities of faults in a component must be determined. The comparison at the end of this chapter uses this final approach with capability modelling in addition to physical modelling.

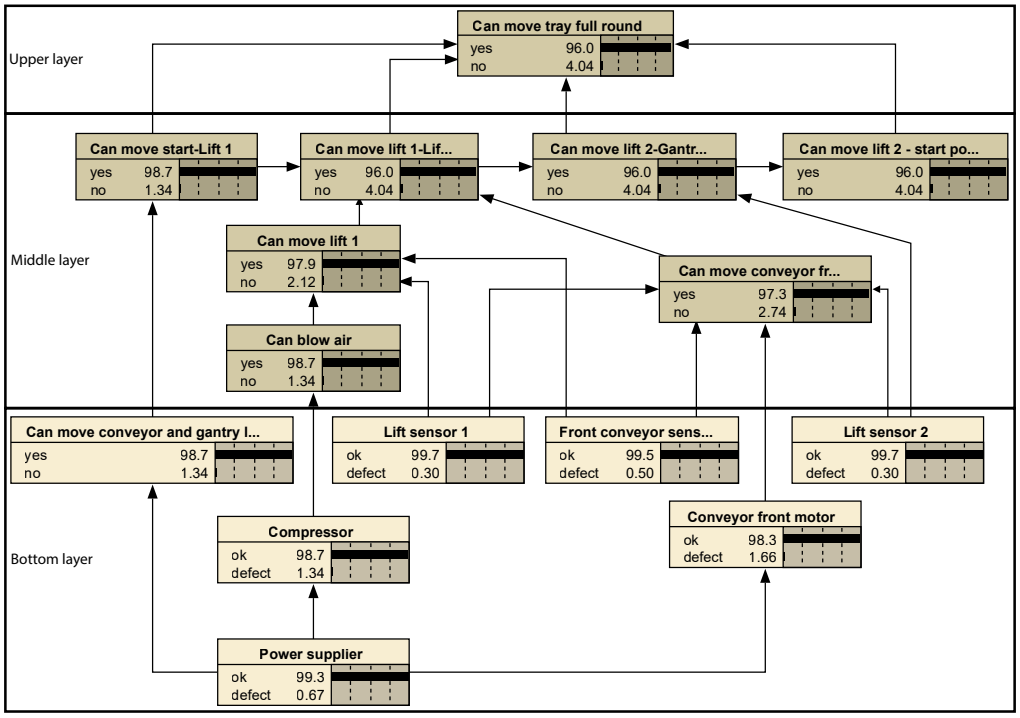


Figure 4.3: MerryGoRound Physical/Capability Bayesian network

4.3 Discrete-event system models

In the proof of concept using DES diagnostic models, several iterations were made before a model was created that was both understandable and could diagnose the system. The first attempts used the classical approach of Sampath *et al.* [33] because it provides a baseline of the capabilities of DES diagnostic models and the approach provides a detailed description of how such a model should be created. The first attempts follow the same steps as the original paper, as described in Section 3.3.4:

1. Create models that determine the behaviour of the system (including faults),
2. Combine the models using synchronous composition, i.e., synchronise the models on their shared transitions,
3. Create a diagnoser from the model created in Step 2.

Using a modular approach The created models with the classical approach were too large and too computationally complex for the given example. Therefore, a modular approach, similar to that of Debouk *et al.* [44] is applied. Instead of a diagnoser for the entire system, a diagnoser is created for each component. A modular approach makes the models readable for a system engineer. It is a small part of the total system model and prevents the complexity of monolithic models. The size of a diagnoser is exponential in the number of states of the system model. Therefore, the computational complexity of and memory required for diagnosers is reduced by using multiple smaller models. Since the physical components often have a loose coupling, it is assumed that this does not significantly impact the diagnosability of the entire system. The approach used the steps described above but repeats the steps for each component.

The next encountered problem is that the proof of concept cannot uniquely determine the fault that caused a detected failure, due to the autonomous controller used for the MerryGo-Round. The controller commands the system to move trays sequentially between various segments. The sequential behaviour means that the system comes to a halt when one of the components fails. Since the system comes to a halt, the diagnosers do not get enough observations to determine the root cause. As the goal is to diagnose systems controlled by a machine control application without changing the original behaviour, changing the behaviour beforehand to make the system diagnosable is out of scope.

Using active diagnosis Instead, the proposed proof of concept changes the behaviour when a failure has occurred. Changing the behaviour to diagnose faults, called active diagnosis, reduces the number of potential faults by stopping the controller and actively executing commands and monitoring the system's output. Based on the behaviour of the system, the active diagnoser can exclude faults. We assume that when a failure occurs, it is possible to stop the controller and command the system to do actions that can determine the root cause. Another assumption made is that only one fault is the cause of a failure. The steps used are still based on Sampath *et al.* [33], but an active diagnosis part is added to the controller after the controller knows that a failure occurred. The following steps are used in the final version of the DES diagnostic model proof of concept:

1. Create models that determine the behaviour of the system (including faults),
2. Create active diagnosis models that synchronise on transitions in the models created in Step 1,
3. Combine the models using synchronous composition, i.e., synchronise the models on their shared transitions,
4. Create a diagnoser (in this case, the active diagnoser) from the model created in Step 3.

Using Cordis models as input As the primary piece of information, the proposed method of active diagnosis extracts the state transition diagram describing the lift's behaviour from Cordis Suite. Figure 4.4 shows the Cordis Suite state machine diagram of the lift. The figure shows the four commands given to the lift: it can go up and go down, load, and unload. These lift commands result in action commands given to a cylinder (also called lift actuator) and a conveyor belt. The cylinder makes the lift go up and down; the lift conveyor belt moves to load and unload trays from the adjacent belts. For readability reasons, the remainder of this proof of concept description focuses on the failure of the lift not going to the 'up position' in the MerryGoRound example.

Adding faulty behaviour The proposed method adds manually defined faults to the model, similar to the faults that the Bayesian network proof of concept used. In contrast to Sampath *et al.* [33], the faults are not determined per atomic component but are based on the lift's potential failures. Hence, in the model, multiple fault transitions are always followed by a failure transition.

The model shown in Figure 4.5, keeps the structure of the Cordis Suite model but has transitions describing a time out. The model has fault transitions that lead to states where time outs start. For instance, the top part of the model shows that after a LIFTCMD(UP), it is possible that the action commands is not followed by a 'lift is in up position' sensor value shown as CYLINDEROUTPSIGNAL(OINUPPOSITION). Then, the system will generate a time out. In this case, it is unclear whether a broken sensor or a broken cylinder caused this time out; hence there are two transitions to the time out state, for both faults.

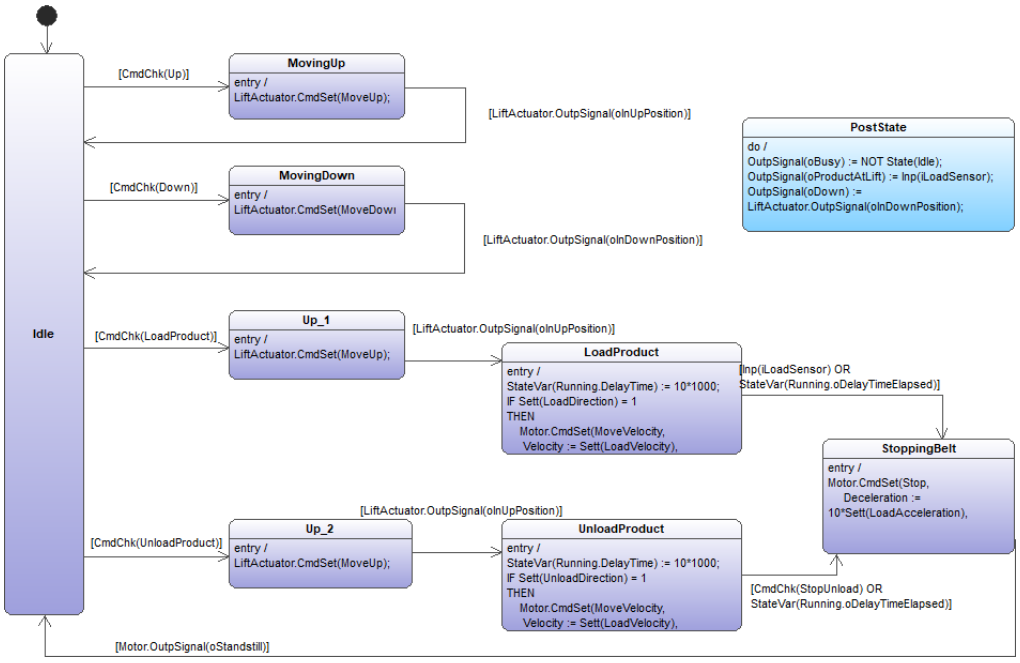


Figure 4.4: The lift behaviour in the MerryGoRound Cordis Suite model

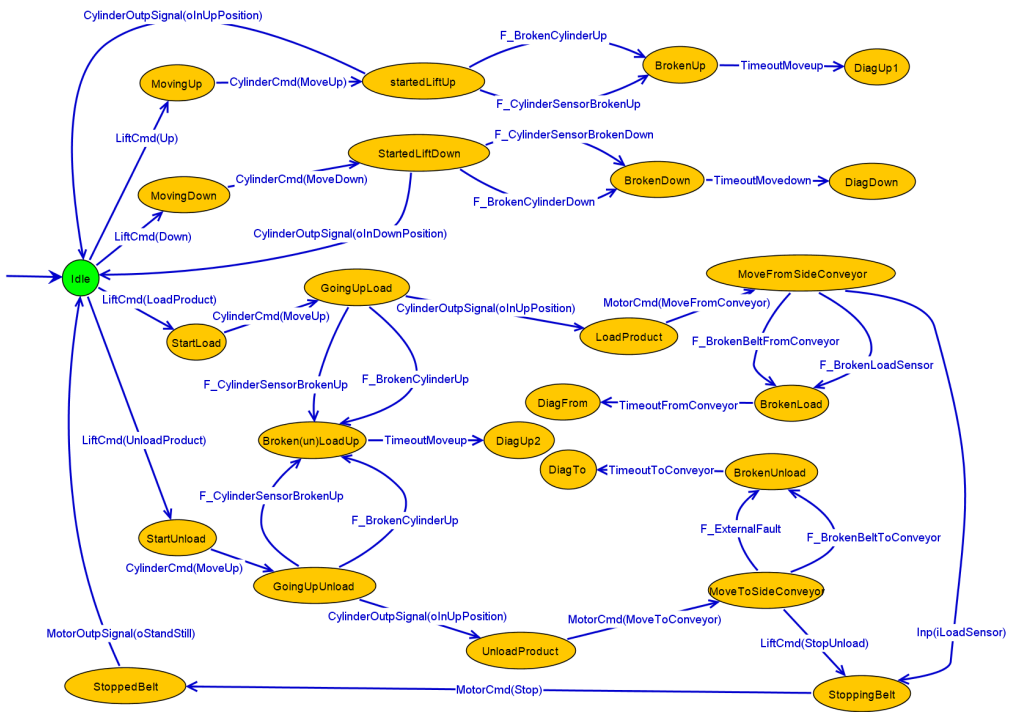


Figure 4.5: The DES diagnostic model including faults based on Figure 4.4

Determining active diagnosis parts The method adds an active diagnosis part to the model to differentiate between the two faults after the failure has occurred. The active diagnosis models are separate models. The method automatically adds the active diagnosis models to the original model for all places where the lift goes up and should report that it is in the up position with a sensor value change. We use synchronous composition to combine the active diagnosis part with the model, but other methods could also be used.

Figure 4.6 shows the active diagnosis part that is modelled manually. The active diagnosis part shows the effect of the unobservable fault F_CylinderSensorBrokenUp and F_BrokenCylinderUp on the system behaviour. Typical for DES diagnostic model approaches, by checking which observable path is followed by the system, the unobservable fault that causes the behaviour is determined.

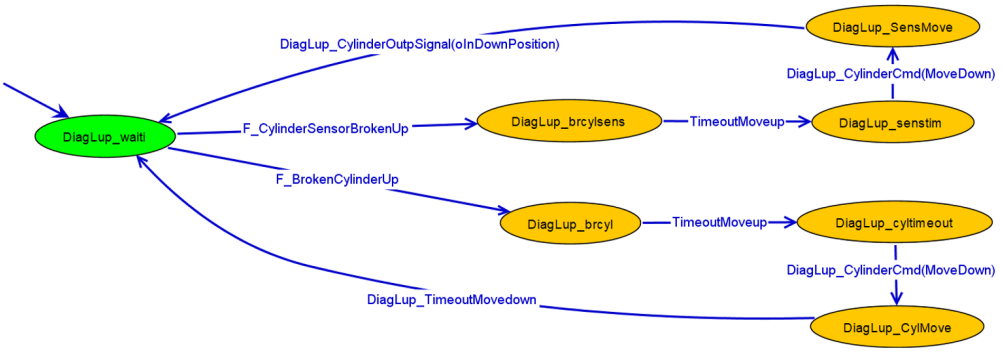


Figure 4.6: Active diagnosis part to differentiate sensor and actuator fault

The active diagnosis part shows two potential traces of the system, based on the possible faults. The active diagnosis part commands the system to move the cylinder down. If the bottom sensor detects the lift, then the cylinder is not broken. The model diagnoses the cause of a time out to be a broken upper lift sensor. If a time out happens after the command to go down, then the cylinder is broken.

Adding active diagnosis to the system model Next, the method creates the synchronous composition of the active diagnosis part and the lift model that includes faults. Figure 4.7 shows part of the resulting model. The two paths of the active diagnoser are visible in the composition.

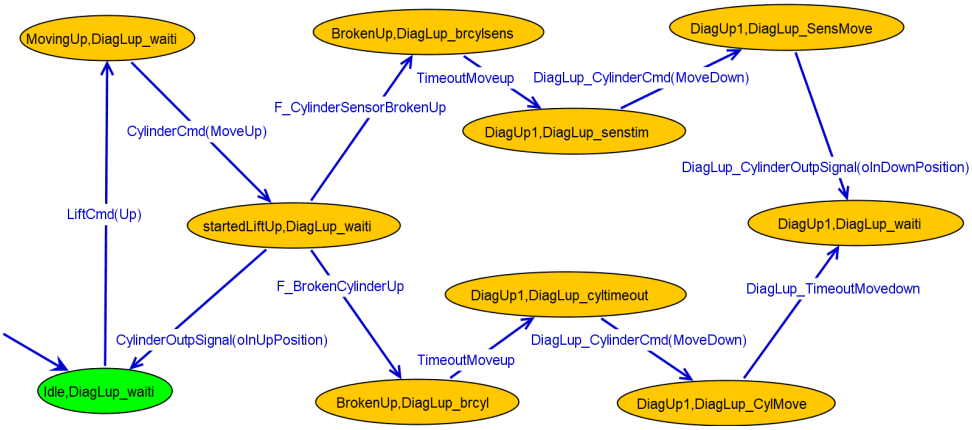


Figure 4.7: Composition of active DES-part and lift model for the ‘lift up’ failure

Calculating the active diagnoser An active diagnoser, automatically constructed from the composition, determines at run-time what the fault is that caused a failure. The diagnoser runs along with the original controller and takes all transitions that the controller also takes.

In Figure 4.8, the unobservable transitions (the faults) are no longer visible. Instead, the states of the diagnoser show the system's potential states. After a time out, the event TIMEOUTMOVEUP, two faults could have occurred, which is described by the state's label. $F1$ corresponds to fault in the cylinder, $F2$ to a fault in the upper lift sensor. The state after the transition TIMEOUTMOVEUP is an F-certain state. After giving the command to go down, the system's response uniquely determines which fault was the root cause of the failure.

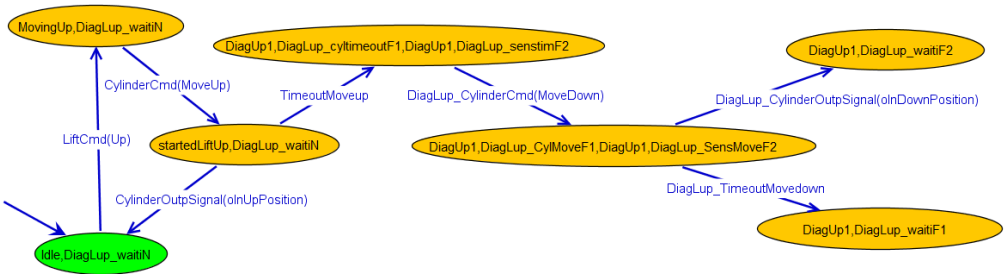


Figure 4.8: Diagnoser for the 'lift up' failure

Generating the active diagnoser requires several forms of information. The proposed method requires a set of faults per component. A system designer needs to create a model that adds faults to the state transition diagram of the system and needs to manually create the active diagnosers. The final comparison below uses the proposed method and the resulting model.

4.4 Comparison

This section compares Bayesian networks and DES diagnostic models based on the related literature described in Chapter 2 and observations made during the development of the models of the MerryGoRound. The comparison aims to determine the most promising method of failure diagnosis for systems controlled by machine control applications based on the criteria defined in Section 4.1.

Effort of modelling The effort of modelling depends on various factors and strongly depends on the information at hand. We assume that a machine control application specification for each system is available that contains state machine diagrams and class diagrams. Other information that might be available for the systems is the system design, including the physical connections of components, such as electrical wiring schemes or CAD drawings. Information on potential faults and failure modes may be available for some systems.

Both approaches need to know the faults and failures of the system. A Bayesian network uses faults in the nodes in the physical part of the model. Bayesian networks use capabilities (in a sense, the opposite of failures) in the model's capability part. DES diagnostic models use faults as transitions to a fault-certain state; these are states that are only entered after some fault occurred with certainty. The model contains the failures (e.g., time outs) as transitions from fault-certain states to states that start active diagnosis. Concluding, the faults and failure modes are a requirement for both approaches. The effort is, therefore, similar to gather this information.

To create a Bayesian network, the physical connections between components and their correlation are needed. It is unclear if it is possible to deduce these dependencies from the class diagrams and state machine diagrams. The parameters of a Bayesian network, the conditional probability tables, lead to additional required effort. There are two types of probability tables

required for the Bayesian network: the probability tables of faults in atomic components and the probabilities of faults in components that depend on other components' functioning. These probabilities are not directly available in a machine control application specification. An option would be to use default probabilities, but this will result in a less exact model.

The DES diagnostic model approach uses the existing state machine diagrams as a basis. Then, as mentioned before, faults have to be added manually. A system designer manually creates the active diagnosis part models in the DES diagnostic model proof of concept. The active diagnosis information is a requirement for the DES diagnostic model approach. The active diagnoser shows a diagnosing sequence of actions after the occurrence of a failure. A Bayesian network should be fed with evidence after a failure has occurred.

Summing up the required effort per approach, adding faults as transitions and creating active diagnosers is an additional effort of the DES diagnostic model approach. The Bayesian networks require additional information about probabilities of faults, the connection between physical components and capabilities. A system designer can handle the additional effort of creating DES diagnostic models. Bayesian networks depend on likely incomplete data on the probabilities of faults. This could be prevented by using default probabilities, but this results in a less exact model, a problem that DES diagnostic models do not have. Therefore DES diagnostic models have an advantage in the required effort of modelling.

Composability Bayesian networks consist of several nodes that together form a network. A group of nodes related to a single component can be coupled to the component after being created once. System designers reuse the physical components in various systems. Hence, engineers can reuse the group of nodes similarly in the corresponding Bayesian network. The size of the interface of the group of nodes, i.e., the edges to other nodes, determines the effort to integrate the Bayesian network of a component into a Bayesian network for a system.

In general, one could capture Bayesian network components in a library and use the smaller networks to compose a Bayesian network for each system. The composition mentioned above works well for the physical components. A challenging atomic component to model is the power supply of a system. The power supply has a causal physical connection to all other nodes that depend on the power delivered. The capability nodes are not based on physical components. They describe the hierarchy of system functions and are more based on the specific system, as could be seen in the proof of concept. The capability nodes make the approach used in the Bayesian network partly composable.

DES diagnostic models are, by definition, created as a composition of models. The approach of Sampath *et al.* [33] computes the synchronous composition of all components' behaviour models. The proof of concept, using a modular and active approach, has similar composability characteristics. The proof of concept avoids the full synchronous compositions of all components. It uses a diagnoser for each component, which is similarly composable. The faults and resulting failures, captured as transitions in a component model can be reused as-is in other systems. Similar to Bayesian networks, one can create a library of component models for the DES diagnostic model approach. This library makes the DES diagnostic model approach composable. Still, the DES diagnostic models have similar issues with components that depend on other components, like a power supply. Then one would have to specify what the influence of other components is on the behaviour of a component reducing the composability. Concluding, both can be composable depending on the interface between components.

Expressiveness The underlying assumptions of both Bayesian networks and DES diagnostic models impact their expressiveness. The DES diagnostic model method of Sampath *et al.* [33], describes the system as a set of components that each have a finite number of states and transitions between states. The approach used in the proof of concept shares the assumption that a set of models can describe a system. The original model of Sampath *et al.* [33] cannot diagnose

failures related to timing. Alternative models are available that can capture failures related to timing [36].

Another assumption made in Sampath *et al.* [33] is that the system consists of a two-level architecture with one central supervisory controller and several low-level controllers. The high-level, supervisory controller coordinates the low-level controllers. The low-level controllers are equipment controllers. The proof of concept uses the same two-level structure. Some methods relax this assumption [44], but relaxing this assumption impacts the effectiveness of the method. An additional assumption in the proof of concept is that only fault can occur at a time.

Bayesian networks do not require a specific architecture of the system. Bayesian networks only assume that the system can be described as a set of nodes describing a conditional probability table. This approach is less restrictive because it makes fewer assumptions. One notable assumption of the static Bayesian network is that it cannot express the impact of time on the variables as shown by the proof of concept.

Both methods share some assumptions. If a controller can no longer function, it cannot be determined by DES diagnostic models whether faults have occurred. A fault causing the power supply to fail, therefore, cannot be diagnosed. A Bayesian network can still function without the actual system running. However, since a system engineer cannot observe the system's properties (as the system no longer functions), the root cause cannot be determined either.

Concluding, the specific method used impacts the expressiveness of both Bayesian networks and DES diagnostic models. Most assumptions can be relaxed, but then a trade-off is made between expressiveness and effectiveness or scalability. As the two proofs of concept show, both models can model a typical MCA-controlled system. In general, however, Bayesian networks are more expressive.

Model adaptability A change in a system requires a change in the corresponding model. New faults, new components or structural changes in the system, result in a change in the structure of both Bayesian networks and DES diagnostic models. In that sense, they are similarly adaptable.

For the additional effort after a system change, DES diagnostic models in the classical sense have to regenerate the synchronous composition and diagnoser. This effort is highly automatable. The proposed method used for the proof of concept only requires a change in the model used for the changed component and is therefore adaptable. However, the active diagnosis parts of the model need to be changed if the expected behaviour of the system changes. The effort of changing the diagnoser depends on the size of the active diagnosis part of the model.

A Bayesian network requires more effort to be corrected after a change in the corresponding system because of its parameterisation. Bayesian network probabilities are learned by using historical data or determined by a domain expert. A change of the probabilities in some part of the system might impact the entire model. Therefore, historical data should be gathered to create a new Bayesian network based on the old model or a domain expert should redetermine the probabilities. Learning a Bayesian network, in general, is NP-complete, but in practice, current methods allow learning of models containing up to one million variables [8].

While it is possible to relearn Bayesian network parameterisation after a system change, this is still additional manual effort when compared to DES diagnostic models. Therefore DES diagnostic models are more adaptable.

Model understandability Model understandability is required for all manual efforts of an approach. Methods using DES diagnostic models require manual effort to add faults and failures as transitions and the creation of active diagnosis parts of the model. In general, the methods require knowledge about state machine diagrams and the basics of DES diagnostic model techniques to create the models. The Bayesian network approach uses probability tables. Bayesian networks, therefore, require an understanding of probability theory. In general, it cannot be decided what is more understandable, as the models require a different kind of knowledge. MCA-controlled system designers are experts on state machine models which may not familiar

with probability theory. Therefore DES diagnostic models are easier to understand for our target audience.

Diagnosis explainability Bayesian networks and DES diagnostic models give different kinds of explanations. Both methods give a list of causes after the occurrence of a failure. If the models start with the same information, diagnosing a system with the models should result in the same faults as causes. Bayesian networks show the propagation of a fault in the bottom layers of the model to failures in the upper layers. Service engineers manually add observations to the model. The model updates the probabilities of faults based on the observations. It is up to service engineers to determine the faults based on the probabilities of faults.

The shown approach in the DES diagnostic model proof of concept shows the system behaviour between faults and failures. The active diagnoser automatically excludes other faults that could have happened after a failure. If the active diagnosers are well designed, service engineers can follow the behaviour to get certainty in the verdict. Concluding, both give a good explanation, but personal preference determines which method is best.

Scalability As mentioned in the expressiveness, the scalability highly depends on the specific type of Bayesian network and DES diagnostic model chosen. Better scalability often reduces the expressiveness of the model and vice versa. As can be seen by the various attempts to improve both models' scalability, neither are very scalable in general. In this paragraph we will focus on the complexity since this generically determines the scalability of an approach independently of the usage of the model. Still we should note that the complexity does not always align with practical scalability.

Classical DES diagnostic model approaches have the problem of state explosion due to the synchronous composition and the diagnoser created. The synchronous composition of the models has in worst case the product of the number of states of each model. The diagnoser of the synchronous composition is exponential in size [35]. The approach used in the DES diagnostic model proof of concept avoids the complexity by combining a modular approach with the assumption of single faults, preventing the need for large synchronous compositions. Hence, only the creation of diagnosers is required. The complexity of the specific approach is not known.

A static Bayesian network cannot capture the impact of the history of the system on the current state of the system. In that case, a dynamic Bayesian network is required. Dynamic Bayesian networks have a high computational complexity due to the number of nodes. The inference of Bayesian networks is the same for static and dynamic Bayesian networks. Exact inference in general is NP-hard [8].

Both models are computationally complex but feasible in practice. There is insufficient information to draw a clear conclusion and we will classify both with 'Medium' scalability.

Effectiveness Effectiveness is related to diagnosability. Based on the observations at hand, a service engineer should determine what the root cause was. Based on our observations, Bayesian networks and DES diagnostic models seem similar in terms of effectiveness when following the methods used in the proofs of concept. Both approaches test capabilities of the system to diagnose the root cause. Bayesian networks require the addition of manual observation to the network, which might be generated by letting the system execute action commands. The DES diagnostic model approach automatically gathers observation based on the active diagnosers. One notable difference is that Bayesian networks could also allow physical observations that cannot be automatically gathered by the active diagnosers.

For the comparison of effectiveness, we used our observation on both models in the proofs of concept. A more in-depth analysis of the effectiveness would require the comparison of the effectiveness using a set of metrics, where both models are given identical information. Based

on our observations, the approaches' effectiveness is similar, but Bayesian networks can add physical observation at the small cost of more manual effort in diagnosing a failure.

Tool support There exist tools for the creation of both Bayesian networks and DES diagnostic models. Three tools were used to create proofs-of-concept. DESUMA [56], the free academic tool used for the DES diagnostic models, is less mature in use than the commercial tools Bayes Server [57] and Netica [58], used for Bayesian networks. The DESUMA tool, containing specific techniques for the creation of DES diagnostic models, is not regularly maintained and contains several bugs. Based on the used tools, the practical support for Bayesian networks is much better. There are several more generic tools available for state machine diagrams that can model DES diagnostic models but do not have all required techniques such as the automatic generation of diagnosers. Altova UModel in combination with Cordis Modeler [59] is a mature and commercial tool to create state machine diagrams.

4.5 Conclusion

This chapter answered Research question Q2: 'Which quality criteria determine how well diagnostic methods fit in the context of machine control applications?' and Research question Q3: 'Which method for diagnosing detected anomalies of industrial systems found by answering Research question Q1 is most fitting according to the criteria of Research question Q2?' The first question is answered by Section 4.1 with nine criteria. The remainder of the chapter answered Research question Q3 by comparing the two most promising methods found in the literature.

Table 4.1 summarises the strengths and weaknesses of both methods. Based on the comparison results, both are promising methods for the diagnosis of anomalies and failures. DES diagnostic models, fit systems controlled by MCAs best because of the smaller effort in modelling. Machine control applications use state machine diagrams as a basis, resulting in a more adaptable model. While Bayesian networks are more expressive in general, the additional restrictive assumptions made by methods using DES diagnostic models are not violated by systems controlled by a supervisory controller. For example, the two-layer controller architecture is followed by MCA-controlled systems. Finally, the practical tool support is better for Bayesian networks, but this does not outweigh the various advantages of DES diagnostic models over Bayesian networks. Therefore, we select DES diagnostic models as the most promising model for diagnosing systems controlled by a machine control application.

Criterion	DES diagnostic model	Bayesian network
Effort of modelling	High	Medium
Composability	High	High
Expressiveness	Medium/High	High
Model adaptability	High	Low
Model understandability	High	Medium
Diagnosis explainability	High	High
Scalability	Medium	Medium
Effectiveness	High	High
Tool support	Medium	High

Table 4.1: Comparison between Bayesian networks and DES diagnostic models

5 Pragmatic DES diagnostic approach

Chapter 4 shows that DES diagnostics is the most promising method of failure diagnosis in the domain of systems controlled by machine control applications. It shows that DES diagnostics is a good fit to MCA-controlled systems, but that the scalability in practice is restricted by the large models generated. This chapter describes a pragmatic approach based on the DES diagnostics literature and the proof of concept in Section 4.3. The main difference between the approach proposed in this chapter and the proof of concept is that we will separate the detection and diagnosis of failures and thereby are able to reduce complexity. We are able to use smaller models that result in improved run-time performance due to less memory usage, but also create a more flexible method in terms of failure detection. Since any failure detection method can be used, more types of failures can be expressed.

The proof of concept in Section 4.3 is based on the work of Sampath *et al.* [33]. The proof of concept uses the synchronous composition of models but avoids complexity by using a modular approach [44]. Additionally, the proof of concept uses active diagnosis to increase diagnosability. This chapter shows a practical approach that uses modular diagnosers, but avoids synchronous composition entirely. The approach applies to systems with manually specified machine control applications.

Section 5.1 describes a process of diagnosing a system in three phases and explains the components involved. One of these components is an active diagnoser; Section 5.2 defines active diagnosers, describes underlying principles and shows how active diagnosers can be created. Section 5.3 describes an architecture of the MCA-controlled system and components used to diagnose the system and Section 5.4 concludes the chapter with a summary.

The chapter answers Research question Q4: 'How can the method found by answering Research question Q3 be refined to use machine control application specifications and data from systems in the field?' and Research question Q5: 'Which diagnostic information of the abnormal behaviour should a method present to system developers?'

5.1 Diagnosis process

The approach proposed in this chapter avoids complexity of the proof of concept in Section 4.3 by eliminating the synchronous composition step entirely and making the passive monitoring and the active diagnosis two successive steps. Figure 5.1 shows the timeline describing the three phases of our approach. First, the method observes a system till a failure is detected. Subsequently, the method diagnoses the failure, resulting in a reduced set of potential faults. In the final phase, the method communicates the diagnosis to a system engineer.

We will use the term *passive monitor* to describe systems that detect failures. Passive diagnosers are examples of passive monitors. In our approach, the passive monitors detect anomalies or failures in the system and activate an active diagnoser. The active diagnoser actively commands the system to reduce the number of faults that could have caused the anomaly/failure, ideally to a single fault. Eventually, the active diagnoser notifies a service engineer of the fault(s) that (may) have led to a detected failure. Figure 5.2 illustrates the proposed process based on the communication between the components in each phase.

Passive monitor As shown in the related work in Chapter 2, various methods can detect anomalies and failures of a system. The proof of concept in Section 4.3 is limited to diagnosing

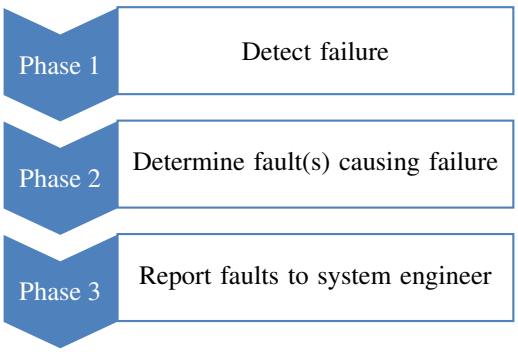


Figure 5.1: Timeline of our diagnosis approach

failures that the controller can detect. This proposed approach uses existing anomaly detection methods as a trigger for active diagnosis. The existing anomaly detection methods are passive monitors of the system. Passive diagnosers, as used in [33], can be used as a passive monitor. In the passive monitoring phase, the controller sends events to the passive monitor. The passive monitor initiates the second phase by activating an active diagnoser after the detection of a failure.

Active diagnoser The active diagnoser should know what faults can cause which failure. A passive monitor starts an active diagnoser with the detected failure and potentially with a set of potential faults, based on the monitor type. Similarly to the DES diagnostic model proof of concept, the active diagnoser tries to eliminate potential faults. It stops the controller and actuates the system, while listening to the events sent by the controller. In contrast to a mere passive approach, this means that the active diagnoser always has to run alongside a running controller. The active diagnoser initiates the third phase by communicating detected faults to the service engineer.

Controller The controller determines the behaviour of the system. It needs to forward the system's events, such as changes in sensor values and the commands it gives to the plant to the external monitors and the active diagnoser, when the active diagnoser is activated. The controller also needs to stop the existing behaviour when requested by active diagnosers. Then, it should forward the commands of the active diagnoser to the plant. While forwarding commands, the controller also needs to notify the active diagnoser of relevant events.

5.2 Definition of active diagnosers

In this section, we propose a definition of active diagnosers, explain underlying principles and propose how active diagnosers should be created. Active diagnosers are a special kind of the diagnosers that are described in Section 3.3.4. Whereas there is only one passive diagnoser in the approach of Sampath *et al.* [33], there are no such restrictions for active diagnosers proposed in our approach. An active diagnoser could be specified for each failure of the system model. The creation of active diagnosers leaves room for personal preference of system designers.

Definition 5.2.1 (Active diagnoser). An active diagnoser for system model $G = (X, \Sigma, \delta, x_0)$ is a finite state machine described by the four-tuple described in Equation 5.1. It contains a discrete set of states Q_{ad}, a set of events Σ_{ad}, the (partial) transitions function $\delta_{ad} : Q_{ad} \times \Sigma_{ad} \hookrightarrow Q_{ad}$, and $q_{ad0} \in Q_{ad}$, the initial state.

$$G_{ad} = (Q_{ad}, \Sigma_{ad}, \delta_{ad}, q_{0ad}) \tag{5.1}$$

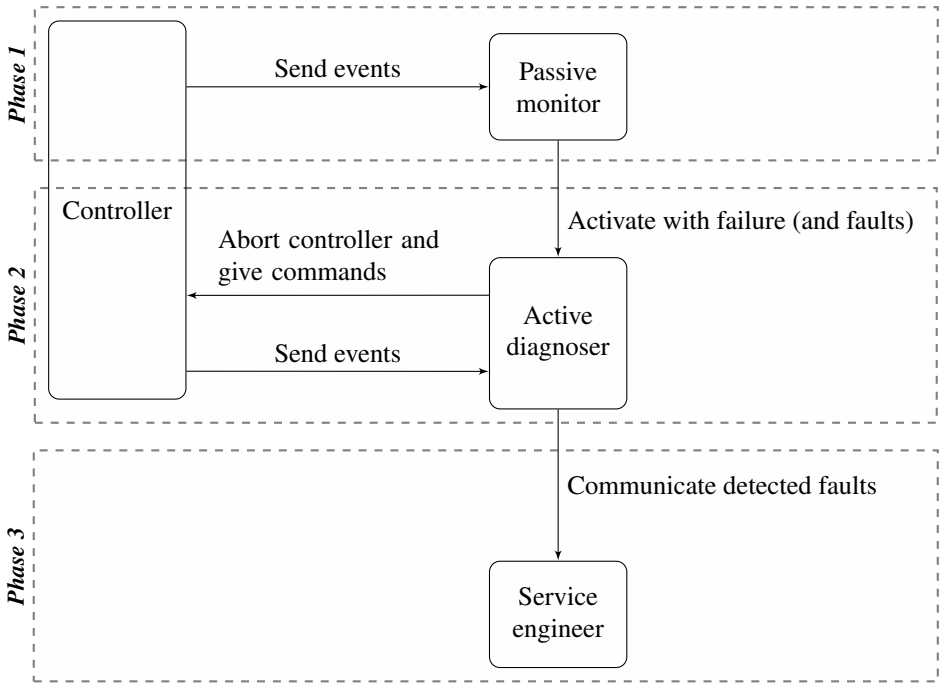


Figure 5.2: Process of diagnosis in proposed method

A state $q_{ad} \in Q_{ad}$ has the form $q_{ad} = \{(x_1, \ell_1), \ldots, (x_n, \ell_n)\}$ where $n \in \mathbb{N}, n \geq 1, x_i \in X$ is a state of G and ℓ_i is a label describing the occurred faults for that potential state x_i. The transition function itself is exactly the same as the transition function used in the passive diagnoser as described in Section 3.3.4; the active diagnoser responds to events that are observable for the controller. The main difference is the meaning of the events in Σ_{ad}. There is a difference between sensor value events and controller command events. Similar to the distinction made in supervisory controller synthesis, we use the notion of controllable and uncontrollable events.

Definition 5.2.2 (Controllable and uncontrollable events). An event set Σ of a MCA-controlled system is a disjoint union of controllable events Σ_c and uncontrollable events Σ_u, i.e, $\Sigma = \Sigma_c \cup \Sigma_u$ [60]. Events generated by the controller are controllable, while all other events are uncontrollable.

We will use the same distinction in the active diagnoser event set Σ_{ad}. Concerning the relating between observability and controllability, observable events are sometimes controllable. Command events are controllable events, while sensor value events are uncontrollable. Unobservable events, such as faults, cannot be observed by the controller and therefore also cannot be controlled, hence $\Sigma_{uo} \subseteq \Sigma_u$.

Definition 5.2.3 (Failure events). Active diagnosers use a new type of event, a *failure*. Failures are events that show that the system shows undesired behaviour. Σ_{fail} describes the set of failures. Similar to the fault set Σ_f, events in the failure set are not controllable, i.e, $\Sigma_{fail} \subseteq \Sigma_u$.

Failure events can either be observable or non-observable. A time out is a typical observable failure, since a controller can detect it. Failures that cannot be detected by the controller are unobservable failures. Passive monitors could detect unobservable failures by observing sequences of observable events.

Active diagnosis transitions A passive diagnoser makes state transitions based on the controller's commands and sensor values events received from the system. In our approach, active

diagnosers take over the control of the system from the controller. Active diagnosers stop the controller and start the active diagnosis process. The active diagnoser actuates the system and listens to sensor value events received from the system. The transition function determines which commands are sent based on received sensor value events and failures.

The initial state of the diagnoser is a singleton consisting of a state x_0 and the label $\emptyset$ resulting in $q_{ad_0} = \{(x_0, \emptyset)\}$. In the initial state, the active diagnoser waits for a passive monitor to detect a failure and communicate the failure (potentially with an already reduced set of faults). The first transition in the active diagnoser is always the failure given by the passive monitor, hence $\delta(x_0, \sigma)$ is only defined for $\sigma \in \Sigma_{fail}$. The second state should be a state that is inconclusive about the failure's root cause (the fault). Therefore it should consist of multiple states of the original model, each containing different faults. Please note that the label ℓ_i in diagnoser state tuple (x_i, ℓ_i) represents all faults that occurred before reaching x_i. If a diagnoser state is a singleton, the faults in ℓ_i are all faults that occurred.

We assume that the label corresponding to each system state contains at least one fault, i.e., for each state of the active diagnoser, except for the initial state, $\forall(x_i, \ell_i) \in q_i : \ell_i \neq \emptyset$ for all $i > 0$. The diagnoser's remaining states and transitions show a sequence of commands and resulting sensor values to diagnose a failure.

States in the active diagnoser Machine control application specifications do not contain a complete system model. The models in a specification represent the desired behaviour of a system, which is a subset of a system model that shows all possible behaviour of a system. We assume that the combination of the MCA specification in combination with the knowledge of the system engineer results in an implicitly available system model, i.e., a system designer knows the potential behaviour of a system. By not requiring a complete system model, we provide a diagnosis method that only requires a system designer to model the part of a system model that is required for the diagnosis.

Example 5.2.1. To illustrate the active diagnoser definition, Figure 5.3 shows the smallest active diagnoser possible, with a single failure and two faults. After the occurrence of some fault (either of type F_1 or F_2), a passive monitor detects a failure. The active diagnoser transitions to the state $\{(x_1, \{F_1\}), (x_2, \{F_2\})\}$. In this case, it is certain that some fault happened (as the failure is observable), but not which one. A command is given to the system by the active diagnoser. The sensor values returned give information about the unobservable faults that resulted in the failure, and the specific fault gets identified.

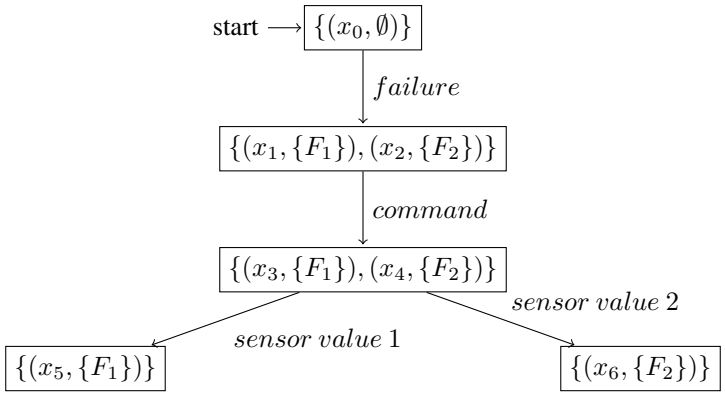


Figure 5.3: Example of an active diagnoser

Structure of an active diagnoser The active diagnoser in Figure 5.3 shows a single command and two different sensor values as system response. More generally, a diagnoser is an acylic

automaton which repeatedly generates sequences of *diagnoser actions* and responds to resulting *system responses*. Based on the system response, the active diagnoser sends different commands to exclude different faults. A diagnoser action may consist of more than one command to the system. Likewise, a system response is a sequence of one or more system events, which are sensor value events or controller events such as time outs. Figure 5.4 defines the structure of an active diagnoser automaton in Backus-Naur form [61], where the states are implicit.

The first event in an active diagnoser is a failure that should be diagnosed. Next, a diagnoser action is executed to determine which fault occurred. The system response gives information about the possible faults. Based on the system response, the diagnoser starts different diagnoser actions. The failure diagnosis rule uses the informal + symbol to shows that there are multiple system responses possible after a diagnoser action. The diagnosis finishes when the set of potential faults cannot be reduced any further, ideally this fault set contains a single fault. While the potential faults set can still be reduced, the active diagnoser will repeat the diagnoser action-system response sequence.

$$
\begin{array}{lll}
\langle \textit{Active diagnoser} \rangle & ::= & \langle \textit{failure event} \rangle \ \langle \textit{failure diagnosis} \rangle \\
\langle \textit{failure diagnosis} \rangle & ::= & \langle \textit{diagnoser action} \rangle \ (\langle \textit{system response} \rangle \ \langle \textit{fault determination} \rangle)^{+} \\
\langle \textit{fault determination} \rangle & ::= & \langle \textit{show minimal fault set} \rangle \\
 & | & \langle \textit{failure diagnosis} \rangle \\
\langle \textit{diagnoser action} \rangle & ::= & \langle \textit{command event} \rangle \mid \langle \textit{command event} \rangle \ \langle \textit{diagnoser action} \rangle \\
\langle \textit{system response} \rangle & ::= & \langle \textit{system event} \rangle \mid \langle \textit{system event} \rangle \ \langle \textit{system response} \rangle \\
\langle \textit{system event} \rangle & ::= & \langle \textit{controller event} \rangle \mid \langle \textit{sensor value event} \rangle
\end{array}
$$

Figure 5.4: Syntax of an active diagnoser in Backus-Naur form

5.2.1 Creation and generation of active diagnosers

In this section, we describe how system engineers should create active diagnosers. As described in the previous section, we assume that there is no complete system model. Therefore, we propose to create diagnosers manually or to create a partial system model and generate a diagnoser from this model. The advantage of manually creating an active diagnoser, is that the active diagnoser itself is a relatively compact finite state machine compared to its system model. On the other hand, a system model might be easier to understand than an active diagnoser model because it does not require knowledge about active diagnoser and their state labeling. Additionally, using the system model helps in keeping consistent models, if more active diagnoser are created, i.e., the active diagnoser should be consistent in terms of actions and the expected system response.

Manual creation The manual creation by a system designer requires an understanding of active diagnosers. The system engineer follow the definition of active diagnoser and creates states with correct labeling. He has to keep track of the potential faults at each diagnoser state and which diagnosis actions can reduce the set of faults.

Defining a part of the system model A partial system model for an active diagnoser describes system traces after the occurrence of each fault and the resulting failure. Each trace starts with

a fault and then repeats a sequence of commands and returned sensor values. Please note that the trace in a system model can only differ based on uncontrollable events, the system events. Figure 5.5 shows an example of a system model. The model, not coincidentally, generates the active diagnoser of Figure 5.3. We use the generation procedure of Sampath *et al.* [33] for the generation of active diagnosers from system models.

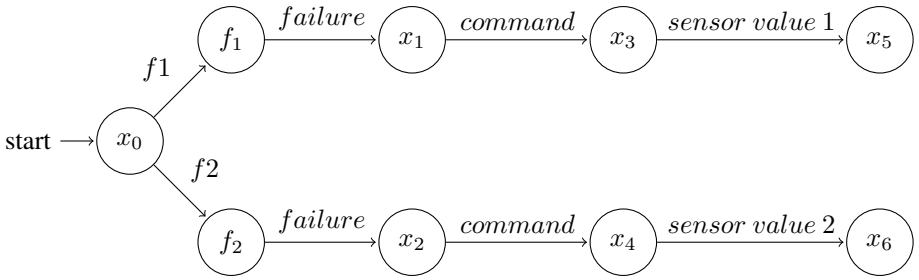


Figure 5.5: Example of a partial system model for an active diagnoser

5.3 Architecture selection

This section describes the selection of an architecture that combines a controller with components used to diagnosis the system. Input for the selection are meetings with experts on system modelling from TNO and controller experts from Cordis Automation. The first meetings resulted in a shortlist of three potential architectures. Based on feedback on the proposed architectures, we gathered the advantages and disadvantages. This section describes the three architectures, including their strengths and weaknesses, and ends with a choice for an architecture. The original system is shown as Cordis Suite, as this is the machine control application design suite used as an example throughout this chapter. The approach, however, is not specific to Cordis Suite.

Architecture 1: External active diagnoser using controller The first architecture uses an external passive monitor and active diagnoser. Figure 5.6 shows the architecture. The controller sends plant events and internal events, such as time outs, to passive monitors and active diagnosers. If no failure occurs, active diagnosers does not use the events. Passive monitors continuously monitor the events from the controller till they detect a failure. When a passive monitor detects a failure, it starts an active diagnoser with the detected failure as input. An active diagnoser stops the system and sends a sequence of commands that reduce the number of possible failure-causing faults. Subsequently, the active diagnoser communicates the resulting diagnosis to a service engineer.

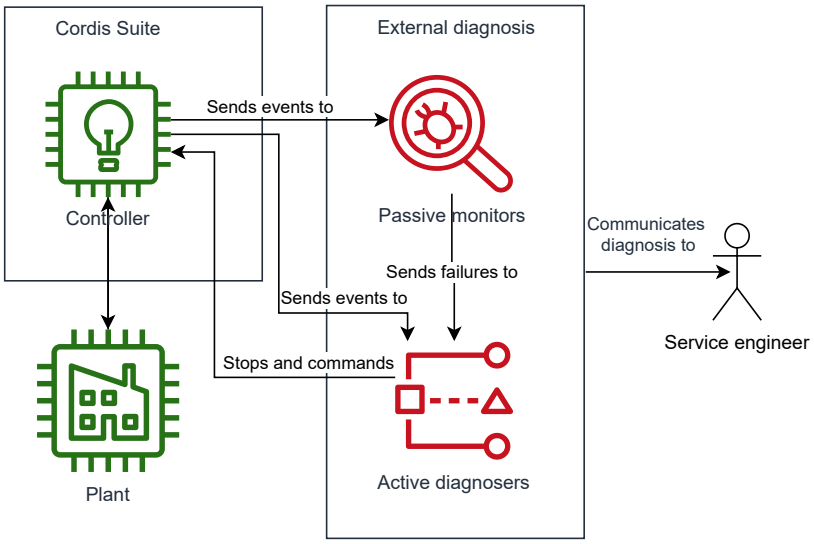


Figure 5.6: DES diagnostic model architecture 1: external active diagnoser using controller

This first architecture requires a controller that can forward events from the plant to the external diagnosers. The controller must also have the functionality to forward the active diagnoser's commands to the plant, including a stop command. The external tool or tools that diagnose the system must also be developed.

Architecture 2: Internal active diagnosis, external passive diagnosis The second architecture uses the same strategy for passive monitoring as Architecture 1, but has a different active diagnosis strategy. The state machine diagrams used to define controllers can also define sequences of actions that need to be executed by an active diagnoser to reduce the number of potential faults. Figure 5.7 shows the second architecture. Instead of a single controller, the diagram shows a set of controller models. The controller models together form the controller that controls the plant. The active diagnoser is a type of controller model. Similarly to Architecture 1, the passive monitor listens to the events sent by the controller. Instead of sending the failure to an active diagnoser, the passive monitor sends the failures to the controller. The controller activates the appropriate active diagnoser. The presentation of faults and resulting failures to service engineers is also different from Architecture 1. As the active diagnosers are part of the controller, it more sensible to use the existing human machine interface to present the diagnosis information.

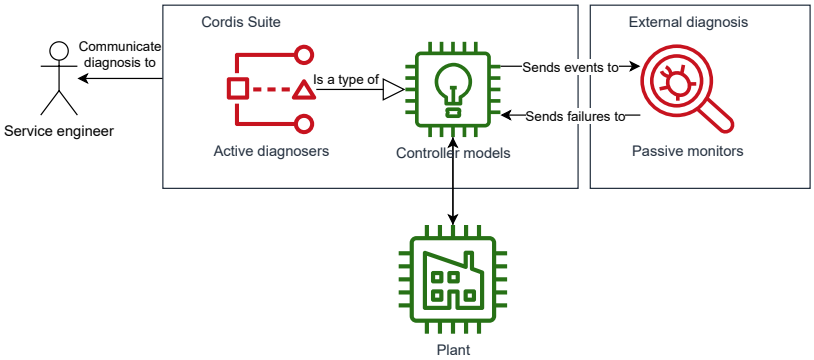


Figure 5.7: DES diagnostic model architecture 2: internal active diagnosis

Architecture 3: Direct plant control The third architecture is a variation on the first architecture. Instead of using the controller to send commands, it is also possible to directly send commands to the plant. The direct communication reduces the dependency on the controller and its functionality. Figure 5.8 shows the third architecture. Similarly to the other architectures, the external passive monitor listens to the events of the controller. After detecting a failure, the passive monitor starts the active diagnoser with as input a list of potential faults as root cause. Next, the active diagnoser first halts the controller and then sends the commands directly to the plant.

For the third architecture, external tooling must be created to passively detect failures and actively diagnose the system. The controller only needs to needs the functionality to stop on request of the active diagnoser. Additionally to the first architecture, the active diagnoser needs to know the plant's interface and needs to send the commands according to the corresponding protocol.

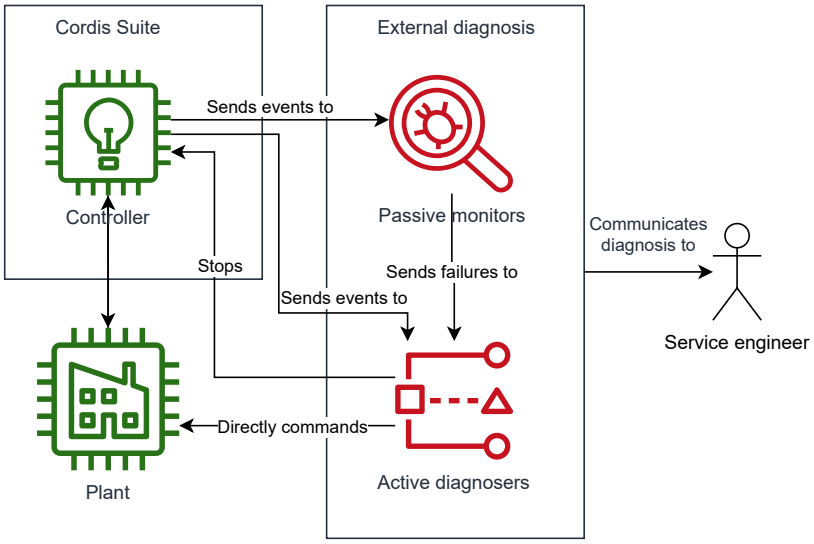


Figure 5.8: DES diagnostic model architecture 3: direct plant control

5.3.1 Architecture comparison and selection

In this subsection, we will compare the three architectures and select the best one. The third architecture has the advantage that the controller does not need to forward commands. Using a controller, however, provides abstraction on the used type of PLC. Machine control applications run on various types of hardware and control various types of systems. The third architecture is infeasible in practice because the active diagnoser needs to know the interface of each of these systems; something that is already solved by the machine control applications. The only advantage of the third architecture is, therefore, also its biggest weakness. The third architecture is not chosen because of this impracticality.

The other two architectures both have their advantages and disadvantages. The first architecture is more flexible because the active diagnosers are external. An update of the active diagnoser only means that the active diagnoser models change, without changes to the controller. In the second architecture, a change of the diagnosers means that the controller needs to be changed. The first architecture can be connected to the system for existing systems, while the second architecture requires a change of the existing system. Architecture 1 is also more flexible because it does not depend on models in a specific modelling environment.

Another advantage of the first architecture is the possibility to create overarching models. Typically the models created to control physical components only allow the control of the specific physical component. While this results in a clear hierarchy of the models and components, it might be required to give commands to several components for the diagnosis of faults. By using the causal relation between components, faults might be detected. This is easiest for the first architecture. Extending the argument, if multiple controllers need to be used, an external active diagnoser could send commands to multiple controllers. An additional advantage is that the PLC does not have to run additional code, which is the case with the internal active diagnosers.

An advantage of the second architecture is the interface of the active diagnoser to the controller. External diagnosers can only make use of the functionality exposed by the controller's external interface. The system engineer can give internal diagnosers any functionality of the controller. This information-hiding advantage might also be called a disadvantage because it gives a high coupling of the active diagnoser and the rest of the controller.

Given these deliberations, a hybrid between Architectures 1 and 2 is selected. We use the first architecture as basis because it is more flexible, but we allow changes to the controller as described in Architecture 2 and also use Cordis Suite to communicate a diagnosis to the service engineer through their Cordis Dashboard. The presentation will be discussed in more detail in Section 5.3.3. The changes required to the controller are case-specific.

5.3.2 Improving practicality with a model library

We assume that the models used to diagnose discrete-event systems are created based on the existing MCA specification models. MCA specification models often determine the behaviour of a physical component. When components of a system and their corresponding models are reused, models of the monitors and diagnoser can also be reused. To improve the practicality, we propose a library of models to be used in the design process. Typically, the following steps need to be taken by a system (diagnosis) designer:

1. Create or generate passive monitors that can determine failures,
2. Determine the faults that can explain each failure,
3. Create active diagnosers that command the system and listen to the events of the controller.

Passive monitors detect failures in a single component. Hence, passive monitor logic for a system component can be used when this component is reused in another system. Passive monitors know the faults and failures of the system to diagnose failures. Hence, the (models of) passive monitors can be stored in a library. For simple monitors, one could create a mapping between events and failures (i.e., describe the events they should listen to). For complex monitors, such as passive diagnosers one could save the models in a library.

The active diagnoser models could similarly be captured in a library. Figure 5.9 shows how models from a component library are used in a controller. The controller's used components determine which models from the passive monitor and active diagnoser libraries should be used.

5.3.3 Presentation of diagnosis

The active diagnosis process illustrated in Figure 5.2 shows that an active diagnoser communicates a diagnosis to a service engineer. This section proposes what the active diagnoser should communicate to the service engineer.

A service engineer typically observes a system through a dashboard. Such a dashboard gives insight into variables and errors in the system and allows the service engineer to change the system's behaviour. These variables of the system might help in understanding the diagnosis of the active diagnoser. Hence, as Figure 5.9 shows, the active diagnoser should communicate

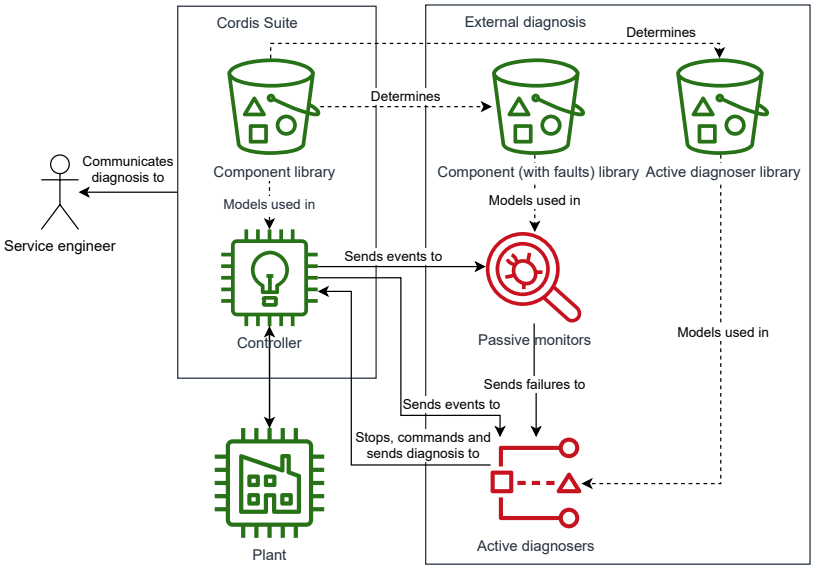


Figure 5.9: Chosen DES diagnostic model architecture

the failure and faults to the controller which in turn sends it to a dashboard. This last step is not shown in the Figure 5.9 to keep the figure simple.

The diagnosis of an active diagnoser results in a reduced set of faults and the caused failure. This information is the bare minimum that the active diagnoser should communicate to a service engineer. The diagnosis can be extended with related variables of the controller to improve understanding of the failure. Additionally, the active diagnoser also knows *how* the fault(s) resulted in the failure. This information could be presented in the form of a sequence of actions, taken by the active diagnoser.

5.4 Conclusion on practical approach

This chapter answers Research question Q4: 'How can the method found by answering Research question Q3 be refined to use machine control application specifications and data from systems in the field?'. We propose a method consisting of three steps. The passive monitors listen real-time to the data from the systems in the field to detect failures. Passive monitors consequently start an active diagnoser. Active diagnosers try to reduce the set of potential faults by giving commands to the physical system and observing the system's events. In the final step, the active diagnoser communicates the diagnosis to a system engineer.

The chapter also answers Research question Q5: 'Which diagnostic information of the abnormal behaviour should a method present to system developers?' Diagnosis information of the abnormal behaviour (failures) are shown through the typical dashboard used by the system. The shown diagnosis minimally contains the reduced set of faults and the resulting failure and can be extended with information such as related variables of the controller and traces of the system model to describe how the fault(s) resulted in a failure.

6 Validation of the proposed method

As described in the thesis objective in Section 1.3, the thesis' goal is not to create a full-fledged diagnosis tool. The goal is instead is to provide the information to design and create such a tool. This chapter describes a prototype that can detect failures and diagnose them by actively giving commands. We use our approach as described in Chapter 5. The prototype shows the validity of the proposed approach in Chapter 5. The chapter answers Research question Q6: 'Does the refined method of anomaly diagnosis found by answering Research question Q4 satisfy the criteria identified in Research question Q2?'

6.1 Implementation example

The prototype uses the MerryGoRound as described in Figure 3.6 as diagnosis example. The MerryGoRound is an example system that is available as a simulator. The diagnosis focuses on lift failures, specifically on failures resulting from the UP command. Figure 6.1 shows the lift's behaviour after the UP command. The lift's normal behaviour is to command the cylinder to go up, and after some time, detect that the cylinder is in the upper position. Equation 6.1 shows the sequence of events resulting from normal behaviour. The model in Figure 6.1 also shows two potential faults: the cylinder or upper sensor can be broken. Equations 6.2 and 6.3 show the effect of the unobservable faults F_CYLINDERSENSORBROKENUP and F_BROKENCYLINDERUP on the behaviour of the lift.

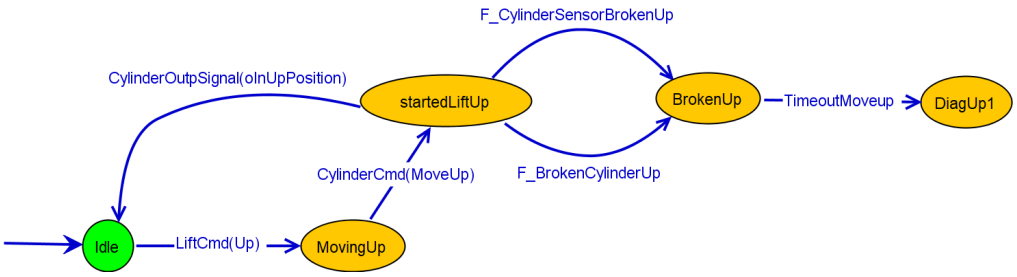


Figure 6.1: Lift behaviour after an UP command in Figure 4.5

Requirements for the passive monitor The normal lift event sequence differs from the fault behaviour sequences. A passive monitor can detect a failure by checking the TIMEOUTMOVEUP event. The observable behaviour of fault sequences are the same for both faults: a time out occurs. Since the event sequence is the same for both faults, a passive monitor cannot determine the fault that caused the time out. The passive monitor needs to start an active diagnoser to determine the fault.

$$\textit{LiftCmd(Up)} \rightarrow \quad \textit{CylinderCmd(MoveUp)} \rightarrow \quad \textit{CylinderOutpSignal(oInUpPos)} \tag{6.1}$$

$$\textit{LiftCmd(Up)} \rightarrow \quad \textit{CylinderCmd(MoveUp)} \rightarrow \quad \textit{TimeoutMoveup} \tag{6.2}$$

$$\textit{LiftCmd(Up)} \rightarrow \quad \textit{CylinderCmd(MoveUp)} \rightarrow \quad \textit{TimeoutMoveup} \tag{6.3}$$

Requirements for the active diagnoser The active diagnoser needs to command the system such that the response of the plant results in two different traces for the two different faults. We use and create the active diagnoser as defined in Chapter 5. Similar to the proof of concept in Section 4.3, the active diagnoser commands the lift to go down after a failure and waits for the system's response. A time out means that the cylinder is broken, while a 'lift is down' events means that the upper sensor is broken. As proposed in Section 5.2.1, we generate an active diagnoser based on a partial system model, based on the machine control application specification. The active diagnoser model shown in Figure 6.2 is created based on the partial system model in Figure 4.6. After a diagnosis, the active diagnoser needs to communicate the fault to the system engineer through a dashboard.

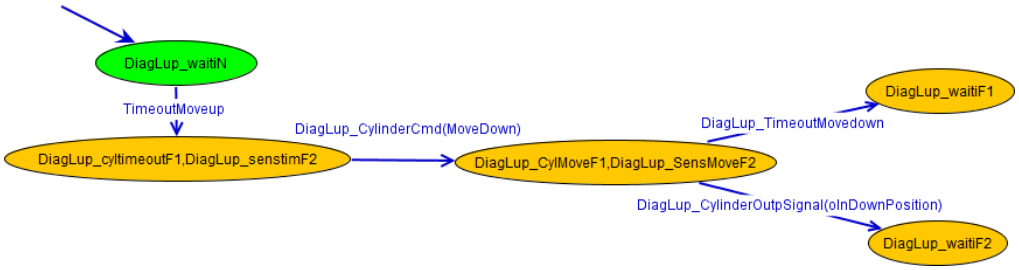


Figure 6.2: Active diagnoser for lift going up

Requirements for the prototype The software prototype needs to implement passive monitors and active diagnosers. The passive monitor listens to the TIMEOUTMOVEUP from the controller and next commands the active diagnoser to start. The active diagnoser halts the system and sends the MOVEDOWN command and waits for a response from the system.

For this, the prototype needs to listen to controllers and interpret the following events:

- TIMEOUTMOVEUP
- CYLINDEROUTPSIGNAL(OINUPPOS)
- TIMEOUTMOVEDOWN
- CYLINDEROUTPSIGNAL(OINDOWNPOSITION)

The prototype needs to send the command:

- CYLINDERCMD(MOVEDOWN)

Additionally, the active diagnoser also needs to stop all system components before starting the diagnosis.

Broken cylinder scenario To illustrate the active diagnoser's usability, we inject a fault in the lift in the controller of the MerryGoRound. For this, a system operator changes a setting in the controller that causes the cylinder or the upper lift sensor to break down. The passive monitor continuously listens to the controller to detect time outs of the lift. After it detects a time out, the passive monitor notifies the active diagnoser. The active diagnoser commands the lift to go down, which results in a second time out. The active diagnoser notifies the service engineer of the fault in the cylinder. Figure 6.3 illustrates the scenario.

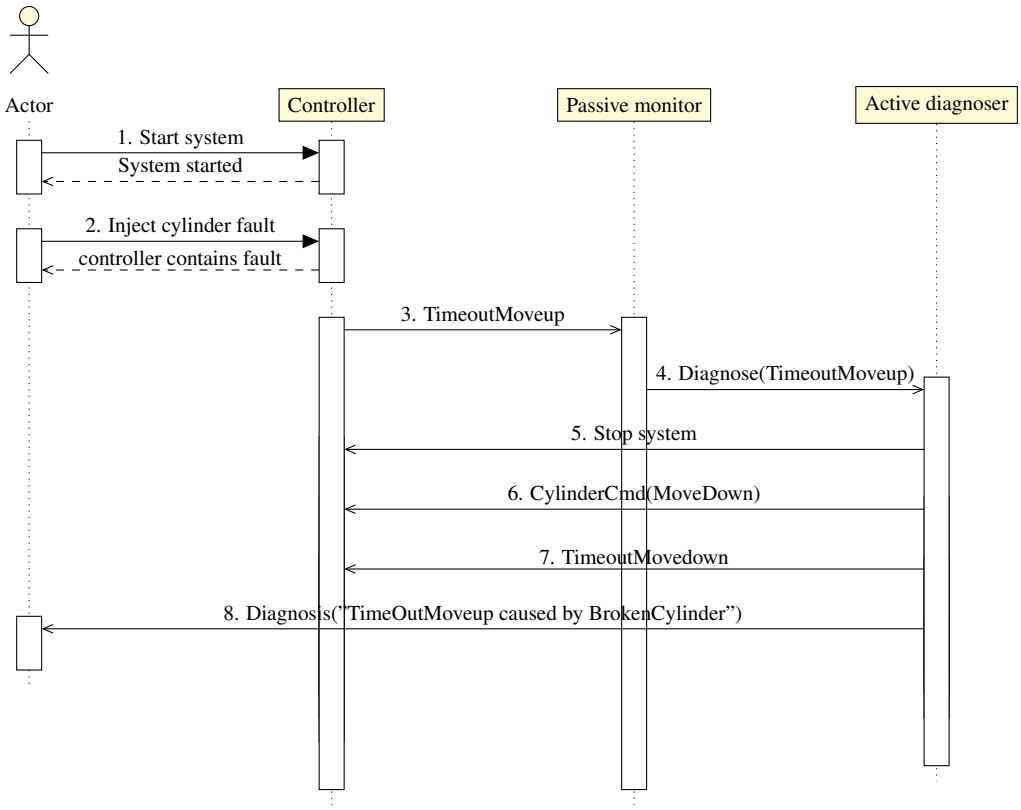


Figure 6.3: Broken cylinder scenario of lift diagnosis

6.2 Prototype

We created the diagnosis prototype as a plugin to the Machine Control Server of Cordis Suite. The plugin, *Acdiag*, is written in the programming language C# and contains all diagnosis logic, i.e., the passive monitors and active diagnosers. Figure 6.4 shows how the plugin relates to the Cordis Suite components. The diagnosis plugin Acdiag communicates with the Cordis Machine Control Server (MCS). The MCS checks during startup which plugins are specified and provides an interface to plugins. The plugins can subscribe to all events of a controller and can send commands to the controller.

6.2.1 Fault injection

As there is no failure diagnosis without failures, faults must be injected at some place in the system. Since the simulation does not support faults, the easiest way is to 'inject' these faults in the controller. A changed version of the MerryGoRound controller supports faults as a setting. When a fault is activated, the controller changes its behaviour to show externally visible behaviour as if a fault manifested itself in the simulation. During run-time, the faults can be activated through the dashboard. The two faults, a fault in the cylinder and a fault in the sensor, are achieved by ignoring certain events. For a broken cylinder fault, the controller ignores the commands to go up, which results in the cylinder never moving. For the broken sensor fault, the controller ignores the sensor events of the upper lift sensor. The lift still goes up, but the upper lift sensor is ignored. Since the lift model of the controller never receives a sensor event, it generates a time out.

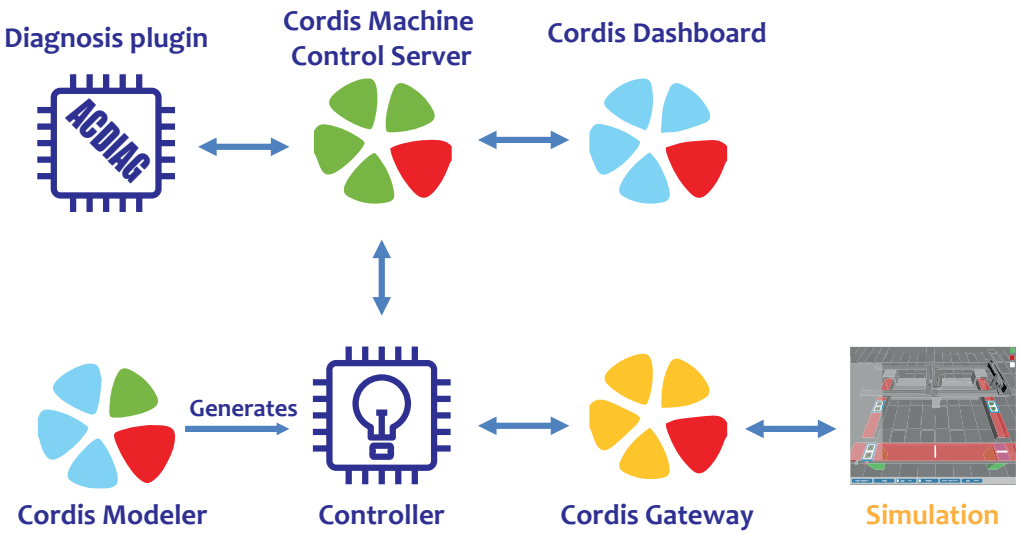


Figure 6.4: Overview of the diagnostic plugin in Cordis Suite

6.2.2 Implementation of passive monitors and active diagnosers

The plugin interface of the Cordis MCS is implemented as a publish-subscribe framework. The diagnosis plugin becomes a subscriber of events of the MerryGoRound. During each scan cycle of the controller, a callback function of the diagnosis plugin is called with the events that occurred. The diagnosis plugin provides a mechanism where classes can subscribe to specific data sent by the controller. The passive monitors and active diagnosers are subscribers to their desired data. The plugin contains an object-oriented framework of passive monitors and active diagnosers consisting of several classes, the most relevant classes and function are shown in Figure 6.5. Abstract classes and functions are shown in *italic*.

Passive monitors We use a typical object-oriented approach to handle multiple passive monitors and active diagnosers. The plugin contains an abstract class called ABSTRACTPASSIVEMONITOR. It describes the required functions of a passive monitor class and also provides some general functionality. Each passive monitor must be created with the component that it listens to as an argument. For instance, for the MerryGoRound example, the passive monitor only listens to the lift component's events.

The implementation of the SIMPLELIFTMONITOR that detects the TIMEOUTMOVEUP failure is straight-forward. It listens to events and starts the specified active diagnoser when the event called TIMEOUTMOVEUP is received.

Active diagnosers Similarly to the passive monitor setup, each active diagnoser is implemented as a subclass of an abstract class called ABSTRACTACTIVEDIAGNOSER. The subclasses are also required to define the component of interest.

The plugin contains a state machine framework that can be used by the active diagnosers. The framework consists of state classes and a context class. The context class is given to each state class and provides the functionality to change to a different state and send commands to the controller. In Figure 6.5, the dependency from STATE to STATE represents that state classes know other state classes to transition to. Each state class describes an ONENTRY function and a HANDLEEVENT function.

Each active diagnoser subclass describes its initial state, which describes the logic when an active diagnoser is deactivated. The dependency in Figure 6.5 from ACTIVELIFTDIAGNOSER

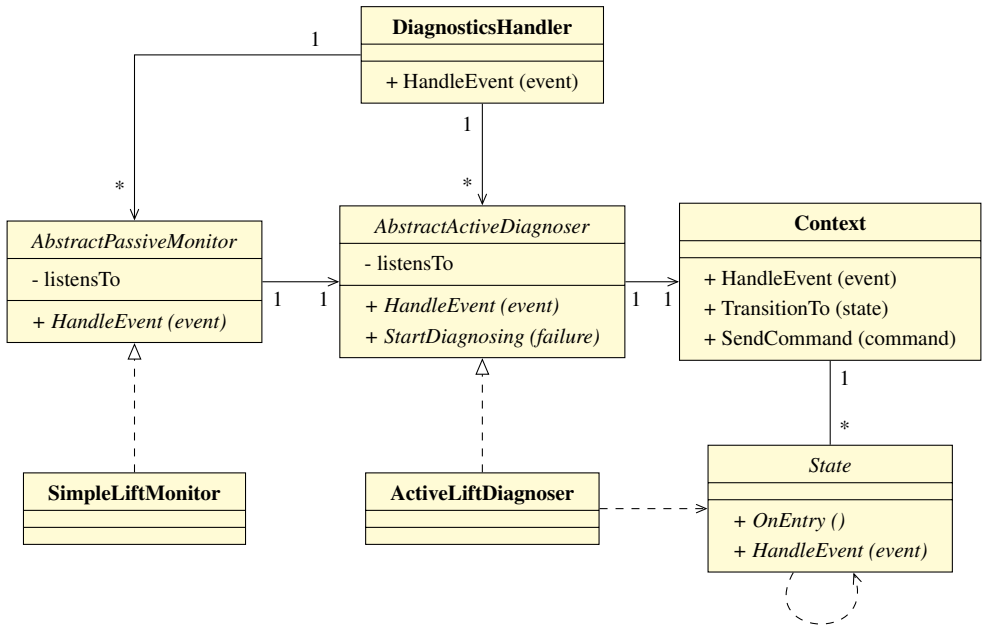


Figure 6.5: Class diagram Actdiag

to STATE represents the relation of the diagnoser to its initial state class. A STARTDIAGNOSING function in the active diagnoser changes the initial state to the first diagnosis state. The state classes contain the diagnosis logic. The specific active diagnoser for the lift going up failure, the ACTIVELIFTDIAGNOSER, is implemented as is defined in Figure 6.2 with additional events to stop the system after a failure and to reset the system after diagnosis.

Diagnostics handler The diagnostics handler provides the mechanism to provide the correct events to the passive monitors and active diagnosers. The passive monitors and active diagnosers all have a callback function to handle events. The diagnostics handler contains active diagnosers and passive monitors and defines the relation between passive monitors and active diagnosers. For each event received from the controller, the diagnostics handler uses the procedure defined in Algorithm 1.

Algorithm 1 Handling controller events

```
1:  monitors: list of passive monitor objects
2:  diagnosers: list of active diagnoser objects
3:  procedure HANDLEEVENT(event)
4:      for all monitor in monitors do
5:          if monitor.listensTo(event) == true then
6:              monitor.HandleEvent(event)
7:          end if
8:      end for
9:      for all diagnoser in diagnosers do
10:         if diagnoser.listensTo(event) == true then
11:             diagnoser.HandleEvent(event)
12:         end if
13:     end for
14: end procedure
```

6.3 Results

The created prototype using our proposed approach can detect the specified lift failure and, more importantly, can distinguish the potential faults preceding the failure.

During implementation, we made a small change in the model presented in Figure 6.2. We expected that a cylinder fault would always result in a time out. If the lift is in the bottom position and gets the command to move down, its position does not change, and therefore, we assumed that the controller would not generate a sensor value event. This is assumption was incorrect; the controller directly generates a 'lift is down' event when the lift gets the command to go down. Therefore, the active diagnoser also checks for an immediate sensor value after the command to go down is issued. The other transitions are implemented as described. The resulting state machine diagram is shown in Figure 6.6.

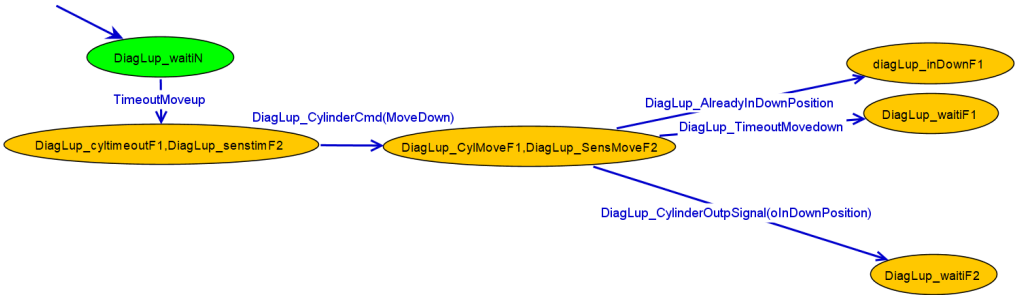


Figure 6.6: Implemented lift diagnoser in Acdiag

In the following paragraphs, we will check the proposed approach against the criteria described in Section 4.1 and will compare our approach to classical DES diagnostics.

Effort of modeling The prototype shows that with our approach, failures can be diagnosed by specifying simple passive monitors and active diagnosers without much effort. There is no need for complete system models; the passive monitors and active diagnosers are small, local models. It is sufficient to model the part of the system model that contains the fault traces, based on the MCA specification. The domain knowledge comes from the MCA specification and from the system designer. The proposed approach reduces the effort of modeling compared to classic DES diagnostics, because it does not require a complete system model.

Composability The proposed method is composable because the monitors and diagnosers are based on single components. Together these models form a complete suite for the detection and diagnosis of failures. Therefore the composability is similar to that of classical DES diagnostics, since it also uses a model for each system component.

Expressiveness The active diagnosers and passive monitors provide sufficient expressiveness to define the required functionality to detect a simple failure, but the functionality can be extended to support more complex failures. The expressiveness of our active diagnosers are comparable to the passive diagnosers used by Sampath *et al.* [33]. Still, since more types of passive monitoring can be used, our approach could express more types of failure. In our prototype we show that we can also use events from the controller (the timeouts), something that is not possible in classical DES diagnostics. It should be noted that we only used a simple passive monitor and the usage of more complex passive monitors is future work.

Model adaptability The adaptability of the models is improved compared to the classical DES diagnostic approach. The active diagnosers are small models that can be changed and

extended with the required functionality when a system component changes. A change in a system, such as introducing a component, only results in adding additional passive monitors and active diagnosers. This shows that our approach is more adaptable in practice than classical DES diagnosis, since this would require the regeneration of the passive diagnoser from the system model.

Model understandability and diagnosis explainability The models' understandability and the diagnosis explainability were not checked with end-users, because these were unavailable due to the COVID-19 pandemic. Instead, we checked these criteria with machine control application experts working at Cordis Automation, who have end-user knowledge. The experts found the diagnosis models to be understandable and found the set of faults given as diagnosis result to give an understandable explanation of the failure. Moreover, the diagnosis models use concepts similar to those in machine control application specification models, which the end-users are used to. Since we use similar concepts, we assume that the understandability of both the model and the diagnosis are similar to classical DES diagnostics.

Scalability The scalability of our approach is promising. Since the diagnosis tooling is external, various monitors and diagnosers can be defined and used without impacting the system behaviour. The computational complexity of the used models is low because they are smaller than the models using the classical DES diagnostic approach. By using modular diagnosers opposed to one monolithic diagnoser, we reduce the total size of the diagnoser because the size of diagnosers is exponential in terms of the state of the original system model.

Effectiveness While our approach is more scalable than classical DES diagnostics, the prototype still shows that our approach can determine faults that cause the detected failure effectively. Still, we have to note that we used a simple failure and it is future work to determine how well the approach scales when used with more complex failures and (bigger) real-life systems.

6.4 Conclusion on implementation

The chapter answered Research question Q6: 'Does the refined method of anomaly diagnosis found by answering Research question Q4 satisfy the criteria identified in Research question Q2?' The prototype can detect and diagnose a simple failure that is introduced by injecting a fault in a controller. The prototype shows that the proposed method satisfies most of the criteria of Research question Q2. The usability was checked with machine control application from Cordis Automation, who have end-user knowledge. It is future work to the usability with end-users. Similarly, the scalability should be checked against a real-life system instead of a simulator.

7 Conclusion

This thesis proposed a semi-automated method to diagnose failures in industrial systems controlled by machine control applications. The thesis started with the observation that a machine control application specification could provide (a part of) the required domain knowledge to create a model to solve an ongoing issue in the industry. A diagnosis model based on domain knowledge can quickly diagnose the root cause of a failure and reduce the duration of unplanned production interruptions. This chapter gives a thesis summary in Section 7.1, discusses the research results in Section 7.2 and proposes future work in Section 7.3.

7.1 Summary

This section gives an overview of the steps taken to establish the proposed method, summarises answers to the research questions, and presents an overview of the main contributions.

The thesis starts with giving an overview of existing methods that use domain knowledge to diagnose anomalies, where the focus was on hardware failure as anomaly. Additionally, it shows several existing methods to detect failures as background information. Chapter 2 answers Research question Q1: 'What are existing diagnostic methods for the diagnosis of anomalies in industrial systems?' There are several methods that create a diagnostic model based on domain knowledge. The found methods primarily use Bayesian networks, DES diagnostic models and decision trees.

To prepare the reader for the remaining chapters, Chapter 3 gives an introduction to discrete-event systems in general, an introduction to Cordis Suite and an introduction to discrete-event system diagnostics. Discrete-event systems are a type of systems characterised by a discrete state set and event-driven behaviour. Cordis Suite provides the tools to specify controllers of discrete-event systems. Discrete-event system diagnostics is an approach that can diagnose failures in discrete-event systems. The approach describes the normal and faulty behaviour in a single model, which can be used to diagnose the system.

Next, the thesis describes a comparison between the methods found in answering Research question Q1, to select the best fitting model found in the literature. One of the three models featured in the overview, decision trees, was not considered in the comparison because the decision tree methods are primarily data-based. We created a set of criteria to guide the comparison. The criteria are shown in Section 4.1 and describe the method's usability together with the expressiveness and effectiveness of the model. The criteria answer Research question Q2: 'Which quality criteria determine how well diagnostic methods fit in the context of machine control applications?'

Besides the criteria, Chapter 4 describes proofs of concept of the application of Bayesian networks and the application of DES diagnostic models to the same diagnosis problem. Based on the criteria and the proofs of concept, Research question Q3: 'Which method for diagnosing detected anomalies of industrial systems found by answering Research question Q1 is most fitting according to the criteria of Research question Q2?' is answered. While both methods appear to be promising, a method using DES diagnostic models fits discrete-event systems best. While Bayesian networks are more expressive in general, the DES diagnostic models have a smaller required user effort since they can be more directly be based on the machine control application specification.

The classical DES diagnostic method is a formal method based on a complete system model,

which results in problems with scalability. We propose a more practical approach in Chapter 5, which uses the classical DES diagnostic method concepts. The proposed method answers Research question Q4: 'How can the method found by answering Research question Q3 be refined to use machine control application specifications and data from systems in the field?'. The proposed two-step method first detects failures and in the second step, diagnoses the failures. The method uses machine control application specifications to create passive monitors and active diagnosers. Based on the system's data, passive monitors detect failures in the first phase, while active diagnosers diagnose the failure in the second step.

Chapter 5 also answers Research question Q5: 'Which diagnostic information of the abnormal behaviour should a method present to system developers?'. The proposed method diagnoses failures. Such a diagnosis results in a set of faults that could have caused the failure. Information such as the controller's related variables to the failure and the sequence of events between the fault and failure are proposed as additional information. The faults and failure are shown to a system engineer through the dashboard that is also used to control the system.

Finally, the paper describes a prototype based on the proposed method in Chapter 6. The prototype helps to answer Research question Q6: 'Does the refined method of anomaly diagnosis found by answering Research question Q4 satisfy the criteria identified in Research question Q2?'. The prototype can detect and diagnose a simple failure that is introduced by injecting a fault in a controller. The prototype shows that the proposed method satisfies most of the criteria of Research question Q2. The usability was checked with machine control application from Cordis Automation, who have end-user knowledge. It is future work to the usability with end-users. Similarly, the scalability should be checked against a real system instead of a simulator.

The main research of the paper can be answered with the answers found on the research questions. '**How can detected anomalies in systems controlled by machine control applications be diagnosed in a generic and knowledge-driven way?**' *Detected anomalies can be diagnosed using a pragmatic approach based on discrete-event system diagnostics. The pragmatic approach can generically be applied to manually made MCA-controlled systems and uses domain knowledge in the form of a machine control application specification as input. The method is able to detect failures and diagnose the root causes of these failures*.

Main contributions The main contributions of this thesis are:

- Providing an overview of diagnostic methods that are primarily based on domain knowledge.
- Explaining discrete-event system diagnostics, focusing on the online diagnosis of systems controlled by manually made machine control applications.
- Demonstrating how Bayesian networks and discrete-event systems can be applied to failure diagnosis of discrete-event systems.
- Providing a structured comparison between Bayesian networks and discrete-event system diagnostics based on their practical usage.
- Presenting a pragmatic method based on the formal discrete-event system diagnostics.
- Describing and implementing a prototype of the pragmatic discrete-event system diagnostic method.

7.2 Discussion

This section discusses the thesis results and proposes future work. The main result of the thesis is the proposed method. The method uses a two-step approach to failure detection and diagnosis,

which seems promising. The approach is flexible because any specified failure in a system can be diagnosed. Instead of the more formal DES approach of Sampath *et al.* [33], it is up to the creator of the models to determine which failures should be detected. One could start with one component and create passive monitors and active diagnosers. Based on observation on the real machine, failures and the corresponding monitors and diagnosers can be added. Thanks to the modularity of the approach, once a component has monitors and diagnosers, these can be used on any system containing the components.

The two-step approach also gives flexibility between passively monitoring and actively diagnosing a system. Based on the specific requirements, one could focus on in-depth monitoring using passive diagnosers. In practice, extensive passive monitoring could help diagnose failures without starting an active diagnoser. Failures that only have a small set of faults as a potential root cause could be resolved without requiring an active diagnoser. Passive diagnosers, for example, are also able to differentiate between faults.

On the other hand, the prototype supports failures with multiple root causes and failures which result in a system stop. In this case, the system no longer can be passively diagnosed. Active diagnosis can reduce the set of faults. For most of the diagnosis scenarios, both passive monitoring and active diagnosis are viable options. We think that system engineers should make a trade-off between the passive monitoring and active diagnosis based on the cost of stopping the system and the cost of giving commands to the system, compared to increased diagnosis speed of using an active diagnoser.

One explicit limitation of the proposed method is that it can no longer determine a system's diagnosability. While the active diagnosers are based on a system model, they do not provide a complete diagnoser automaton that can be used to determine diagnosability. The active diagnosers only model paths in the system model that provide information on a failure. If the determination of diagnosability of a system is required, a complete (modular) system model should be created including faults. Automatic generation of the diagnoser from that system model and F-indeterminate cycle checking in the diagnoser as described in Section 3.3.6 can help determine diagnosability of a system.

7.3 Future work

In this section, we will describe the method's limitations and propose ways to improve the proposed method.

Application to a real example The proposed method and the created prototype were only applied to an example system. The MerryGoRound does not have all properties of a real system, such as the size. The proposed method's scalability and usability should be evaluated against a real system.

Likewise, in the current example scenario, it is always possible to stop the system. In practice, such a hard stop of the system might not be beneficial for the system's health and might not even be possible. Hence, when extending the prototype for a real system, it should be determined how active diagnosers, or a step between passive monitoring and active diagnosis, could use safe aborts. A starting point could be the idea of marked locations used in supervisory control synthesis [42]. These system models describe safe states that are always reachable. Going to such a state before diagnosis should always be possible. The right safe location should be found that enables active diagnosis.

Relaxing assumptions The proposed method assumes that faults are caused by components that are broken. There are more types of failure in practice, such as drift-like faults, where a component's behaviour degrades over time and intermittent faults that cause failures to be

happening on an irregular interval. Further research should evaluate whether these faults can also be detected and diagnosed by the proposed method.

Furthermore, our example assumes that the system only has a single fault. In the example, only the cylinder or a sensor is broken. In practice, one fault might also cause other faults. It should be researched what the impact is on active diagnosers when the single fault assumption is relaxed.

The prototype currently uses a simple passive monitor that starts an active diagnoser with a failure. While we do not assume that the interface between passive monitors and active diagnosers only consists of giving failures, the prototype example used this simple interface. It should be researched what the implications are of also giving a set of faults that should be reduced to the active diagnoser. For example, a passive diagnoser could already have reduced the potential faults set from the total fault set of the system.

Furthermore, we assume that active diagnosers are directed acyclic graphs. There could be cases where repetition is required to detect a failure. This required repetition would directly result in a non-diagnosable system based on the definition of diagnosability in DES diagnostics. Therefore we think that in principle cycles are never required. Furthermore, for a finite number of loops, a diagnoser model could also describe the required behaviour several times instead of describing a single loop.

Improvements to active diagnosers While the structure and principles of active diagnosers were described in this thesis, the creation of the active diagnosers is a manual process based on (traces of) a system model. We have some ideas on how the creation of active diagnosers could be improved. First, there is a relation between active diagnosis and model-based testing. Model-based testing also uses a model to test system properties [62]. In cases of non-determinism, such testing frameworks also need to take into account in which states the system could be [63]. It should be researched whether the generation of active diagnoser can use principles of generating test traces. This could reduce the manual effort of creating an active diagnoser by determining the traces that exclude potential faults.

Like Chanthery and Pencolé [47] propose, the active diagnosis generation can also be approached as a planning problem. They generate an optimum active diagnoser based on the system model and a set of criteria. A planning algorithm could determine paths through the system model and represent these paths as an active diagnoser defined in this paper. Their planning algorithm assumes that there are multiple ways to implement the system function. We noticed that this might not always be the case in autonomous systems. In our case study on the MerryGoRound, we noticed that this only has one sequence of events to implement the system function. It should be researched whether these planning algorithms also apply to discrete-event systems in general.

Finally, one of the disadvantages of the proposed method compared to Bayesian networks is that the proposed method only acts based upon sensor values observed by the controller. Bayesian networks can also use human, external observations such as a visual inspection of system parts. It might be interesting to research the advantages to allow human input during the execution of an active diagnoser.

A Applying rule-based decision trees

A.1 Decision trees

The experiment with decision trees is based on the rule-based approach of Abdelhalim *et al.* [29]. The rationale behind this decision is experts' ability to look at the rules before they are generated into a decision tree. This makes it the least data-driven approach found for the generation of decision trees. A rule-set is used to determine if a semi-automatic luggage scanner has a failure. If a tray is present (unobservable for the system) and the sensor is high, the personnel should accept the tray to start the conveyor belt.

Figure A.1 shows the set of rules. The first line explains each column. Each subsequent lines represent one rule. Each rule expresses, based on observations (the first four columns), if a failure occurred and if so, which failure (last column). The observations in the table below are the presence of a TRAY, the SENSOR_SIGNAL, the state of the BELT and the action of the PERSONNEL. The failures include a broken conveyor, a broken tray-presence sensor and a Human Machine Interface (HMI) error. DONTCARE means that the observation is not relevant to determine the fault.

The following observations can be made about the rules in Figure A.1. It shows that if there is an inconsistency between the tray being present and the sensor detecting the tray, the sensor is most likely broken (second and third rule). If a tray is present and accepted, but the belt does not run, the conveyor is most likely broken. Finally, if a tray is present, but never accepted, the personnel did not accept it or was never notified, hence the HMI error.

The resulting decision tree generated by the approach is shown in Figure A.2. It shows that the observations are now decisions to determine the most probable fault. The order of the decisions is based on the information each decision gives [29]. In total, four criteria are used. The criteria are applied in order and only if a tie comes out of a criterion, the next criterion is applied. The first criterion, the Attribute effectiveness, tries to find the rule that has the least DONTCARE values. In Figure A.2, the sensor signal and tray presence are the first two decision because they do not have any DONTCARE values. For the entire sequence of criteria, we refer to Abdelhalim *et al.* [29].

```
tray,   sensor_signal, belt_running, personnel, Failure?
present high           no            accepted   conveyor_broken
present low            dontcare      dontcare   sensor_broken
absent  high           dontcare      dontcare   sensor_broken
present high           no            waiting    HMI_error
present high           yes           accepted   no_error
absent  low            no            waiting    no_error
```

Figure A.1: List of rules as source for the decision tree

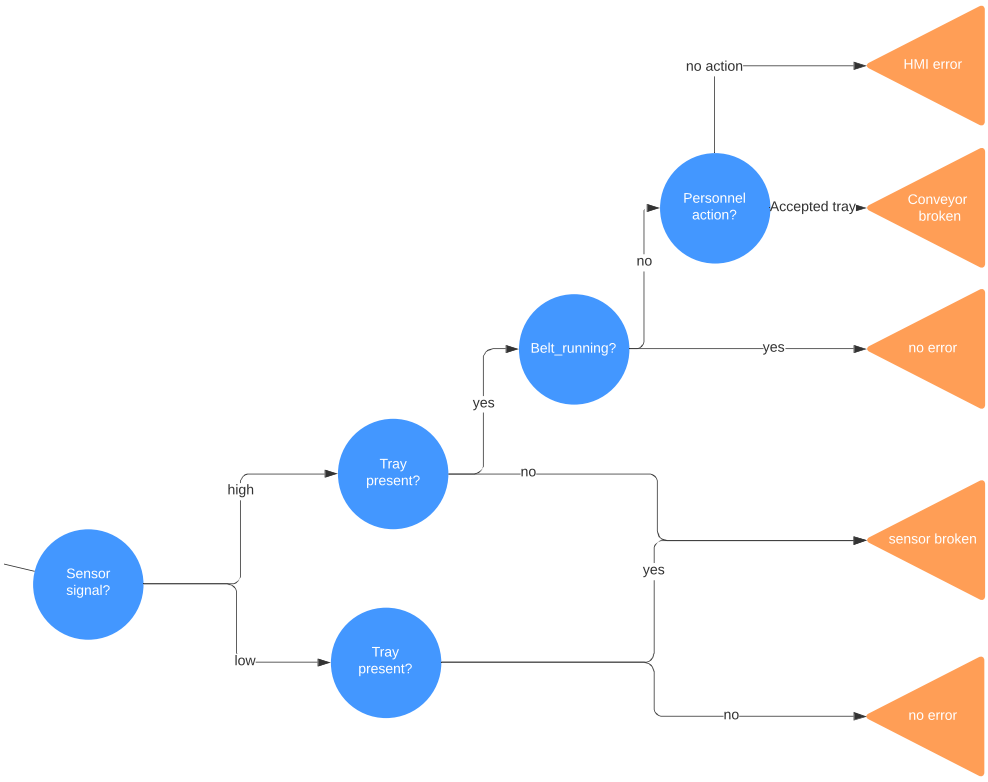


Figure A.2: Decision tree created from rules

Bibliography

[1] A. Detzner and M. Eigner, 'A digital twin for root cause analysis and product quality monitoring', in *DS 92: Proceedings of the DESIGN 2018 15th International Design Conference*, 2018, pp. 1547–1558.

[2] L. Barbini, C. Bratosin and E. van Gerwen, 'Model based diagnosis in complex industrial systems: A methodology', in *PHM Society European Conference*, vol. 5, 2020, pp. 8–8.

[3] G. Weidl, A. Madsen and E. Dahlquist, 'Object Oriented Bayesian Networks for Industrial Process Operation', in *In Proc. Workshop on Bayesian modelling, Uncertainty in AI*, 2003.

[4] Machinaide, *About the machinaide project*, 2019. [Online]. Available: `https://www.machinaide.eu/overview` (accessed 25th Sep. 2020).

[5] ICT Group, *Using Cordis Suite to Design Control Logic*, 2017. [Online]. Available: `https://ict.eu/wp-content/uploads/2017/11/Case-Studie-Cordis-Suite-HR-DEF_EN-002.pdf` (accessed 3rd Nov. 2020).

[6] A. Avižienis, J.-C. Laprie, B. Randell and C. Landwehr, 'Basic concepts and taxonomy of dependable and secure computing', *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11–33, 2004. DOI: `10.1109/TDSC.2004.2`.

[7] R. Baheti and H. Gill, 'Cyber-physical systems', *The impact of control technology*, vol. 12, no. 1, pp. 161–166, 2011.

[8] M. Solé, V. Muntés-Mulero, A. I. Rana and G. Estrada, 'Survey on Models and Techniques for Root-Cause Analysis', *CoRR*, vol. abs/1701.08546, 2017. arXiv: `1701.08546`. [Online]. Available: `http://arxiv.org/abs/1701.08546`.

[9] T. D. Nielsen and F. V. Jensen, *Bayesian networks and decision graphs*. Springer Science & Business Media, 2009.

[10] C. Li and S. Mahadevan, 'Efficient approximate inference in Bayesian networks with continuous variables', *Reliability Engineering & System Safety*, vol. 169, pp. 269–280, 2018, ISSN: 0951-8320. DOI: `https://doi.org/10.1016/j.ress.2017.08.017`.

[11] R. Hofmann and V. Tresp, 'Discovering structure in continuous variables using Bayesian networks', in *Advances in neural information processing systems*, 1996, pp. 500–506.

[12] P. J. Lucas and A. Hommersom, 'Modeling the interactions between discrete and continuous causal factors in Bayesian networks', *International Journal of Intelligent Systems*, vol. 30, no. 3, pp. 209–235, 2015.

[13] V. Mihajlovic and M. Petkovic, *Dynamic Bayesian Networks: A State of the Art*, Undefined, ser. CTIT Technical Report Series. Netherlands: University of Twente, Oct. 2001, vol. TR-CTIT-34, Imported from CTIT.

[14] S. Mascaro, A. E. Nicholso and K. B. Korb, 'Anomaly detection in vessel tracks using Bayesian networks', *International Journal of Approximate Reasoning*, vol. 55, no. 1, Part 1, pp. 84–98, 2014, Applications of Bayesian Networks, ISSN: 0888-613X. DOI: `https://doi.org/10.1016/j.ijar.2013.03.012`.

[15] J. de Kleer and J. Kurien, 'Fundamentals of model-based diagnosis', *IFAC Proceedings Volumes*, vol. 36, no. 5, pp. 25–36, 2003, 5th IFAC Symposium on Fault Detection, Supervision and Safety of Technical Processes 2003, Washington DC, 9-11 June 1997, ISSN: 1474-6670. DOI: https://doi.org/10.1016/S1474-6670(17)36467-4.

[16] M. Borth and H. von Hasseln, 'Systematic generation of Bayesian networks from systems specifications', in *Intelligent Information Processing*, ser. IFIP Advances in Information and Communication Technology, vol. 93, Boston, MA, 2002, pp. 155–166, ISBN: 978-1-4757-1031-1.

[17] G. Provan, 'A graphical framework for stochastic model-based diagnosis', in *2016 3rd Conference on Control and Fault-Tolerant Systems (SysTol)*, 2016, pp. 566–571.

[18] B. Cai, L. Huang and M. Xie, 'Bayesian Networks in Fault Diagnosis', *IEEE Transactions on Industrial Informatics*, vol. 13, no. 5, pp. 2227–2240, 2017. DOI: 10.1109/TII.2017.2695583.

[19] G. Weidl, A. Madsen and S. Israelson, 'Applications of object-oriented Bayesian networks for condition monitoring, root cause analysis and decision support on operation of complex continuous processes', *Computers & Chemical Engineering*, vol. 29, no. 9, pp. 1996–2009, 2005, ISSN: 0098-1354. DOI: https://doi.org/10.1016/j.compchemeng.2005.05.005.

[20] L. Perreault, M. Thornton, S. Strasser and J. W. Sheppard, 'Deriving prognostic continuous time Bayesian networks from D-matrices', in *2015 IEEE AUTOTESTCON*, 2015, pp. 152–161.

[21] X. Yu, J. Liu, Z. Yang and X. Liu, 'The Bayesian network based program dependence graph and its application to fault localization', *Journal of Systems and Software*, vol. 134, pp. 44–53, 2017, ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2017.08.025.

[22] P. E. Utgoff, 'Incremental Induction of Decision Trees', *Machine Learning*, vol. 4, no. 2, pp. 161–186, Nov. 1989, ISSN: 0885-6125. DOI: 10.1023/A:1022699900025. [Online]. Available: https://doi.org/10.1023/A:1022699900025.

[23] B. Moret, 'Decision Trees and Diagrams', *ACM Computing Surveys*, vol. 14, pp. 593–623, Dec. 1982. DOI: 10.1145/356893.356898.

[24] S. Wahl and J. Sheppard, 'Extracting Decision Trees from Diagnostic Bayesian Networks to Guide Test Selection', *Annual Conference of the Prognostics and Health Management Society, PHM 2010*, Jan. 2010.

[25] M. Demetgul, 'Fault diagnosis on production systems with support vector machine and decision trees algorithms', *The International Journal of Advanced Manufacturing Technology*, vol. 67, no. 9-12, pp. 2183–2194, 2013.

[26] J. Vieira and C. Antunes, 'Decision tree learner in the presence of domain knowledge', in *Chinese Semantic Web and Web Science Conference*, Springer, 2014, pp. 42–55.

[27] A. Bouza, G. Reif, A. Bernstein and H. Gall, 'Semtree: Ontology-based decision tree algorithm for recommender systems', *CEUR Workshop Proceedings*, vol. 401, 2008, ISSN: 16130073.

[28] M. R. A. Iqbal, S. Rahman, S. I. Nabil and I. U. A. Chowdhury, 'Knowledge based decision tree construction with feature importance domain knowledge', in *2012 7th International Conference on Electrical and Computer Engineering*, 2012, pp. 659–662.

[29] A. Abdelhalim, I. Traore and B. Sayed, 'RBDT-1: A New Rule-Based Decision Tree Generation Technique', in *Rule Interchange and Applications*, ser. Lecture Notes in Computer Science, G. Governatori, J. Hall and A. Paschke, Eds., vol. 5858, Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 108–121, ISBN: 978-3-642-04985-9.

[30] S. Strasser and J. Sheppard, 'An empirical evaluation of Bayesian networks derived from fault trees', in *2013 IEEE Aerospace Conference*, 2013, pp. 1–13.

[31] S. Bayat, M. Cuggia, D. Rossille, M. Kessler and L. Frimat, 'Comparison of Bayesian Network and Decision Tree Methods for Predicting Access to the Renal Transplant Waiting List', *Studies in health technology and informatics*, vol. 150, pp. 600–4, Feb. 2009. DOI: 10.3233/978-1-60750-044-5-600.

[32] J. Sleuters, Y. Li, J. Verriet, M. Velikova and R. Doornbos, 'A Digital Twin Method for Automated Behavior Analysis of Large-Scale Distributed IoT Systems', in *2019 14th Annual Conference System of Systems Engineering (SoSE)*, 2019, pp. 7–12. DOI: 10.1109/SYSOSE.2019.8753845.

[33] M. Sampath, R. Sengupta, S. Lafortune, K. Sinnamohideen and D. C. Teneketzis, 'Failure Diagnosis Using Discrete-Event Models', *IEEE Transactions on Control Systems Technology*, vol. 4, no. 2, pp. 105–124, 1996.

[34] J. Zaytoon and S. Lafortune, 'Overview of fault diagnosis methods for Discrete Event Systems', *Annual Reviews in Control*, vol. 37, no. 2, pp. 308–320, 2013, ISSN: 1367-5788. DOI: https://doi.org/10.1016/j.arcontrol.2013.09.009.

[35] M. Sampath, R. Sengupta, S. Lafortune, K. Sinnamohideen and D. Teneketzis, 'Diagnosability of Discrete-Event Systems', *IEEE Transactions on Automatic Control*, vol. 40, no. 9, pp. 1555–1575, 1995.

[36] M. Roth, J.-J. Lesage and L. Litz, 'The concept of residuals for fault localization in discrete event systems', *Control Engineering Practice*, vol. 19, no. 9, pp. 978–988, 2011, Special Section: DCDS'09 – The 2nd IFAC Workshop on Dependable Control of Discrete Systems, ISSN: 0967-0661. DOI: https://doi.org/10.1016/j.conengprac.2011.02.008.

[37] X. Zhao, G. Lamperti, D. Ouyang and X. Tong, 'Minimal Diagnosis and Diagnosability of Discrete-Event Systems Modeled by Automata', *Complexity*, vol. 2020, pp. 1–17, Feb. 2020. DOI: 10.1155/2020/4306261.

[38] M. V. Moreira and J.-J. Lesage, 'Fault diagnosis based on identified discrete-event models', *Control Engineering Practice*, vol. 91, p. 104 101, 2019, ISSN: 0967-0661. DOI: https://doi.org/10.1016/j.conengprac.2019.07.019.

[39] S. Jansen, 'A Modular Approach Towards the Runtime Verification of Machine Control Applications', Master's thesis, Radboud University, 2020. [Online]. Available: https://www.ru.nl/publish/pages/769526/sam_jansen.pdf.

[40] F. F. H. Reijnen, M. A. Goorden, J. M. van de Mortel-Fronczak and J. E. Rooda, 'Supervisory control synthesis for a waterway lock', in *2017 IEEE Conference on Control Technology and Applications (CCTA)*, 2017, pp. 1562–1563. DOI: 10.1109/CCTA.2017.8062679.

[41] F. Reijnen, A. van de Mortel-Fronczak, K. Rooda and J. van Dinther, 'Synthesis and implementation of supervisory control for infrastructural systems', in *Proceedings of the 38th Benelux Meeting on Systems and Control*, 2019, p. 92. [Online]. Available: https://research.tue.nl/nl/publications/synthesis-and-implementation-of-supervisory-control-for-infrastru.

[42] F. Reijnen, M. Reniers, J. van de Mortel-Fronczak and J. Rooda, 'Structured Synthesis of Fault-Tolerant Supervisory Controllers', *IFAC-PapersOnLine*, vol. 51, no. 24, pp. 894–901, 2018, ISSN: 2405-8963. DOI: https://doi.org/10.1016/j.ifacol.2018.09.681.

[43] M. Goorden, J. van de Mortel-Fronczak, M. Reniers and J. Rooda, 'Structuring multilevel discrete-event systems with dependency structure matrices', in *2017 IEEE 56th Annual Conference on Decision and Control (CDC)*, 2017, pp. 558–564. DOI: 10.1109/CDC.2017.8263721.

[44] R. Debouk, R. Malik and B. Brandin, 'A modular architecture for diagnosis of discrete event systems', in *Proceedings of the 41st IEEE Conference on Decision and Control*, vol. 1, 2002, pp. 417–422. DOI: 10.1109/CDC.2002.1184530.

[45] F. G. Cabral, M. V. Moreira and O. Diene, 'Online fault diagnosis of modular discrete-event systems', in *2015 54th IEEE Conference on Decision and Control (CDC)*, 2015, pp. 4450–4455. DOI: 10.1109/CDC.2015.7402914.

[46] M. Sampath, S. Lafortune and D. Teneketzis, 'Active diagnosis of discrete-event systems', *IEEE Transactions on Automatic Control*, vol. 43, no. 7, pp. 908–929, 1998. DOI: 10.1109/9.701089.

[47] E. Chanthery and Y. Pencolé, 'Monitoring and Active Diagnosis for Discrete-Event Systems', *IFAC Proceedings Volumes*, vol. 42, no. 8, pp. 1545–1550, 2009, 7th IFAC Symposium on Fault Detection, Supervision and Safety of Technical Processes, ISSN: 1474-6670. DOI: https://doi.org/10.3182/20090630-4-ES-2003.00252.

[48] S. G. Krantz and S. G. Krantz, *Handbook of complex variables*. Springer Science & Business Media, 1999.

[49] C. G. Cassandras and S. Lafortune, *Introduction to Discrete Event Systems*, 2nd. Springer Publishing Company, Incorporated, 2010, ISBN: 1441941193.

[50] F. Yan, H. Zhou, Z. Peng, M. Chen, Y. Xiong and Z. Ren, 'An Industrial Environmental Security Monitoring System in Mobile Device Based on Soft PLC', in *2020 International Conference on Intelligent Transportation, Big Data Smart City (ICITBS)*, 2020, pp. 711–715. DOI: 10.1109/ICITBS49701.2020.00157.

[51] S. Ould Biha, 'A Formal Semantics of PLC Programs in Coq', in *2011 IEEE 35th Annual Computer Software and Applications Conference*, 2011, pp. 118–127. DOI: 10.1109/COMPSAC.2011.23.

[52] Cordis Automation, *Cordis Modeler Language Reference*, 2019.

[53] P. J. G. Ramadge and W. M. Wonham, 'The control of discrete event systems', *Proceedings of the IEEE*, vol. 77, no. 1, pp. 81–98, 1989. DOI: 10.1109/5.21072.

[54] S. Lafortune and E. Chen, 'A Relational Algebraic Approach to the Representation and Analysis of Discrete Event Systems', in *1991 American Control Conference*, 1991, pp. 2893–2898. DOI: 10.23919/ACC.1991.4791933.

[55] V. Venkatasubramanian, R. Rengaswamy, K. Yin and S. N. Kavuri, 'A review of process fault detection and diagnosis: Part I: Quantitative model-based methods', *Computers & Chemical Engineering*, vol. 27, no. 3, pp. 293–311, 2003, ISSN: 0098-1354. DOI: https://doi.org/10.1016/S0098-1354(02)00160-6.

[56] L. Ricker, S. Lafortune and S. Genc, 'DESUMA: A Tool Integrating GIDDES and UMDES', in *Proc. 8th International Workshop on Discrete Event Systems Workshop*, 2006, pp. 392–393. DOI: 10.1109/WODES.2006.382402.

[57] Bayes Server, 'Bayesian network & causal modeling software for Artificial Intelligence', *Bayesserver.com*, [Online]. Available: https://www.bayesserver.com/ (accessed 2nd Dec. 2020).

[58] Norsys Software Corp., 'Netica Application', *Norsys.com*, [Online]. Available: https://www.norsys.com/netica.html (accessed 2nd Dec. 2020).

[59] Altova, 'UModel UML Modeling Tool', *altova.com*, [Online]. Available: https://www.altova.com/umodel (accessed 2nd Dec. 2020).

[60] W. M. Wonham, K. Cai *et al.*, *Supervisory control of discrete-event systems*. Springer, 2019.

[61] D. E. Knuth, 'Backus Normal Form vs. Backus Naur Form', *Communications of the ACM*, vol. 7, no. 12, pp. 735–736, 1964.

[62] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton and B. M. Horowitz, 'Model-based testing in practice', in *Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No.99CB37002)*, 1999, pp. 285–294. DOI: 10.1145/302405.302640.

[63] J. Tretmans, 'Model based testing with labelled transition systems', in *Formal methods and testing*, Springer, 2008, pp. 1–38.