TNO innovation for life

Guided diagnosis of functional failures in cyber-physical systems

SD2Act 2023

TNO Public ⟩ TNO 2024 R10833
April 30, 2024

ICT, Strategy & Policy
www.tno.nl
+31 88 866 50 00
info@tno.nl

TNO 2024 R10833 – April 30, 2024

SD2Act 2023

Guided diagnosis of functional failures in cyber-physical systems

Author(s)	T.C. (Thomas) Nägele, L. (Leonardo) Barbini, R. (Robert) Passmann, A. (Alvaro) Piedrafita Postigo

Classification report	TNO Public
Title	TNO Public
Report text	TNO Public
Number of pages	66 (excl. front and back cover)
Number of appendices	0
Sponsor	ASML
Project name	SD2Act 2023
Project number	060.55515/01.01

Contents

1 Introduction

1.1 Project goals and scope

The System Design to Service Actions (SD2Act) project is a cooperation between TNO-ESI and ASML, world leader in developing and manufacturing high-tech lithographic systems for the semiconductor industry.

The generic goal of the SD2Act project is to develop a methodology that minimizes the time to diagnose failures occurring in an ASML-like system.

The methodology should be generically applicable to high-tech systems also from manufacturers other than ASML.

The business driver for this project is that as high-tech systems become more and more capital intensive there is a strong drive to have no unscheduled down-time achieved through predictive maintenance. On the long road towards predictive maintenance lie several milestones, one of which is to keep the amount of unexpected down time of high-tech systems to a minimum for a fixed number of failures. A large contributor to down-time is the amount of time needed by the complex diagnostic process. This is the process that infers the root cause of a failure from its symptoms. Nowadays this diagnostic inference process highly relies on human reasoning, i.e. system's experts solve diagnostic cases one by one.

The specific goal of the SD2Act project is to aid and (when possible) replace the human diagnostic reasoning with a form of computer reasoning based on design information.

The approach used in this project is to develop such a reasoning system making as much as possible use of the knowledge on the high-tech system's design (this is described by the edge *create/generate model* in Figure 1.1)

For high-tech systems there are both many classes of causes that can lead to a system failure and many classes of symptoms that could results from these causes. Examples of classes of causes are: hardware malfunctions, software bugs, operator errors, abnormal environmental conditions. Examples of classes of symptoms are a system which: stops production, fails to execute a desired operational function, produces outputs with a quality lower than desired, requires too much energy to produce the desired outputs. Notice that a diagnostic problem can be the result of a many-to-many connection among elements of these two different classes of causes and symptoms.

In the scope of the SD2Act project, failures are caused by hardware malfunctions which manifest by a system failing to execute one of its operational functions.

Ideally, industry needs a generic diagnostic methodology that infers causes given symptoms, independently of their membership classes. Research on other aspects of this methodology is described below.

1.2 Project position in the diagnostic landscape

In this section we position the SD2Act project with respect to other projects done by TNO-ESI in the diagnostic domain. We give a list of commercially available off-the-shelf tools for diagnostics . Finally we briefly describe its position with respect to academic research.

1.2.1 Relation to TNO-ESI diagnostic projects

TNO-ESI had conducted in the past years other projects with high-tech industrial partners in the domain of diagnostics. Specifically, the CareFree project with Canon Production Printing and the Assisted Diagnostic in Action (ADiA) project with ASML.

The CareFree and ADiA project have a diagnostic scope different than the one of the SD2Act project:

- the scope of the CareFree project is on diagnosing down-time situations caused by HW failures that manifest as a system which stops production (rather than as a system that does not perform a function). More details on this project can be found in the TNO report (van Gerwen, 2024).
- the ADiA project also focuses on HW failures, however the scope in ADiA differs as it assesses the diagnosability level of a system, i.e. the scope is shifted from diagnostics at operational time to diagnostics at design time. More details on this project can be found in (Barbini et al., 2021).

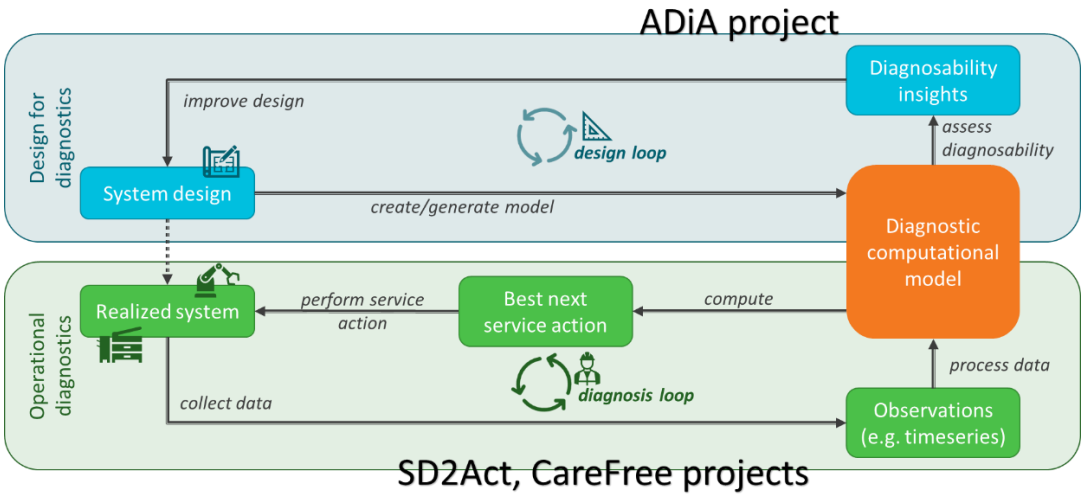


Figure 1.1 Overview of TNO-ESI diagnostics projects.

An overview of these projects and their position with respect to the design and operational diagnostics scopes is given in Figure 1.1.

Despite the different scopes, the approach adopted in the different projects is overlapping. All the projects implement a model-based approach, with diagnostic computational models created/generated using knowledge of the system design. Furthermore, as shown in Figure 1.1, in the proposed approach the same diagnostic model is used both at design time and operational time. This implies that a unification of the approaches, yielding the overall diagnostic methodology needed by industry, even if not yet achieved, seems reasonably possible. An overview of this unified diagnostic methodology and its connection to model-based system engineering approaches is detailed in (van Gerwen et al., 2022).

1.2.2 Relation to off-the-shelf tooling

A non-exhaustive list of off-the-shelf tools that, during the SD2Act project, have been investigated or have been identified as potentially interesting is given in Table 1.1.

Table 1.1: Non-exhaustive overview of off-the-shelf diagnostic tools.

Name	Website	Reference
MADe	www.phmtechnology.com	(Hess et al., 2008)
TEAMS	www.teamqsi.com	(Deb et al., 1995)
Kairos Workbench	www.kairostech.no	(Lind, 2011)
RODON	www.combitech.com	(Adén & Stjernström, 2013)

The criteria for a tool to be considered were that it must offer a scalable modelling approach, be already successfully applied in an industrial context, and offer, to some degree, both design and operational diagnostics support.

Furthermore, the selected tools all implement a model-based diagnostic approach, as in the SD2Act project. Several tools are nowadays being introduced in the diagnostic landscape that use a more data-driven approach, these have not been investigated in the SD2Act project. It is suggested that such tools are investigated in a follow-up project.

The tool that is identified as most promising and as a consequence was in-depth investigated is MADe. The investigation was carried out by modelling example systems inspired by the ASML system. The conclusion of the investigation is that the MADe tool does not yet offer enough diagnostic support to be adopted by ASML. Specifically, high-tech systems like the ASML one:

- are cyber-physical systems with intertwined HW and SW. However, the MADe tool does not offer a suitable way to model SW and its role in the propagation of failures from component to system-level observables;
- span several HW physical domains such as optics, thermal, mechanical and electromagnetic. All of these should be equally modelled in order to have an effective diagnosis. However, the MADe tool is focusing on a subset of these physical domains;
- have a high level of complexity and diagnostic support is needed at operational time. However, the main goal of MADe is design for diagnostics and hence it offers limited operational diagnostic support.

Based on these considerations, the conclusion is that the SD2Act and similar research projects are very much needed by the high-tech sector to solve their diagnostic challenges.

1.2.3 Position with respect to academic research

The SD2Act project provides a clearer picture of the gaps between academic research and the needs of industry. Yet, the SD2Act does not deliver a full state-of-the-art report on diagnostic methods for high-tech systems. We suggest this as an activity for a follow-up project.

The SD2Act project found that academic research in diagnostics is mostly focusing on component level in the HW domain, with detailed model-based (physics), data-driven, or hybrid approaches developed to infer the state of a single or a small set of HW components. These academic approaches usually deal with systems in the order of tens of components. However, high-tech industrial systems have in the order of thousands of HW components belonging to different physical domains and, as stated above, intertwined with control and safety SW. This extension of diagnostic methodologies from component-level to system-level, defines a gap between the current academic state of the art and the industrial needs.

The issue is not merely one of scalability, understood as the ability to systematically build models for large systems and have computation on those models take a reasonable amount of time. But more fundamentally, the issue lies in the emergence of behaviours at system level which are not derivable from modelling the interactions of single components, i.e. integration with system properties is needed. Academic research on this last aspect is needed.

The SD2Act project attempts at filling this gap identified in academic research by focusing on methods to diagnose functional failures, i.e. failures that manifest by a system failing to execute one of its operational functions.

2 Methodology

2.1 Overview

Whereas the ultimate goal of the SD2Act methodology is to both improve the diagnosability of a system during its design phase (design for diagnostics) as well as to assist service engineers to come to a diagnosis when the system is deployed (operational diagnostics), the focus in SD2Act is on the latter. In this light, a diagnostic model is created on the basis of readily available system design information. This information is incorporated in a diagnostic assistant that is made available for the service engineer.

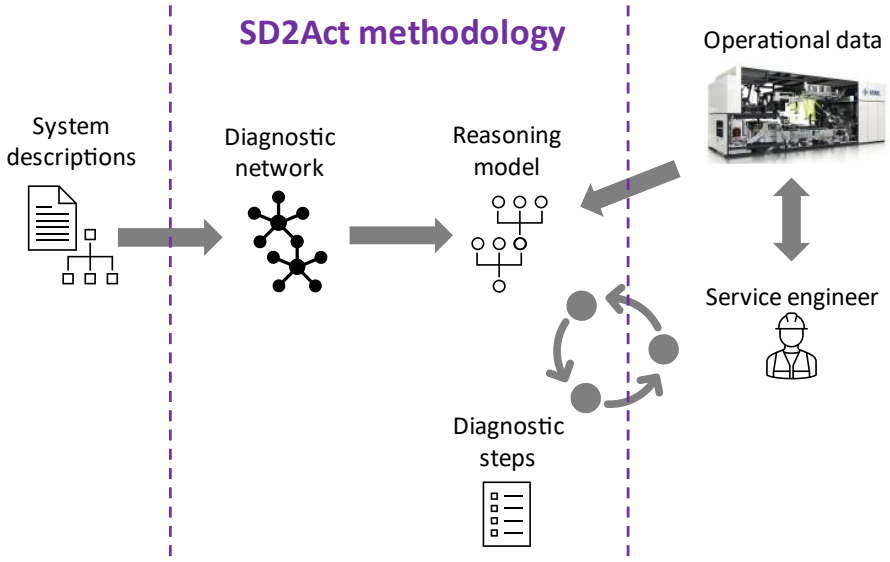


Figure 2.1: High-level overview of the SD2Act methodology.

The high-level overview of the SD2Act methodology workflow is depicted in Figure 2.1. On the left-hand side the design information is shown, which corresponds to the system design in Figure 1.1. On the right-hand side the operational diagnosis loop is shown, which is equivalent to the loop in the bottom of Figure 1.1.

The SD2Act methodology follows a top-down approach to enable system-level diagnostics. In order to deal with the complexity of high-tech systems, the system is usually broken down into subsystems, which are again broken down into modules, and so forth. Finally, there are the individual hardware pieces and software components realizing the small subtasks within the whole system. The breakdown of the system represents a separation of concerns, where every element is designed to fulfil its own purpose, and the collection of them all makes the system function as designed.

All levels in the system breakdown have their own concern: to fulfil their designed function(s). Moving to the diagnostic domain, a starting point for conducting a root cause analysis is usually something the system does incorrectly or not at all: a high-level function is not performed as expected. The root cause analysis consists of a drill-down from higher level observations to the root cause by focussing on the relevant pieces of the system, based on a set of observations, by inspecting the system step by step. This inspection is often done

on a functional level: 'Does function X still work?' Depending on the answer, the scope of the root cause analysis is narrowed.

The SD2Act project aims for this kind of functional drill-down, as it allows for system-level root-cause analysis and it is a rather natural way of doing *exclusion diagnostics*. To support this, we develop a methodology for the creation of functional diagnostic models, which captures system functions, their dependencies and their hierarchy.

To allow for this, all of these dependencies are captured in a diagnostic network. This network represents the knowledge about the system while abstracting away from the reasoning formalism being used to do the diagnostic reasoning. This reasoning model is created based on the information in the network, after which it can be used for diagnosis by inserting findings from the system and service engineer into the reasoning model. Step by step, the reasoning model is capable of suggesting the best next steps to the service engineer.

2.2 Methodology implementation

To be able to experiment with the proposed methodology and to perform validation, a proof-of-concept implementation was made in Python. This proof-of-concept Python library is named *MBDIyb* and allows to experiment at scale, by easily specifying large networks. The library also formalizes and consolidates the research done in the project, and is actively being worked on through the course of the SD2Act project series. The high-level overview of the SD2Act methodology as implemented in *MBDIyb* is shown in Figure 2.2. See also Figure 2.1 for comparison.

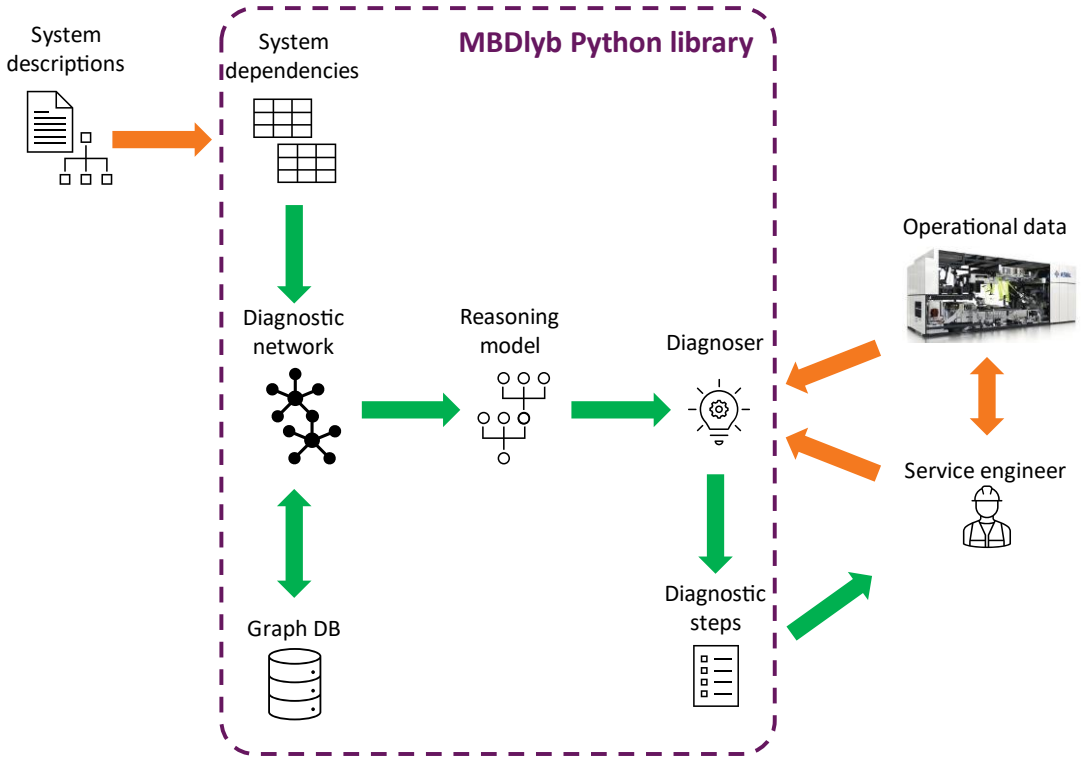


Figure 2.2: High-level overview of the SD2Act methodology's workflow as it is implemented in *MBDIyb*. Orange edges (currently) require human involvement while green edges represent fully automated transformations or computations.

The implementation choices made in *MBDIyb* result in additional transformation steps compared to the generic method depicted in Figure 2.1. The system dependencies form the input to the diagnostic network, and the diagnoser is added as an interface between the reasoning model and the service engineer or system data. Also, a storage in the form of a graph database is added. Each of the different stages in the *MBDIyb* workflow is briefly described below.

Stages in the workflow

The conceptual meaning of each of the six stages shown in Figure 2.2 is briefly described below. The following sections elaborate more on the details of each of the stages.

- **System descriptions**
  The system descriptions represents all system design information, both structured and unstructured, that is relevant for the creation of the diagnostic model.
- **System dependencies**
  This stage represents the manual specification of the (functional) dependencies that should be in the diagnostic model. This is essentially the entry point of the methodology that converts design information to a format for which a semi-formal interpretation is defined.
- **Diagnostic network**
  The diagnostic network represents all the knowledge that was provided for the creation of the functional diagnostic model. The semantics of the internal format are defined so that it allows being transformed into a formal reasoning model.
- **Graph DB**
  The diagnostic network itself is instantiated by the library and only lives in memory for the time the MBDIyb application is running. A graph database provides a persistent storage space in which all knowledge of the diagnostic can be stored, so that the diagnostic model can be loaded at a later moment. Currently, Neo4j[1] is the graph database use by *MBDIyb* to store the information in.
- **Reasoning model**
  The reasoning model is the model that has a formal (mathematical) semantics that allows for diagnostic reasoning.
- **Diagnoser**
  The diagnoser takes in findings from the system being diagnosed and uses the reasoning model to compute a diagnosis. It then suggests diagnostic tests to do and again takes in the findings. These steps are repeated until a final diagnosis is reached. Hereby the diagnoser implements operational diagnostics as depicted in the bottom of Figure 1.1.

The remainder of this report considers the implemented methodology as the main methodology rather than the conceptual one presented in Figure 2.1, as most of the discussed features have been implemented in the proof-of-concept methodology implementation.

[1] https://neo4j.com/

2.3 Modelling

Design information

The design information needed to create the functional diagnostic model mainly consists of high-level system design information, such as functional breakdowns and system decompositions.

In addition to those types of design information, also more structured system architectural models could be used as input. Capella (Roques, 2017) is a modelling workbench supporting the Arcadia method (Voirin, 2017). This systems engineering approach supports the system design by going systematically from the user's needs to functions the system needs to perform. Multiple abstraction levels help in translating those system functions into subsystems with their own subfunctions, including the relations between these subsystems. Eventually, a connection to the concrete hardware pieces that need to realize such functions is made. This type of structured breakdown in the Capella model provides a good starting point for the creation of the functional diagnostic model. Being a model-based approach as well, the Capella model could in the future also allow for automatic model-to-model transformation.

To allow the diagnostic model to estimate the health of individual hardware pieces, or groups thereof, these hardware pieces and their mapping to the system functions they realize must be known. For this, a material or part list could be used to add the components to the model. The functional deployment of the system functions onto those hardware pieces follows either from the breakdown of the part list or other types of design documents or models. The mapping of system functions onto hardware pieces can also be provided by a Capella model, as shown in Figure 2.3.

Figure 2.3: Functional deployment onto hardware pieces as specified in a Capella model. The function 'ConvertACtoDC' is realized by the hardware piece 'Converter'.

Systems log a lot of data during operation, e.g., timeseries containing sensor data, software events and errors. Some of these observations provide direct health indicators for hardware pieces or report on the fulfilment of system functions. For example, software may report an error when it is unable to fulfil its requested task. If this task represents a system function that software is controlling, the reporting of this error is a direct indictor for a functional failure. These observables are all made automatically by the system and can often be inserted immediately into the diagnostic model, already eliminating many potential root causes from the diagnosis. Documentation where such information can be found are event/error lists of software components or Key Performance Indicator (KPI) lists. It is not always trivial how to map an observable to a function or hardware piece, so an expert in the loop may sometimes be needed to manually do this at the time of model construction.

Besides the observations made by the system itself, there is a set of diagnostic tests that can be done to acquire additional information about the state of the system. Such diagnostic tests are typically documented in a service manual. The diagnostic tests should be linked to the specific functions or hardware pieces they provide information for regarding its health status, i.e., whether the function is performed (correctly) or whether the hardware piece is potentially unhealthy.

The types of information required to build a diagnostic model according to the presented methodology include the following:

- Functional breakdown
- Replaceable part/hardware lists and breakdowns
- Functional deployment information onto the hardware
- KPI/observability lists
- (Diagnostic) tests and calibration lists

Functional dependencies

All relations are expressed by means of different types of functional dependencies. If function A requires function B to be performed, a functional dependency from B to A is specified. This specification can be done via a dependency matrix, in which all functions are listed both in the rows as well as in the columns. If the function in the column depends on the function in the row, the corresponding cell can be marked, as shown in Table 2.1.

Table 2.1: An example dependency matrix.

	Function_A	Function_B	Function_C	Function_D
Function_A		X	X	
Function_B				X
Function_C				X
Function_D				

In this table, Function_B and Function_C both depend on Function_A, and Function_D depends on Function_B and Function_C. A similar matrix can be made to also express the dependencies of functions on hardware pieces, or to express the dependencies of observables and tests to functions or hardware pieces. These dependency matrices represent the functional dependencies that form the input to the methodology workflow.

2.4 Diagnostic network

From the functional dependency matrices, a diagnostic network is created. The diagnostic network is essentially a knowledge graph that forms the basis for the transformation to the reasoning model. This knowledge graph is automatically constructed from dependency tables, which are the result of a (currently) manual formalization step to capture the relevant design information. The dependencies from the dependency table are transformed into types of nodes and relations, representing different types of dependency relations. An example network is depicted in Figure 2.4. Different colours represent different types of nodes and the edges represent relations, for which the types are labelled.

Details regarding the semantics of all types and the validity of certain relations are described in Section 2.4.1. The diagnostic network also provides potential for discovery of new functional dependencies that have not been documented explicitly by combining the documented knowledge with basic physics knowledge. This extension is described in Section 2.4.2. An Excel-based approach to specify a basic diagnostic network is explained in Section 2.4.3.

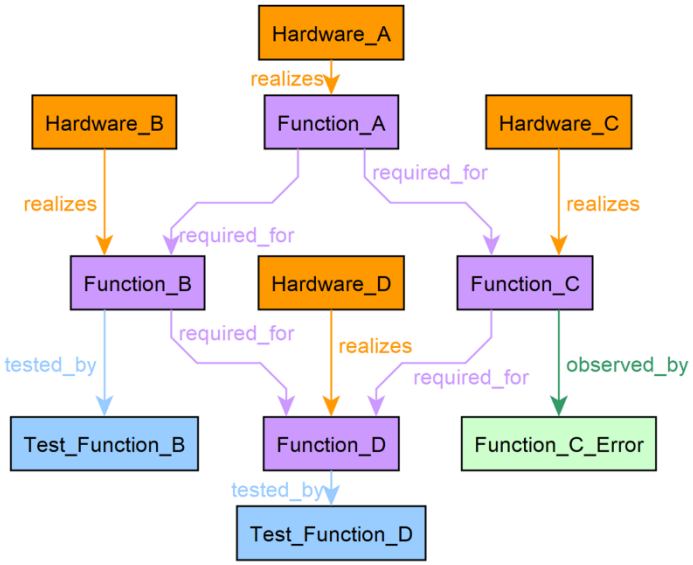


Figure 2.4: An example diagnostic network consisting of four different functions, four pieces of hardware, one system observables and two diagnostic tests. The functional dependencies between the functions (purple) correspond to the functional dependencies shown in Table 2.1.

2.4.1 Building blocks

The diagnostic network is a directed knowledge graph that consists of four types of nodes and six types of relations. The representation of each node and relation type and its semantics are explained below. Figure 2.4 shows an example diagnostic network, also illustrating the colours used to visualize the types. Note that not all types of relations are shown in the picture.

Types of nodes

Hardware node (orange)

A hardware node represents the health of one or more hardware components in the system. At a high level in the system hierarchy, e.g., at subsystem-level, a hardware node represents the health of the collection of hardware pieces within that subsystem.

A hardware node has at least 1 state valuation: *Healthy*, which represents that there is no problem with the hardware piece(s). Upon specification of a hardware node, one may specify the possible hardware faults and their prior probabilities to add more state valuations to the node. This allows for example for the specification of a hardware part that is 95% likely to be healthy, 4% likely to be contaminated and 1% likely to be malformed. These included faults and their likelihoods will determine the hardware's priors.

Function node (purple)

A function node represents a system function and whether this system function is performed as expected or not. Many system functions break down into lower-level system functions, which are the subfunctions of such system function. For example, on a high level a system may have *Heat up the house* as system function, which usually breaks down into subfunctions like heating up some water, circulating the water, dissipating the heat towards the environment, etc. All of these functions are considered system functions and are expressed by function nodes, though they reflect different levels of the system hierarchy.

Function nodes have two possible state valuations: *Ok* and *NOk*, where the first indicates that the system function is performed according to expectation and the latter indicates it does not. A function node cannot be a root node in the network and therefore does not need any priors to be set, as the function can only be performed if either the realization of the function is *Healthy*, represented by Hardware nodes, or it's realizing subfunctions are *Ok*.

Direct observable node (green)

A direct observable node represents an automatic interpretation of an observation made by the system itself, e.g., an event in the log, a reported reading from a sensor or a KPI. Key is that this system observation must be automatically interpretable, which means that no human intervention is needed to estimate what the node's state should be. An example is an error logged by software, for which its presence in the event log can be checked automatically. An example based on sensor readings is when the reading must be within a certain predefined range, which can also automatically be checked. While using the diagnostic model to diagnose, these nodes determine the initial state of the diagnosis.

Direct observable nodes have two possible state valuations: *Ok* and *NOk*. The first one indicates that the reading of this observable is not suspicious, i.e., it does not indicate a problem. The latter indicates a potential problem. For example, if a node represents the occurrence of an error in the event log its state should be *NOk*.

Diagnostic test (blue)

A diagnostic test node represents a diagnostic test that has to be triggered or executed by a service engineer to acquire additional data from the system. These are manual diagnostic tests, such as measuring the power in a certain place or testing whether a sensor triggers upon a manual action. Additionally, many systems have diagnostic self-tests that need to be started by a service engineer and for which the results need human interpretation. Another example is a parameter being logged automatically, for which also human interpretation is needed to assess whether or not the data is as expected. While using the diagnostic model to diagnose a certain situation, some of the diagnostic tests are executed, providing additional findings on top of the direct observations to come to a diagnosis.

Identical to the direct observables, diagnostic tests have two state valuations: *Ok* and *NOk*, where the first indicates the diagnostic test result is as expected (according to normal operation conditions) and the latter does not.

Types of relations

Different types of relations allow for connecting specific types of nodes to each other. All relations are represented by directed edges from one node to the other. Table 2.2 provides an overview of the types of relations in the diagnostic network, including the valid source and target node types for each of the relations.

Table 2.2: An overview of the types of relations in the diagnostic network.

Name	Label	Color	Source node types	Target node types
Realizes	realizes	Orange	Hardware	Function
Required for	required_for	Purple	Function	Function
Affects	affects	Red	Hardware	Function
Subfunction	subfunction_of	Pink	Function	Function

Name	Label	Color	Source node types	Target node types
Test	tested_by	Blue	Hardware, Function	Diagnostic Test
Observable	observed_by	Green	Hardware, Function	Direct Observable

Realizes (orange)

The realizes relation represents the deployment of a function onto a hardware piece or group of hardware pieces. The relation edge direction is from a hardware node to a function node. The semantics of the relation is such that the function needs its realizing hardware pieces to be *Healthy* in order to function according to expectation, i.e., to be *Ok*. If one of the hardware pieces that realizes the function is not *Healthy*, the function will be *NOk*.

Required for (purple)

The required for relation ('required_for') represents a functional dependency of one function on another. Quite literally, the target function of the relation needs the source function to function properly (to be *Ok*) in order to function itself. This means that if a function has multiple incoming required for relations from other functions, it depends on all of them to be functioning. In logic, this is represented as an AND gate: the function is only *Ok* if all of its required functions are also *Ok*, otherwise it is *NOk*.

Affects (red)

The affects relation represents the potential disturbance a piece of (malfunctioning) hardware may pose on the fulfilment of a function. This could be used to represent physical effects, such as heat generation or vibration that is caused by a broken piece of hardware, that cause a function to fail, even though all of the functional dependencies of this function are in order. The main difference with the *realizes* relation is that this relation represents a non-functional relation between the hardware and the function.

Subfunction (pink)

The subfunction relation ('subfunction_of') represents part of the functional breakdown of higher-level functions into lower-level functions. The relation takes the lower-level function as a source and the higher-level function as a target. Semantically, this means that the higher-level function can only be performed as expected once all of its subfunctions are *Ok*.

Test (blue)

The test relation ('tested_by') connects either a hardware node or a function node to a diagnostic test node. It represents the observability of the health of the hardware node or the capability to fulfil its function of a function node by doing the diagnostic test. A diagnostic test may be connected to multiple hardware or function nodes by means of multiple test relations, if the test provides observability on all those connected nodes.

Observable (green)

The observable relation ('observed_by') represents the observability of either the hardware node's health or the ability to perform a system function correctly by means of a function node on a direct observable node. Similar to the test relation, multiple observable relations can connect different observed nodes to a single direct observable if the target observable provides information on the state of those source nodes.

2.4.2 Non-functional dependencies

As described in Section 2.4.1, functional dependencies constitute the main source of information of the proposed diagnostic methodology. For complex systems however there might be diagnostic problems for which the path from a root cause to the observed functional symptoms might be via non-functional dependencies.

With non-functional dependencies here we mean failure situations, like a leaking pipe or an overheating component, for which as a consequence a function impacts other functions. This is a dependency which in normal conditions is not there, that is such a function does not have an impact on the other functions. During the design of a system, these dependencies might be known to designers but likely are left out from the functional models so in a sense non-functional dependencies can be seen as not documented functional dependencies. These non-functional dependencies are left out based on implicit assumptions made on the environment in which the system is deployed and the normal operational status of its components.

The goal of this section is to introduce an approach to systematically reconstruct these non-functional dependencies for a system. To do so, we propose the workflow shown in Figure 2.5.

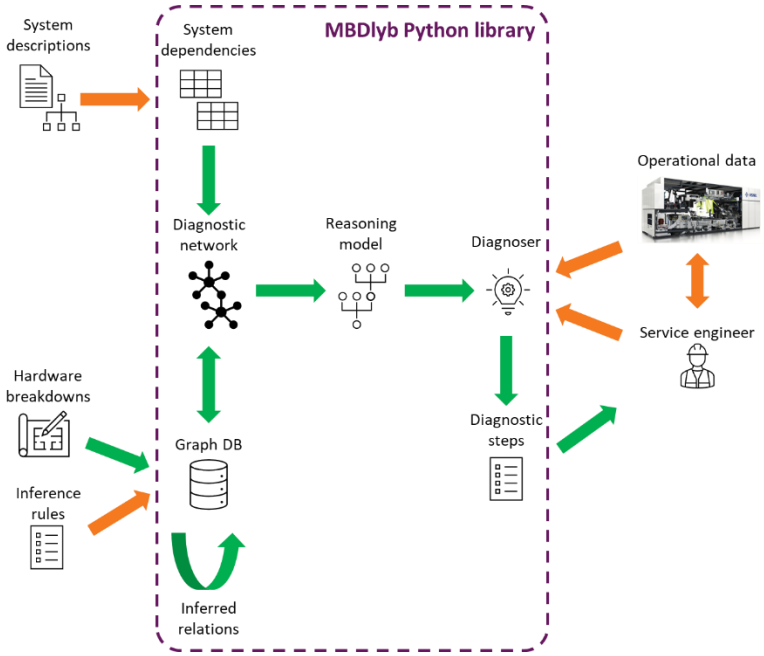


Figure 2.5: SD2Act methodology workflow for inference of non-functional dependencies.

The difference with Figure 2.2 is on the lower left side. In order to reconstruct such non-functional dependencies, we make use of hardware breakdowns and expert-based inference rules. The hardware breakdowns are first transferred to the graph DB, then inference rules are applied on the DB to infer the non-functional relations. In this way, we leverage the DB technology's functionality to quickly perform such an inference, i.e. we do not need to write specific algorithms to infer these new relations. Based on a single inference rule, many non-functional relations can be inferred, which minimizes the effort for designers.

In more detail, the hardware breakdowns contain material and proximity information about parts. This can be gathered from sources like CAD drawings and parts databases. Examples are shown in Table 2.3 and Table 2.4. Table 2.3 shows the distance information, extracted from CAD drawings using the built-in clearance analysis functionality resulting in an $M \times M$

matrix for a system with M HW components. Table 2.4 shows the property information for the parts, where each part can have associated any one of n multiple property value. Examples of properties are the type of material, weight and thermal conductivity.

Table 2.3: Computed distance between hardware components computed from the CAD drawing.

From HW	To HW	Distance
Hardware_A	Hardware_B	d_AB
…	…	…
Hardware_A	Hardware_M	d_AM
Hardware_B	Hardware_C	d_BC

Table 2.4: Property information on each of the hardware components.

HW	Property_1	…	Property_n
Hardware_A	Value_1	…	Value_n
Hardware_B	Value_2	…	…
…	…	…	…
Hardware_M	Value_1	…	…

This information is then added to the graph DB and combined with the information from the functional dependencies via the HW nodes. As an example, Figure 2.6 shows the resulting graph DB after adding the distance information on top of the existing DB of Figure 2.4 from the Hardware_B node to the other hardware nodes. Not shown in the figure is that properties from Table 2.4 above have also been added to the Hardware nodes, such as *HeatSource* or *SensitiveToHeat*.

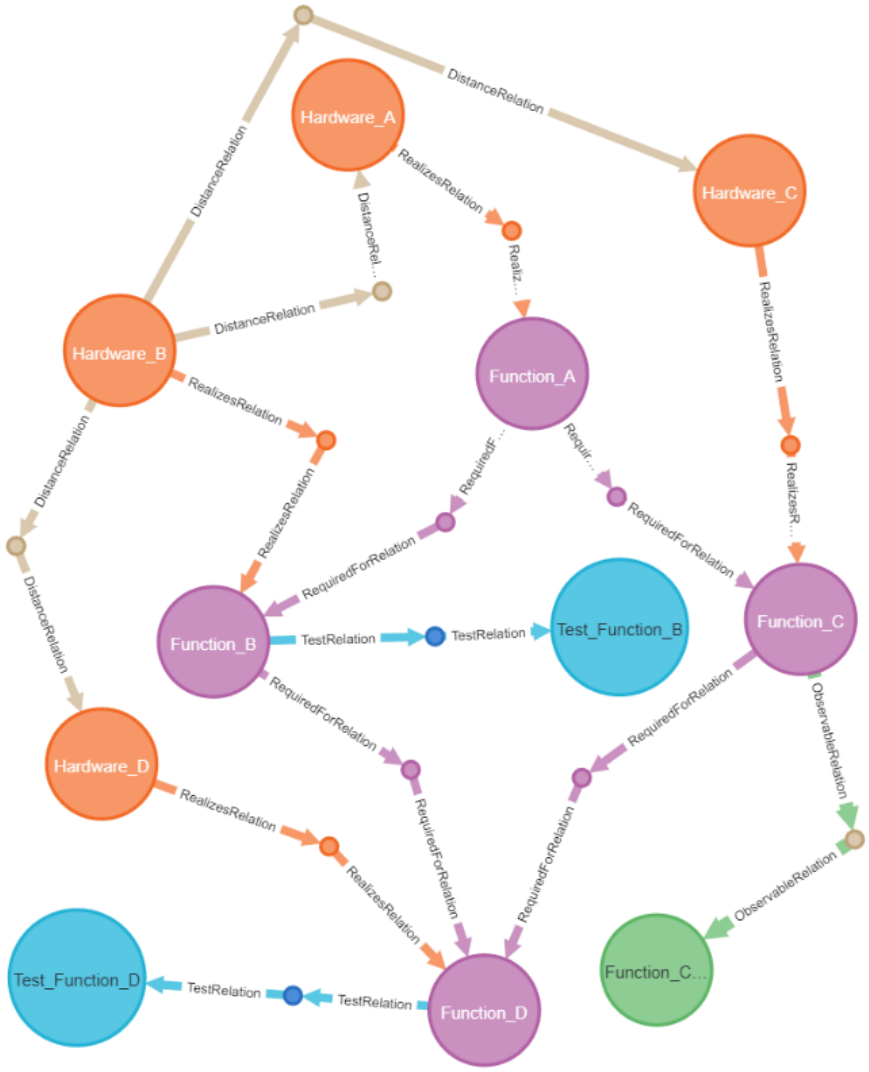


Figure 2.6: Example of graph DB with distance relations.

Once such information is stored in the DB, we can perform expert-based queries that infer non-functional relations. As an example, to infer heat coupling between components close in space, one could perform the following query, where we used Neo4j Chyper query language syntax:

```
1  MATCH (from_HW_node : Hardware{property :  'HeatSource'})-->
2            (d : DistanceRelation WHERE d.mm < min_distance)-->
3            (to_HW_node:Hardware{property:'SensitiveToHeat'})-->
4            (f:RealizesRelation)-->(to_FN_node)
5  CREATE (from_HW_node)-[:InferredNonFunctionalHeatInfluence]->(to_FN_node)
```

In lines 1 to 3 we find the HW nodes with a property HeatSource which are closer than min_distance to HW nodes with a property SensitiveToHeat. In line 4 we then find the function nodes realized by the latter HW nodes. In line 5, for each of combination of the above HW and function nodes we add a new relations of type InferredNonFunctionalHeatInfluence to the DB.

Executing such a query on the DB will then add the relation shown in Figure 2.7. Following our workflow, this inferred relations will then be transformed back into a diagnostic network as an *Affects* relation to be used in a diagnosis since, in the extended network, a failure of Hardware_B can become a cause of a malfunctioning of Function_D.

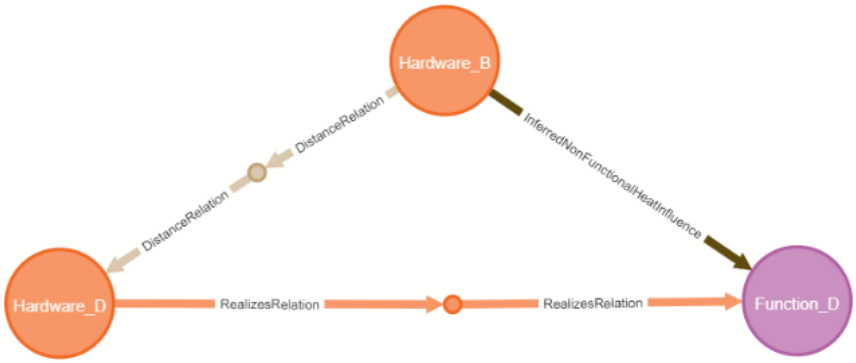


Figure 2.7: Inferred non-functional relation.

Some considerations on this way to infer non-functional relations:

- For large systems, computing the distance between all pairs of HW parts could be too computationally expensive. A possible solution would be to compute distances using the hierarchical decomposition. For example in a recursive way one could compute the distance among any two modules and then among any two sub-module inside each single module.
- The expert-based inference rules should be collected and maintained in a library. This library should also specify the way in which the inferred relations have to be transformed into the diagnostic network.

2.4.3 From system design to a diagnostic network

To express the system dependencies for the creation of the diagnostic models, we propose to use Excel to collect all the information (nodes and relations) in a structured way that should be stored in the Graph DB. The proposed template allows for relatively easy specification of a system at the cost of a number of assumptions, as an Excel template puts some limits on the expressive freedom of the model creation. This is in principle not a problem, as the template is merely used to demonstrate the creation of the model, rather than to provide a fully-fledged tool. An example of the proposed Excel template is shown in Figure 2.8.

			Functions					Observables			
			System	Subsystem_1	Subsystem_2	Subsystem_3		Direct	Direct	DiagnosticTest	DiagnosticTest
Cluster	Type	Name	MainFunction	Function_11	Function_21	Function_31	Function_32	Function_21_Error	Function_31_KPI	Test_Function_11	Test_Hardware_21_31
Subsystem_1	Function	Function_11									
Subsystem_1	Hardware	Hardware_11									
Subsystem_2	Function	Function_21									
Subsystem_2	Hardware	Hardware_21									
Subsystem_2	Hardware	Hardware_22									
Subsystem_3	Function	Function_31									
Subsystem_3	Function	Function_32									
Subsystem_3	Hardware	Hardware_31									
Cost		Time								5	15

Figure 2.8: Example system specification using the Excel template.

The template consists of an individual sheet per (sub)system to be specified. In Figure 2.8, the sheet name is 'System', which represents the top-level view of our example system. The breakdown into smaller parts of our system is done via so-called *Clusters*. For every cluster,

one or more main functions and/or hardware pieces need to be specified. All are assigned a name. These specifications are shown in the first three columns of the template.

The grey column(s) represent the main function(s) of the cluster being specified in the sheet. The name of the displayed cluster specification is 'System' and the main function of this cluster is named 'MainFunction'. Every cell marked green indicates a dependency from the function or hardware piece represented by the row to the function or observable represented by the column. The two marked dependencies of the 'MainFunction' are 'Function21' and 'Function31' of clusters 2 and 3 respectively. This results in those functions to be subfunctions of the 'MainFunction' function of the cluster 'System'.

In the blue columns, only the main functions of the clusters are listed. The second row represents the cluster of the function, while the third represents the function name. Again, green cells represent a dependency from the row to the column. Three interpretations of marked cells are valid:

- Function → Function: this translates to a *Required for* relation from the first function to the latter.
- Hardware → Function, within the same cluster: this translates to a *Realizes* relation from the hardware node to the function node.
- Hardware → Function, in another cluster: this translates to an *Affects* relation from the hardware node to the function node.

The orange columns represent the observables, which can be either 'Direct' or 'DiagnosticTest'. This is specified in the second row and ensures the observables to be translated into the proper node type in our diagnostic model. Whenever a cell is marked green, the observable in the respective column provides information about the function or hardware in the respective row. This is translated to an *Observed by* relation or a *Tested by* relation, depending on the observable type. Additionally, one can specify the costs attached to conducting a diagnostic test. The final row(s) of the table have 'Cost' as main header and each row must have a subheader indicating the name of the costs represented. In this example, only 'Time' costs are considered. Every diagnostic test should have a number assigned for each type of cost to distinguish (which may be 0).

			Functions					Observables			
			System	Subsystem_1	Subsystem_2	Subsystem_3		Direct	Direct	DiagnosticTest	DiagnosticTest
Cluster	Type	Name	MainFunction	Function_11	Function_21	Function_31	Function_32	Function_21_Error	Function_31_KPI	Test_Function_11	Test_Hardware_21_31
Subsystem_1	Function	Function_11									
Subsystem_1	Hardware	Hardware_11									
Subsystem_2	Function	Function_21									
Subsystem_2	Hardware	Hardware_21									
Subsystem_2	Hardware	Hardware_22									
Subsystem_3	Function	Function_31									
Subsystem_3	Function	Function_32									
Subsystem_3	Hardware	Hardware_31									
Cost		Time								5	15

Legend	
Nodes	Relations
Hardware	Subfunction
Function	Required for
Direct Observable Node	Realizes
Diagnostic Test	Affects
	Observed by
	Tested by
	Invalid relation

Figure 2.9: Color-coded mapping of the cells to the relation it is being translated into if the cell is marked green in the template. Grey cells are invalid to mark green.

To represent all possible relations and node type being derived from the template, Figure 2.9 shows a color-coded mapping to the different types of nodes and relations. The grey cells are invalid to mark as a dependency. A visual representation of the model as specified in Figure 2.8 is shown in Figure 2.10.

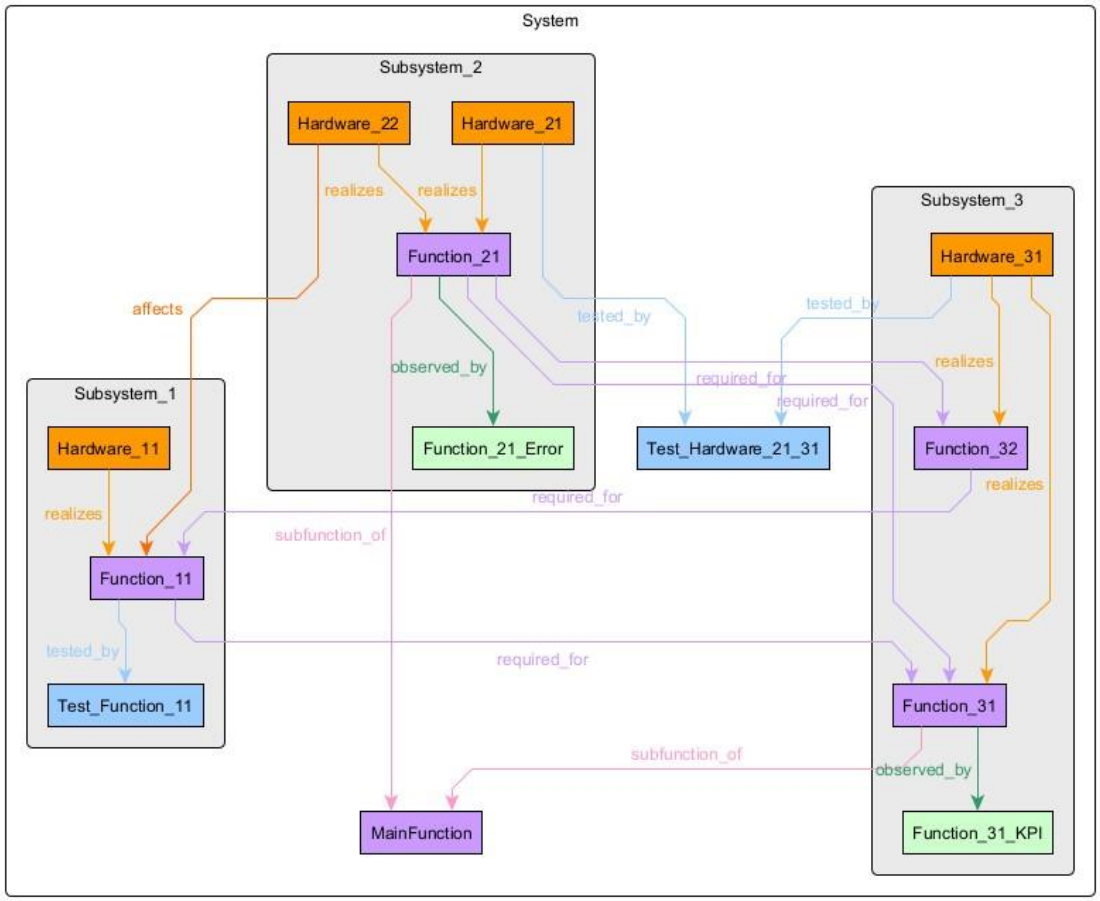


Figure 2.10: A visual representation of the diagnostic model as specified in Figure 2.8.

The Excel template also provides support for hierarchical specification of a model. A cluster can be further detailed by adding a sheet to the Excel file with the name of the cluster. The template of the sheet is identical to the one in Figure 2.8. The grey part of the sheet should reflect the name of the cluster and all of its main functions as specified in the higher-level sheet. For example, 'Subsystem_3' in the example has two main functions, thus two grey columns are needed, as shown in Figure 2.11.

			Functions				Observables	
			Subsystem_3		Subsystem_31	Subsystem_32	Direct	DiagnosticTest
Cluster	Type	Name	Function_31	Function_32	Function_311	Function_321	Function_311_Error	Test_Function_321
Subsystem_31	Function	Function_311						
Subsystem_31	Hardware	Hardware_311						
Subsystem_32	Function	Function_321						
Subsystem_32	Hardware	Hardware_321						
Cost		Time						20

Figure 2.11: Detailing out of 'Subsystem_3' in the Excel template.

This more detailed model of the cluster adds upon what has already been specified for the cluster. Apart from making sure the cluster name and main function names correspond to the upper level, the template works exactly the same. This cluster by itself, without loading it in the context of the higher-level system, is visualized by Figure 2.12.

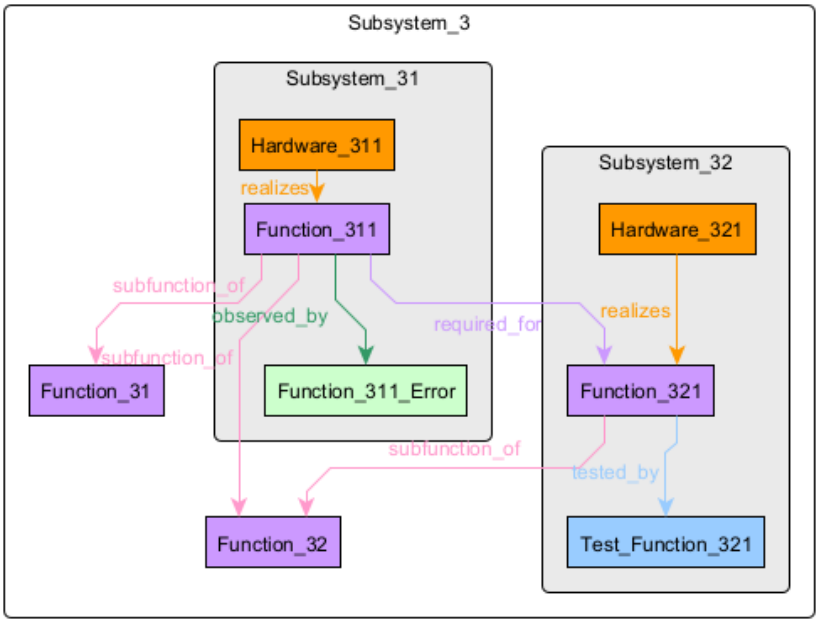


Figure 2.12: Visual representation of only the specification of Subsystem_3, as specified in Figure 2.11.

By detailing out a cluster, the interfaces from other clusters to the detailed clusters are often unclear. Originally, functions of the other clusters were connected to the main functions of the to-be detailed cluster, while these target functions have now gotten one or more subfunctions, as shown in Figure 2.12. Consequently, it is unclear which subfunctions in the detailed cluster are the targets of the external functions that pointed to the cluster's main functions. When loading the model, this will be prompted and a choice should be made by the user of the library. An example prompt for the above system is shown in Figure 2.13. The main function 'Function_32' has been detailed by assigning subfunctions to it and needs to be refined. The user may now chose to either maintain the original main function to be connected, or to clarify this connection to start from either one of the two subfunctions of the main function. In this case, option 3 ('Function_321') was selected. Note that if the selected function also has a more detailed specification available, a new prompt will appear to further refine. This does not happen if an explicit choice has been made to maintain the originally connected function (option 1).

```
Ambiguous source function: Subsystem_3.Function_32 -> Subsystem_1.Function_11,
please specify Subsystem_3.Function_32:
    1. Subsystem_3.Function_32*
    2. Subsystem_3.Subsystem_31.Function_311
    3. Subsystem_3.Subsystem_32.Function_321
Comma-separated number(s):3
```

Figure 2.13: Example prompt to clarify the interfaces to a more detailed cluster.

While these interface descriptions should be part of the model, these are not part of the Excel template. As such, the specification of these interfaces is saved to a separate file in JSON format to enable reloading of the model without having to answer all questions again. While it is suboptimal that these interface specifications are not included in a more structured form and part of the model specification itself, this approach allows the individual clusters to be specified in full isolation, without knowing in which context these will be placed. Upon initialization of the model, its placement is known and the further specification is needed. Future work may come up with a more structured approach that still maintains this benefit.

The final model, after detailing out one of the clusters, is shown in Figure 2.14. The four *Required for* relations crossing the border of 'Subsystem_3' replaced the *Required for* relations to and from the main functions 'Function_31' and 'Function_32'. The main functions in 'Subsystem_3' have no more *Required for* relations connected, as these are now going to subfunctions of those main functions.

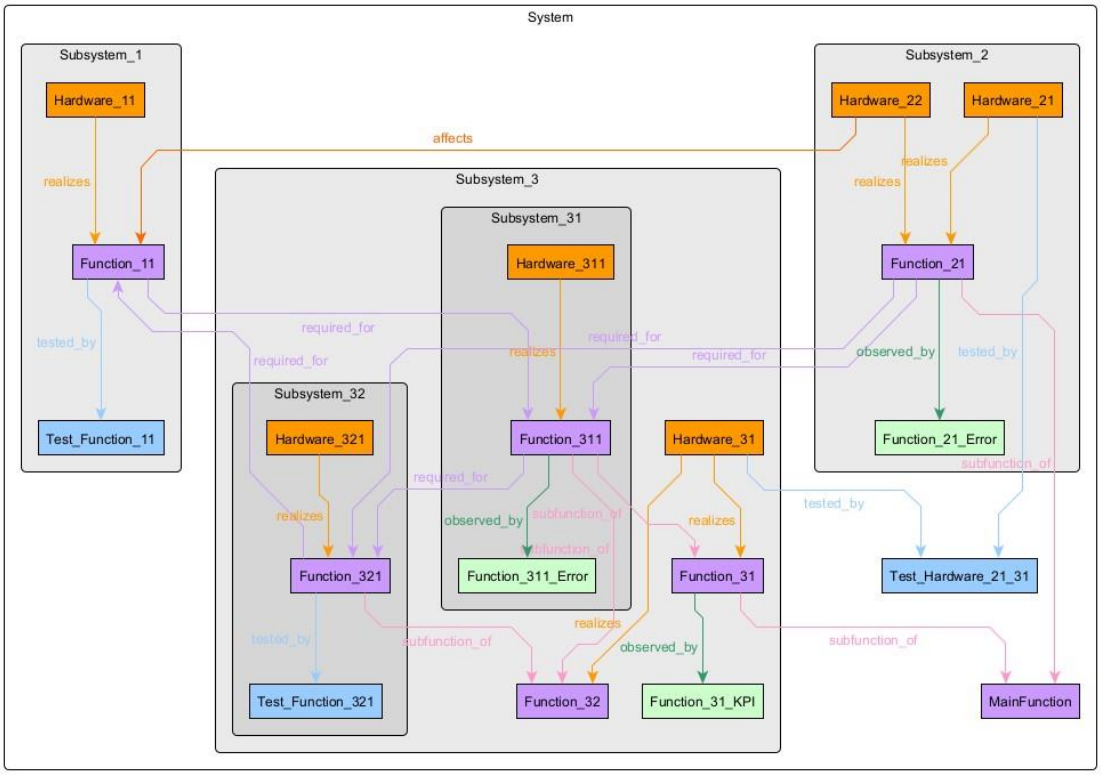


Figure 2.14: The visualization of the final example model, including the detailed out 'Subsystem_3' cluster.

The expressiveness of the current Excel template is limited in some ways to simplify the model specification. These limitations and the assumptions to deal with them are listed below.

- The template does not allow for the specification of individual fault modes of the hardware nodes. Besides the default *Healthy* state, an *Unhealthy* state is created for each hardware node. Since the priors of the individual states cannot be added in the template either, the prior is set to be 99% *Healthy*, and 1% *Unhealthy*.
- Direct observables may sometimes be faulty, being either a false positive or a false negative, meaning that the observable reports *NOk* while the function or hardware does not have a problem, or that it reports *Ok* while there is a problem respectively. To provide room for these faulty types of observations, each direct observable node has a 0.1% chance of being a false positive and a 1% chance of being a false negative. Rationale for these different values is that it is usually more likely to have something not being reported while it should than having something reported mistakenly. In future implementations, these numbers may be changed.
- Like the direct observables, also diagnostic tests may be wrongly interpreted. To provide room for these manual observations, the diagnostic test nodes have a 0.01% change of being a false positive and a 0.1% chance of being a false negative.

2.5 From a diagnostic network to a reasoning model

Before we dive into discussing the different reasoning models, their virtues and challenges, and our implementation of those, we must discuss what is meant by reasoning in this context, and why is it necessary to construct a reasoning model on top of a diagnostic network.

By *reasoning* we mean the deduction of (true) statements about a subject matter off of a collection of established (true) statements. These statements can be deterministic (if it rains the streets are wet) or probabilistic (tomorrow it will rain with 90% probability). An important caveat here is that the statements need not be true with respect to the real world, but with respect to a model. If our model is only approximately correct, or the evidence inserted in the model is incorrect, then its *true* statements are only approximately true.

As we have discussed in Section 2.4, a diagnostic network is a rich knowledge graph designed to contain a functional *representation* of a system. In this representation we include everything that is relevant for the diagnosability of the system, such as nodes representing component health, component functions and their relations, abstract representations of KPIs etc., which allows us to understand how failures propagate through the system.

As such, there is quite a lot of explanatory power captured within the knowledge graph itself, and one could use it as is to 'reason' about the system without need of any *reasoning* model.

Through the introduction of rules and queries, one can use a diagnostic network to deduce that, for example, component A and component B are thermally coupled, since component A is made out of metal, component B provides active temperature control, and they are contiguous. This form of rules-based reasoning is possible within the knowledge graph, but is not the only kind of reasoning that one might want to do on a system.

Most times, the kind of reasoning we are interested in is causal reasoning. We observe effects (system behaviour) and want to know what the causes are: which system components function as expected and which ones do not. Then, two problems arise. On the one hand, one effect can have many causes and we would like to have a way to quantify the likelihood of each one, particularly so in the complex systems we consider. On the other, one cause may only cause an effect with a certain probability.

Hence a knowledge based approach is quickly not good enough. That is why we need to add a probabilistic reasoning framework to our diagnostic network.

2.5.1 Network transformation

While the semantics of all nodes and relations in the diagnostic network are described in Section 2.4.1 the formalization step to transform a diagnostic network into a probabilistic network is still missing. This section specifies the rules needed to transform from the first formalism to the other to allow for reasoning, regardless of which probabilistic formalism is selected.

Every node in the diagnostic network is transformed to a conceptual equivalent in the probabilistic network. We shall refer to this conceptual equivalent as the *probabilistic node* in the selected reasoning formalism. The exact probabilistic formula embedded in each probabilistic node depends on all of the incoming relations to the node, as those represent the dependencies for this node. Essentially, all probabilistic nodes can be seen as AND-gates, as they require all of its parents to be 'as expected'. In details, this means the following.

Hardware

The probabilistic node for a hardware node only represents the prior probability of failure of the hardware it represents. It does not have any incoming edges, thus has no parents. Table 2.5 shows an example conditional probability table (CPT) for a hardware node with only one faulty state, being *Unhealthy*, besides the default state *Healthy*.

Table 2.5: Example conditional probability table for a probabilistic hardware node with only one faulty state.

P(Healthy)	P(Unhealthy)
0.99	0.01

Function

A function node requires all of its parent function nodes, regardless of whether its relation is of type *Subfunction* or a *Required for*, to be *Ok* and all of its parent hardware nodes (via a *Realizes* relation) to be *Healthy* in order to be *Ok*. In all other cases, the function node's state is *NOk*. Table 2.6 shows an example CPT for a function node with two parents.

Table 2.6: Example CPT for a probabilistic function node with two parent nodes: one function node and one hardware node.

Parent function node	Parent hardware node	P(Ok)	P(NOk)
Ok	Healthy	1.0	0.0
Ok	Unhealthy	0.0	1.0
NOk	Healthy	0.0	1.0
NOk	Unhealthy	0.0	1.0

Direct observable

The probabilistic node for a direct observable is similar to the function node, as it also requires all of its parents to be *Healthy* or *Ok* in order to be *Ok*. Main difference is the incorporation of the false positive and false negative rates for the given observable. With these rates set to their default values for diagnostic networks, the CPT becomes like Table 2.7.

Table 2.7: Example CPT for a probabilistic direct observable node with two parent nodes.

Parent function node	Parent hardware node	P(Ok)	P(NOk)
Ok	Healthy	0.99	0.01
Ok	Unhealthy	0.001	0.999
NOk	Healthy	0.001	0.999
NOk	Unhealthy	0.001	0.999

Diagnostic test

The probabilistic node for a diagnostic test is very similar to the direct observable node and takes false positive and negative rates into account. An example for a diagnostic test with a single parent is shown in Table 2.8.

Table 2.8: Example CPT for a probabilistic diagnostic test node with one parent function node.

Parent function node	P(Ok)	P(NOk)
Ok	0.999	0.001
NOk	0.0001	0.9999

2.5.2 Candidate reasoning formalisms

In the project SD2Act 2023 we have investigated 3 different reasoning formalisms: Bayesian networks, tensor networks, and probabilistic programming. Dynamic Bayesian networks were also studied in SD2Act 2022, but have not been further studied and thus will not be discussed in this report.

The choice of investigated reasoning formalisms was motivated by the form of our diagnostic networks and the nature of the available data. Our reasoning formalisms should be discrete and graphical because a diagnostic network captures the structure of a system in a graph with hardware, function and observable nodes whose states are discrete. Given the small number of data available, and the probabilistic nature of the fault propagation and detection in the system, our formalism should also be Bayesian and model-based.

All this considerations point us toward probabilistic graphical models as our best candidate framework. The most general formalism in this category of mathematical models is Factor graphs, of which tensor networks are an implementation (Robeva & Seigal, 2019). Bayesian networks are a subclass of factor graphs and can be implemented separately or as a special class of tensor networks.

2.5.2.1 Bayesian networks

The first formalism considered is that of Bayesian belief networks, hereafter shortened to Bayesian networks (BNs).

A Bayesian network is a probabilistic model that uses directed acyclic graphs (DAGs) to encode joint probability distributions over a set $\boldsymbol{X} = \{X_1, \dots X_n\}$ of discrete random variables.

Every variable $X_i \in \boldsymbol{X}$ has associated a set of parents $Pa(X_i) \subseteq \boldsymbol{X} \backslash X_i$ and a conditional probability table(CPT), $P(X_i|Pa(X_i))$. The children of a node X_i are the variables that have X_i as a parent.

The set of random variables, together with their CPTs, define the nodes of a directed graph whose edges are determined by the dependencies in the CPTs. One edge for each parent→child relation.

The Bayesian network, in the sense of the DAG with the variables and their CPTs, defines the joint probability distribution:

$$P(X_1, \dots, X_n) = \prod_{i=1}^{n} P(X_i|Pa(X_i))$$

In this way one can insert evidence by fixing the value of that variable in the factorization.

The Bayesian network encodes the joint probability distribution implicitly and can be used to answer questions like "What is the probability of a given assignment $(X_1, \dots X_n) = (x_1, \dots, x_n)$?", or "What is the marginal probability of a variable X_k?". That is, what is the result of

$$P_k(X_k) = \sum_{X_{\bar{k}}} P(X_1, \dots, X_n) = \sum_{X_{\bar{k}}} \prod_{i=1}^{n} P\big(X_i\big|Pa(X_i)\big),$$

Where $X_{\bar{k}}$ means that the summation goes over all variable save X_k.

The process of summing over all variables but the target ones is called *variable elimination*, and its complexity depends greatly in the order and the intermediate information that is stored.

In Bayesian networks, the algorithm that performs variable elimination is called *message passing (MP)* or *belief propagation (BP)*, and can compute all marginals in time:

$$O(n \cdot \max(\dim(\mathrm{X_i}))\max(\dim(X_i))^{\max(\deg X_i)})$$

where, $\dim(X_i)$ is the dimension of the variable X_i, $\deg(X_i)$ is the in-degree of X_i in the network, and n is the number of variables.

Observe that for this algorithm to be useful for large networks, the maximum degree of nodes in the network must be independent of n, otherwise the exponential factor in the complexity would quickly take over and make computation impossible. A fully interconnected network where everything is allowed to directly depend on everything else is thus a non-starter. Thankfully, the systems we deal with are not fully interconnected but rather locally connected, they are *structured*. The point here is that structure is not only helpful, it is *necessary* for computation.

For a more in-depth discussion of Bayesian networks, see (Koller & Friedman, 2009).

2.5.2.1.1 Limitations and challenges of Bayesian networks in the context of SD2Act

In the course of the ADiA and SD2Act projects up to 2022, several limitations and challenges of Bayesian networks have been identified, namely:

1. *Enforcing global constraints:* BNs do not allow the user to easily enforce system-level constraints to, for example, express certain laws of physics and force the model to adhere to these at all times.
2. *Modelling stateful systems:* Bayesian networks represent equilibrium states or states at a single moment of time. Stateful systems, such as state machines, which can change their state over time must be approached using dynamic/temporal Bayesian networks, but these cause scalability challenges as the network needs to be copied an unknown number of times.
3. *Reasoning over control and probabilistic loops:* Bayesian networks cannot, by definition, contain loops. Control loops, however, are to be expected in cyber-physical systems, which will translate into Bayesian networks with cycles. Probabilistic loops are another name for causal loops and are to be expected in systems where different components can affect each other in ways that are not directionally clear or unique.
4. *Description of the problem in terms of (probabilistic) relations by design engineers:* Most engineers are not used to express system behaviour in term of probabilistic relations such as CPTs.

To these limitations we must now add three.

5. *Reasoning with continuous variables*: System variables may exist that are not well described by a binary or discrete set of states (e.g. degradation), but rather require continuous random variables to be described. BN in their current formulation cannot easily deal with continuous variables. This constitutes a bottleneck to extending the methodology to performance diagnostics, i.e. the diagnosis of underperforming systems.
6. *Temporal data*: The current framework does not allow one to make use of time series or otherwise ordered data which could resolve some of the diagnostic ambiguity present in a static model. As discussed before, dynamic Bayesian networks are a possibility but have been found to have scalability issues.

7. *Computation of joint entropies*: The assisted diagnostic with hierarchy approach of section 4 relies on computing the conditional entropy of the hardware nodes with respect to the possible tests. This is a computationally expensive task that grows exponentially with the number of health nodes to be considered, and which is now a bottleneck to computation for the current implementation of Bayesian networks.

Some of these limitations no longer apply in the functional modelling framework of SD2Act.

The nature and importance of these limitations must be reassessed in the new modelling framework.

Stepping away from modelling physical behaviour and towards functional modelling can potentially circumvent the need for enforcing global physical constraints (limitation 1), although non-functional relations can still exist that are best described as constraints between hardware nodes in the diagnostic network.

Regarding the modelling of stateful systems and the reasoning of control loops (limitations 2 and 3), the functional approach alleviates the limitations of Bayesian networks since the details of the control loop can now be abstracted away. Controller states no longer need to be tracked in time and are simply described as either performing their function or not. The same can be said, up to a point, about state machines. Further research is needed in this direction.

Limitation number 4, i.e. difficulty of problem description by design engineers still applies but has now changed in nature. Whereas before the difficulty was in translating physical interactions into probabilistic variables and CPTs, the difficulty now is in describing a physical system at the functional level, a problem that has been the subject of much work in the model-based system engineering community. This is no coincidence, as the functional approach to diagnostics was conceived with this synergy in mind.

For limitations 5 and 6, reasoning with continuous variables and temporal data, neither Bayesian networks nor tensor networks provide satisfactory solutions. We are currently looking into probabilistic programming to tackle these. On the one hand, limiting our scope to functional behaviour and machine hard-downs reduces the necessity of continuous variables, because both hardware and functional nodes are discrete ('Healthy/broken', 'OK/NOK'). On the other hand, the data from the field oftentimes comes in the form of time-series' that cannot automatically be inserted into our discrete reasoning engine, and a machine hard-down could be the result of a collection of underperforming factors, none of which is caused by a broken hardware, but which together compound to an out of spec functionality. Our current approach is to have service engineers use their field expertise to interpret the results of tests and KPIs yielding non-binary data and translate them into probabilistic 'OK/NOK' evidence.

Limitation 7 is still under consideration. Possible solutions are a switch to approximate estimation of entropy via sampling, using computationally less expensive bounds on entropy as proxy for the real quantity or using the high-parallelization implementation of tensor networks to reduce computation time of the full conditional entropy.

Let us summarize. The turn to functional modelling has ameliorated but not completely supressed the limitations of Bayesian networks as a reasoning formalism. We have not yet encountered the need for global variables, modelling state machines and control loops has not yet been attempted in a full case-study but we have reason to believe the functional approach goes a long way towards solving limitations 2 and 3. Functional modelling also allows us to piggyback on the systems engineering efforts and expertise to tackle limitation 4.

Nevertheless, some difficulties remain. Limitations 5, 6 and 7 are largely unaddressed. The translation between system behaviour and functional behaviour is far from unique. The

problem with probabilistic loops still has to be addressed. For this last item, we propose a work-around that allows us to compute inference in Bayesian networks with loops. This workaround, described in Section 2.5.2.1.2, comes at a computational cost, and so we later introduce tensor networks, a generalization of Bayesian networks that can directly deal with loops while maintaining all the functionalities of Bayesian networks.

2.5.2.1.2 Loop cutting in Bayesian networks. A house heating example

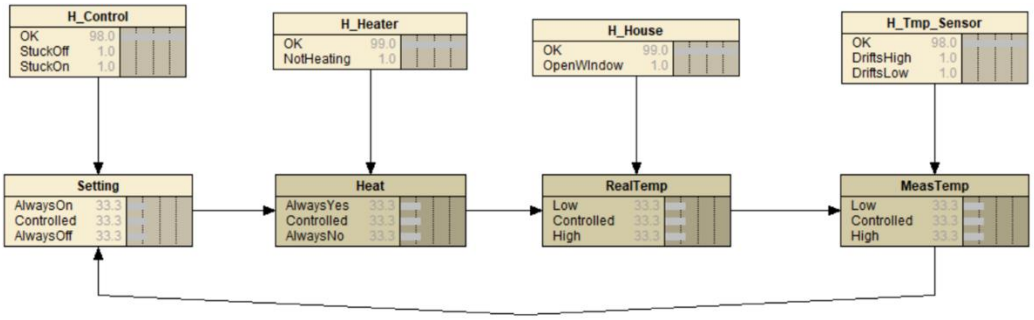


Figure 2.15: A simple example of a loopy system in the form of a house heating control loop.

As was mentioned in the previous section, the issue of directed loops in the causal graph defined by a Bayesian network remains. Bayesian networks with directed causal loops are invalid inputs for the inference algorithms for two reasons:

1. Belief propagation relies on locally sending messages along the DAG that flow from the roots to the leaves of the networks (and then back up). The problem with loops is that messages are then passed along the loop without ever hitting a leaf node and the algorithm never halts.
2. A network with causal loops might not define a well-formed joint probability distribution, meaning that the sum over all variables need not sum up to one. In other words, if the network contains loops, the factorized joint function that it encodes via Equation 2.2 need not be a normalized joint probability distribution.

In the following, we describe a procedure that modifies a BN with a loop allowing us to compute inference in the new network. We illustrate the procedure with an example.

Consider the house heating system in Figure 2.15. Although not in the syntax of diagnostic networks described in Section 2.1, this is a kind of functional model because the states of the variables do not track the instantaneous states of the room, hardware components or settings, but rather their long term behaviour: '*Is the controller controlling?*, '*Is the heater heating?*.

H_Control	MeasTemp1	AlwaysOn	Controlled	AlwaysOff
OK	Low	100	0	0
OK	Controlled	33.333	33.333	33.333
OK	High	0	0	100
StuckOff	Low	0	0	100
StuckOff	Controlled	0	0	100
StuckOff	High	0	0	100
StuckOn	Low	100	0	0
StuckOn	Controlled	100	0	0
StuckOn	High	100	0	0

Figure 2.16: CPT of Setting

The CPTs of the system are all rather trivial and can be easily deduced from the name of the states. Only the CPT for setting is non-trivial (see line 2 of Figure 2.16), this is to allow for combinations where the measured temperature is controlled, but the system is not controlled, as would be the case with a drifting sensor.

This seemingly sensible Bayesian network is ill-defined, as was explained in the previous section. The problem lies in the loop, and therefore we must find a way of formally cutting

the loop (i.e. modifying the network) while preserving the desired correlations in the new network.

The method for cutting the loop proceeds in four steps.

1. Identify a node in the loop and create a copy of the node, meaning, a new node with the same states as the original one, severing the connection between Node and its children, and replacing it with a connection between Copy and Node's children.
2. Create a new node 'Equal' that is a child of the original node and its copy and whose CPT is defined by $P(Equal = 'yes' | Node = n, Copy = m) = \begin{cases} 1, & n = m \\ 0, & n \neq m \end{cases}$
3. Insert as evidence 'Equal=yes'
4. Give uniform priors to 'Copy'.

In the example of the house heating example, the loop is cut by identifying the Node in step 1 with 'MeasTemp' and adding 'MeasTemp1' as its copy node (see Figure 2.17).

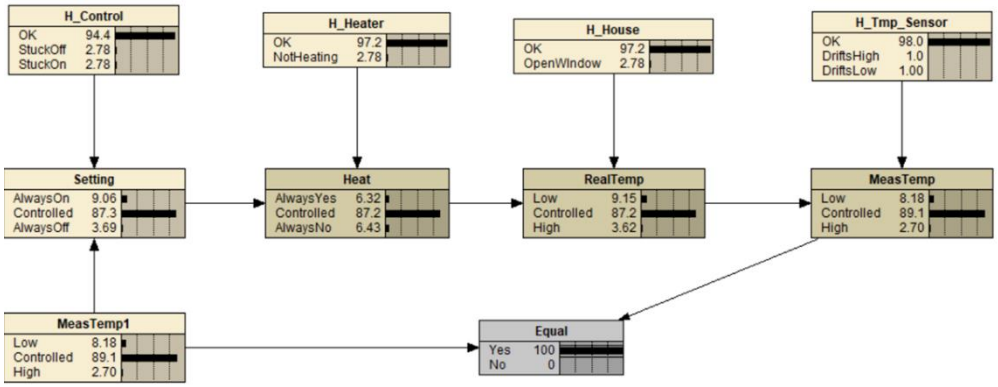


Figure 2.17: Cut loop in a Bayesian network.

This (conditioned) Bayesian network is now well-defined because it contains no direct cycles. The combination of giving 'MeasTemp1' uniform priors and conditioning on 'Equal=yes' makes 'MeasTemp'and 'MeasTemp1' perfectly correlated, which allows us to 'pass' the value of 'MeasTemp' to the variable 'Setting' in the update phase, by actually passing it the value of 'MeasTemp1'. This constitutes an indirect way of enforcing constraints.

One last observation before we move on is that it is the presence of a virtual Evidence node in the network here that allows us to cut the loop. If any of the variables of the loop were to be observed and inserted as evidence there would be no issue in updating the probabilities of the rest of the variables. In essence, it seems that adding evidence to loops is what makes them manageable, and this method is a way to insert evidence in the network when there is none from the field.

This method does not come without drawbacks. We are modifying the network and the causal flow, and the choice of assigning flat priors to B' is somewhat arbitrary, but most importantly, it requires us to find the loops and decide where to cut them. Loop finding is a very computationally expensive task that scales badly in the size of the network.

One solution to this problem is to use tensor networks as a reasoning formalism. As discussed in Section 2.4.2.2, tensor networks generalize Bayesian networks with the added benefit that they can naturally deal with loops. In Appendix A we prove that this way of cutting a loop in a BN is equivalent to the way tensor networks naturally compute loops.

2.5.2.2 Tensor networks

Tensor networks are the second probabilistic reasoning formalism that we study. There are many ways to define tensors, the easiest one is as multidimensional arrays of numbers. Every tensor has a set of *indices* that take values in their respective index domain. Mathematically a tensor $T \in \mathbb{R}^{d_1} \times \ldots \times \mathbb{R}^{d_n}$ being an array of numbers is denoted as: $T = [T_{X_1,\ldots,X_n}] \quad where\ X_j \in \{1,\ldots,d_j\}\ for\ j = 1,\ldots,n,$ or $T = T_{X_1,\ldots,X_n}$ for short. Capital letters denote the indices (lower-case letters denote values that those indices can take). If the number of indices is small, lower-case roman letters are often used to denote indices.

The number of indices of an array is called the tensor *order*, (rank is also used, but should not be confused with matrix rank). A vector is thus an order-1 tensor, a matrix an order 2 and so on (see Figure 2.18).

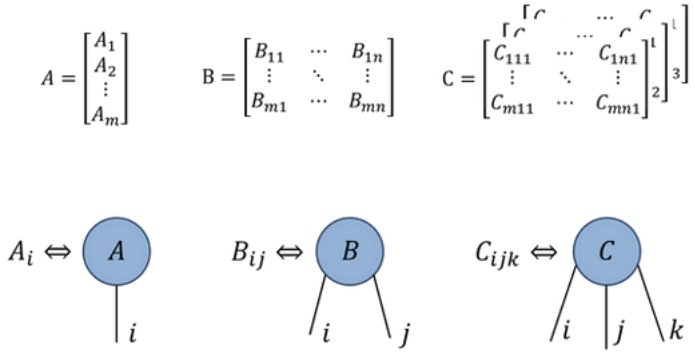


Figure 2.18: Examples of tensors of order 1 to 3. The indices of the tensor act as coordinates on the array.

A tensor network is a graphical representation of a product of tensors with (possibly) overlapping indices.

Equation 2.1

$$\tilde{T} = [\tilde{T}_{X_1,\ldots,X_n}] = \prod_{j=1}^{l} T^{(j)}_{X_{s_j}}$$

where S_i is a subset of the set of all indices in the network. For example, let $S_i = \{2,3,5\}$ be the set of indices of a tensor $T^{(i)}$, then $T^{(i)}_{X_{s_j}} = T^{(i)}_{X_2,X_3,X_5}$

In its graphical representation, the network contains one node for each factor in Equation 2.1, and a (hyper-)edge for every index connecting all tensors that share that index (see Figure 2.19).

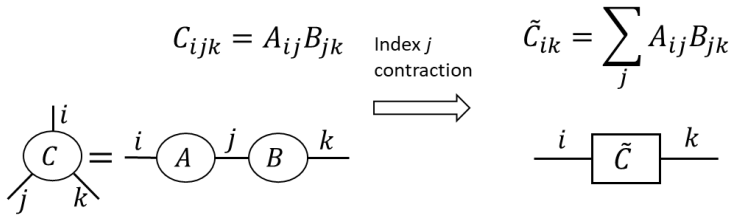


Figure 2.19: Tensor network before and after contraction and its graphical representation.

The basic operation on a tensor network is called *tensor contraction* and consists of summing over one or more of its indices. Contraction is done locally on the network, and the result of contracting an index between two or more tensors is a new tensor whose indices are the indices of the old tensors that were not contracted.

When several indices are contracted at once, the order of contraction can have a huge impact in the complexity of the operation. Although finding the optimal order of contraction is an NP-complete problem for arbitrary networks, there exist many algorithms that can quickly find "good" contraction orders. The more structured and tree-like the network, the easier it is to contract.

Another way of understanding tensors is as maps from their indices to the real numbers.

$T: [d_1] \times \ldots \times [d_n] \to \mathbb{R}$

$T(X_1, \ldots, X_n) = T_{X_1,\ldots,X_n}$

In this notation, a tensor network is a representation of the map:

Equation 2.2

$$T: [d_1] \times \ldots \times [d_n] \to \mathbb{R}$$
$$T(X_1, \ldots, X_n) = \prod_{j=1}^{l} T_j(X_{S_j}).$$

Observe that this notation is reminiscent of that for the joint probability distribution of a Bayesian network.

For the remainder of this report we will mostly use the second notation, but both are useful. Thinking of tensors and tensor networks as multivariate functions allows us to equate them to random variables and probability distributions, while thinking of them as arrays allows us to express them in terms a computer can better handle.

2.5.2.2.1 *Inference in tensor networks*

As has already been hinted at, tensor networks can be used to encode factorizations of joint probability distributions. Just by looking at Equation 2.2, we see that all we really need for this to be a probability distribution is that the factor tensors be non-negative and that it be normalized, i.e. contracting all indices equals 1.

Every index thus represents a variable and every tensor a factor in the factorization of the joint probability distribution.

Tensor contraction is then nothing but marginalization, also known as variable elimination.

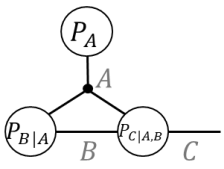
Similarly, inserting evidence is the same as fixing an index to a particular value or adding a delta factor to the factorization and contracting the index away.

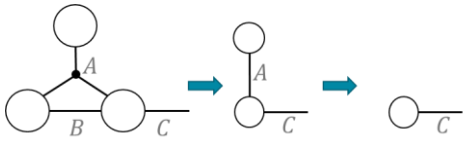
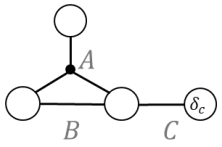
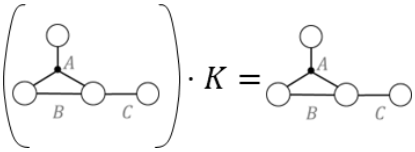
Moreover, because of the form of Equation 2.2, multiplication by constants commutes with the operation of index contraction. This means that we do not need to keep track of normalization constants, we simply compute them after marginalization (a much easier task).

This turns out to be quite handy for application of the chain rule of probability, i.e. $P(X|Y = y) \propto P(X, Y = y)$.

Having marginalization, insertion of evidence and the chain rule we have all the ingredients necessary for Bayesian inference (see Table 2.9). An in-depth analysis of this correspondence can be found in (Robeva & Seigal, 2019) and (Glasser et al., 2019).

Table 2.9: Conceptual parallels between tensor networks and Bayesian inference.

Joint probability distribution	tensor network
$P(A, B, C) = P(A)P(B\|A)P(C\|A, B)$	

Marginalization	Tensor contraction
$P(C) = \sum_{A,B} P(A,B,C)$	
Insertion of evidence	addition of delta node
$P(A,B,C=c) = P(A,B,C) \cdot \delta_{C=c}$	
Chain rule	Constant invariance
$P(A,B\|C=c) \propto P(A,B,C=c)$	

The current implementation of marginalization on MBDlyb computes all marginals sequentially. The complexity of this naïve algorithm (if indeed one could call it an algorithm) is $O(n^2)$, where n is the number of variables in the network. This is worse than the $O(n)$ algorithm for marginalization in Bayesian networks, but still fast enough for us at this stage of development.

Further improvements in the algorithm are possible by storing intermediate computations and fetching their result rather than recomputing them like we do now.

2.5.2.2.2 *Bayesian networks as a subclass of tensor networks*

In a Bayesian network, we put variables and their CPTs into nodes and connect them through directed edges to the nodes' parents and children. That is nothing but a graphical representation of a factorized joint probability distribution of the form:

Equation 2.3

$$P(X_1, \ldots, X_n) = \prod_{i=1}^{n} P(X_i | Pa(X_i))$$

This is but a special case of the factorized function in Equation 2.2.

where $S_i = \{X_i, Pa(X_i)\}$, $m = n$, and each f_i is a conditional probability table.

But as we have seen, this can be also expressed graphically as a tensor network. In general, the changes to the graphical representation are that nodes in a BN that were labelled by a variable name, call it Y, and contained its CPT are replaced by:

a. A tensor with one index per parent of Y and one for Y itself, containing the CPT of Y,
b. One edge per variable. If the variable appears multiple times (because it is a parent to many nodes), this is a hyperedge.

A simple example will illustrate this perfectly. Consider the Bayesian network in Figure 2.20.

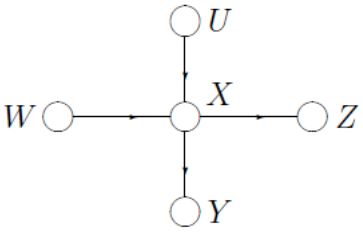


Figure 2.20: Example Bayesian network.

This network is a representation of the joint probability distribution:

$$P(U,W,X,Y,Z) = P_U(U) \cdot P_W(W) \cdot P_{X|U,W}(X|U,W) \cdot P_{Y|X}(Y|X) \cdot P_{Z|X}(Z|X)$$

A visualization of this joint probability distribution as a tensor network looks is shown in Figure 2.21.

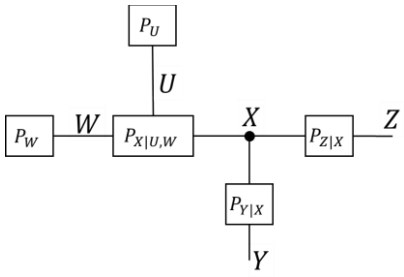


Figure 2.21: Tensor network visualization of the example Bayesian network.

Indeed, we see here that the tensor network contains one tensor per element in the factorization and one edge for each variable, including a hyperedge for X because this variable appears in the tensors $P(X|U,W), P(Y|X), P(Z|X)$.

2.5.2.2.3 *Loops in tensor networks*

Tensor networks can natively deal with both of the issues that make loops incompatible with Bayesian networks.

On the one hand, tensor contraction, the operation used for variable elimination on tensor networks, bypasses the problems encountered by message passing in evaluating loops altogether by not being directional.

On the other hand, the issue of loops of local conditional dependencies that do not amount to a well-defined joint probability distribution is also not a limitation of tensor networks. That is because tensor networks are not limited to such functions. As long as we remember to normalize our results at the end, computation on networks with loops is always possible, and the result, as we will see at the end of this section, is always proportional to a well-formed conditional probability distribution on the modified Bayesian network that breaks the loop (see Section 2.5.2.1.2).

2.5.2.2.4 *Example of a loop in a functional diagnostic network*

Let us illustrate how tensor networks deal with loops with the simple case of two functions, F_1, F_2, each with their own hardware and KPI nodes, and each dependent on the other (see Figure 2.22).

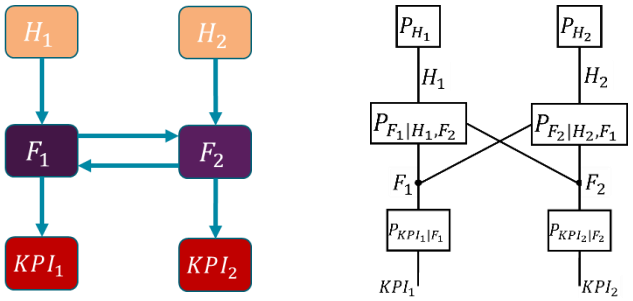


Figure 2.22: A simple functional network containing two function nodes forming a probabilistic loop. The left diagram represents the diagnostic network, while the right one is the resulting tensor network. This is not a complete characterization because the priors and CPTs are yet to be specified.

In order to fully describe this system we need to specify the hardware priors, conditional probability tables of the function nodes and CPTs of the observable nodes. For the prior probabilities of H_1 and H_2 we choose $[P(H_i = Ok) = 0.95;\ P(H_i = NOk) = 0.05]$, while for the KPIs we choose a false positive and false negative rate of 0.01.

Special care must be taken when choosing the CPTs of F_1 and F_2. By default, in MBDIyb we define the CPT of a node to be the logical AND of its parent nodes (Section 2.5.1). If we were to apply the default in this case, our variables F_1 and F_2 would become perfectly correlated. We would not have a loop but rather the same variable twice. In principle, it could happen that a system has a set of function variables that form a loop where each one is strictly required to be '*Normal*' for the next one to be '*Normal*'. In that case, the tensor network would (correctly) make all variables in the loop perfectly correlated. For our purposes now, however, this is not insightful enough. We would still like the hardware nodes to be necessary for their functions to be performed, but will relax the assumption that an abnormal function necessarily makes its functional children perform abnormally. Moreover, we will make it such that an *'Abnormal'* F_1 is unlikely to make F_2 abnormal, while an abnormal F_2 is very likely to make F_1 malfunction. This is made manually, but a working solution is part of the library. The resulting tensor arrays are those in Figure 2.23.

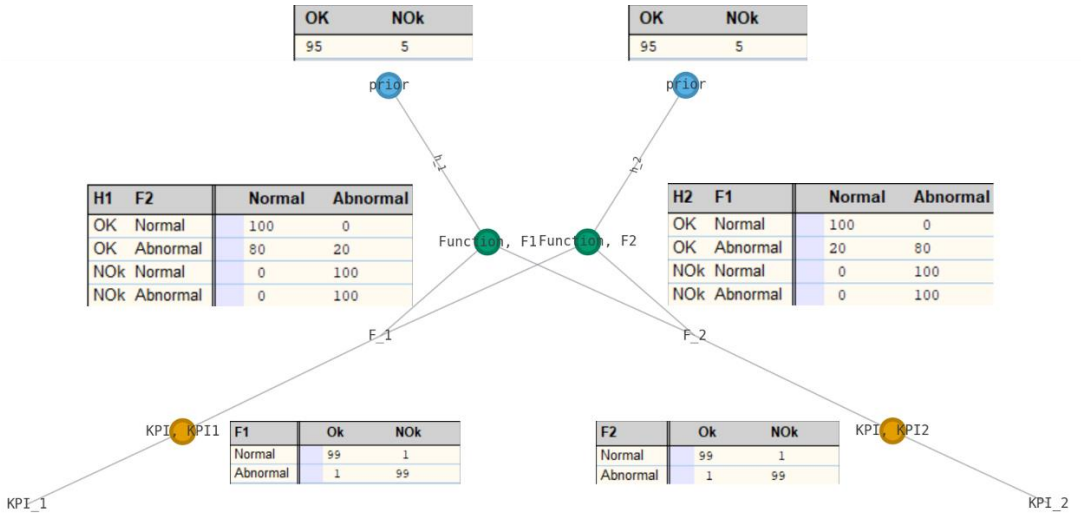


Figure 2.23: Tensor network with tensor arrays fully specified. Observe the difference in the second lines of both middle arrays. These different values make for a weak influence from right to left and a strong influence from left to right.

The first question one might have is what are the posterior probabilities of all nodes prior to inserting evidence. These correspond to rows 1 and 2 of Table 2.10. If we now insert as evidence $KPI_1 = NOk$, or alternatively $KPI_2 = NOk$, the resulting posteriors are those in rows 3-4 and 5-6 of Table 2.10, respectively.

These are the same numbers one obtains if one uses a Bayesian network with the loop cut according to the procedure in Section 2.5.2.1.2.

What transpires from looking at these numbers, especially the cases where we insert evidence, is the asymmetry in the cross-influence between F_1 and F_2. When the evidence is $KPI_1 = NOk$, the posterior probability of H_1 being broken jumps considerably, while that for H_2 stays nearly constant, at the same time, the probability of $F_2 = NOk$ is very high (which is attributed to the strong influence of F_1 on F_2 rather than to H_2 being broken).

In contrast, when $KPI_2 = NOk$, both hardware nodes are considerably more likely to be broken than before in roughly equal ways. What is happening here is that, because of the strong influence of F_1 on F_2, the latter is an indicator for the former, so when strong evidence comes for $F_2 = NOk$, one possibility is $H_2 = Broken$, but another one is that $F_1 = NOk$, which, if true, would be caused by $H_1 = Broken$. We will have another look at these numbers in Section 2.5.2.2.5.

Table 2.10: Posterior probabilities for the system with no evidence inserted, after inserting evidence for KPI_1 and after inserting evidence for KPI_2.

Case	State/variable	H_1	H_2	F_1	F_2	KPI_1	KPI_2
No evidence	*Healthy/Ok*	0.956	0.956	0.822	0.797	0.815	0.791
	Broken/NOk	0.044	0.044	0.178	0.203	0.185	0.209
$KPI_1 = NOk$	*Healthy/Ok*	0.766	0.942	0.044	0.087	0	0.095
	Broken/NOk	0.234	0.058	0.956	0.913	1	0.905
$KPI_2 = NOk$	*Healthy/Ok*	0.832	0.793	0.195	0.038	0.201	0
	Broken/NOk	0.168	0.207	0.805	0.962	0.799	1

2.5.2.2.5 *Advantages and Challenges in tensor networks*

Although tensor networks are a strict generalization of Bayesian networks, in SD2Act we do not make use of their superior expressive power to create models, meaning that the diagnostic models described in section 2 are very BN-like. The use of TNs has therefore no effect on the kinds of models we create, and their impact is felt on the computational back-end of things. In particular, using TNs allows us to easily deal with loops and has implications (both positive and negative) on the computation complexity of marginalization and entropy calculations, and the interpretation of results in the presence of loops. We would be remiss if we did not mention too any of the features of tensor networks not explored in SD2Act, so we close this section with a brief discussion of those.

- *Loops*
  The computation of marginals (and therefore entropies) on networks with loops is possible on the TN without workarounds or additional steps.

- *Computation of marginals*
  Our benchmarking experiments show that the basic contraction of edges (the TN version of variable elimination) on the chosen TN library *Quimb* (see quimb

documentation[2]) is much faster than the equivalent computation of messages on the BN library PyAgrum (see PyAgrum documentation[3]).

However, this advantage is cancelled by the fact that our algorithm for computing all marginals is very primitive, requiring $O(n^2)$ operations, rather than the $O(n)$ required for belief propagation in the BN.

Our results show that belief propagation on a BN takes $\sim K_1 \cdot n$ steps, where n is the number of nodes in the network, whereas the naïve algorithm for computation of marginals on the corresponding TN takes $\sim K_2 \cdot n^2$ steps, where $K_1 \approx 10^{-3}$ and $K_2 \approx 10^{-4}$.

This means that there is both need and room for improvement in the TN algorithm for computing marginals.

The main reason sequential marginalization of the TN has worse scaling than belief propagation on a BN is redundant computation. BP manages to avoid redundant computation by storing the result of all intermediate computations and fetching it every time the algorithm calls for a computation it has already performed.

At the moment we do no such thing in the TN algorithm.

Additionally, we are not making use of the parallelization and GPU acceleration that is supported by Quimb because of the need to run the computation on a Linux environment.

- *Computation of joint entropies*
  Our chosen TN implementation allows for the in-situ computation of functions like the joint entropy of a subset of nodes, whose joint probability would be too large to store in memory (and too costly to compute).

  In the example given in a tutorial[4] for entropy calculations, one can compute the joint entropy of 27 variables (without GPU acceleration) in a network with 304 nodes in under 90 seconds. The computation time is reduced to 13 seconds if one considers only 18 variables.

  Further reductions would follow for less complicated networks (this one is an extreme case) and if one were to use GPU acceleration.

  We must stress that to the best of the author's knowledge, these computations are simply out of reach in the Bayesian network formalism.

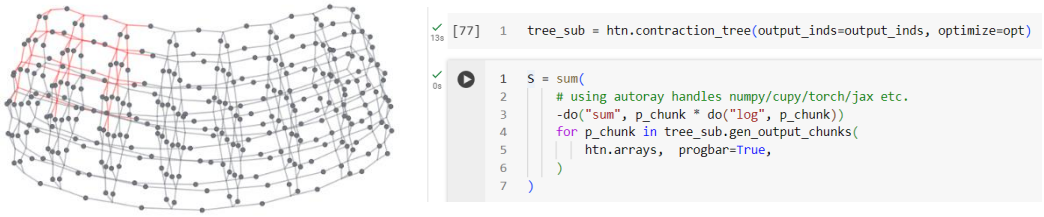


Figure 2.24: [Left] TN with 304 nodes and 18 output indices in red. [Right] Code computing the contraction tree and the joint entropy of all 18 output indices. Observe that most of the computation time is in computing the contraction tree. For very hierarchical and tree-like networks this would be much faster.

[2] https://quimb.readthedocs.io/en/latest/index.html

[3] https://pyagrum.readthedocs.io/en/1.11.0/

[4] https://cotengra.readthedocs.io/en/main/examples/ex_large_output_lazy.html

- *Interpretation of results in networks with loops*
  In the example detailed in Section 2.5.2.2.4 we have a TN with a loop between $F_1$and F_2. In this network, we set the influence of $F_1 = NOk$ on F_2 much bigger than the influence of $F_2 = NOk$ on F_1. As we already discussed, this leads us to reasonable diagnostics in the sense that $KPI_1 = NOk$ makes both F_1 and F_2 very likely NOk but only suspects H_1 of being broken. However, the actual numbers for the posterior probabilities of H_1 and H_2 vary quite a lot with the strength of the mutual influence between F_1 and F_2, which is ultimately a modeler's choice.

  While it seems that the diagnostic result is quite robust to the choice of influence strength, it remains an open question what are the right values for these influences.

  We have already encountered some non-trivial behaviour on an ASML-specific case whose simplified structure is that of

  Figure 2.25.

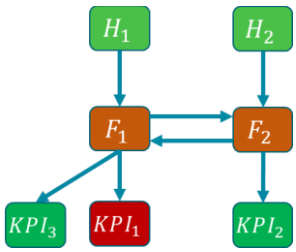


Figure 2.25: A likely false positive that shows the ability of these models to create causal artifacts.

  In this network, KPI_2 and KPI_3 suggest that both components in the loop are healthy while, at the same time, KPI_1 reports *NOk*. This is very likely a false positive of KPI_1, which the tensor network detects because the posteriors of both health nodes report *Healthy* with very high probability. However, depending on the strength of the influence between F_1 and F_2, the network reports a substantial probability that they are both *NOk.* For cross-influences that are both 0.75, the model reports a 33% and 29% chance of F_1 and F_2 being *NOk*, respectively. But this happens at the same time that both H_1 and H_2 are very likely *Healthy.* The functions are making each other malfunction. When the coupling between function nodes is weakened to 0.75 on one direction and 0.05 in the other, the causality loop (predictably) disappears.

  What we have here are malfunctioning functions that are not caused by any malfunctioning hardware. This is an artifact of the model, but one that does not seem to lead to a misdiagnosis. More investigation is required.

- *Features of tensor networks identified but not realized in MBDIyb*
  - *Hyperedges as global variables and constraints*. We consider this not necessary at this stage (see Section 2.5.2.1.1).
  - *Non-causal relations.* The biggest unused feature of tensor networks is their ability to encode non-causal relations through the use of tensors whose rows (or columns) do not sum up to one, i.e. tensors that are not conditional probability tables.
    For example, consider two Boolean variables, X_1 and X_2. If we want to enforce that $X_1 = X_2$ (no causality here, just a constraint), one can introduce in the network a tensor δ whose array is simply a 2x2 identity matrix. One could go further and enforce any equation on a group of variables of the form $f(X_1, \dots, X_k) = 0$ by introducing a tensor T whose array is defined by:

$$T_{X_1 \dots X_k} \begin{cases} 1 & if\ f(X_1, \dots, X_k) = 0 \\ 0 & else. \end{cases}$$

In physics, when two particles affect each other we do not use the concept of causality, but rather, we say that they interact. This is then translated into an interaction term in the equations of motion of the system.

One could attempt the same in diagnostic networks where we have two functions that directly affect each other, translating a loop into an interaction tensor (see Figure 2.26). As long as it is a non-negative tensor, any interaction tensor G_{X_1,X_2} is allowed.

Figure 2.26: [Left] Interaction between variables expressed as a bidirectional causal relation. [Right] Interaction expresses as a non-causal factor.

- *Correspondence to other computational formalisms:*
  It is known in the tensor network community that tensor networks can be used to encode a few other computational formalisms. While here we have used the correspondence between tensor networks and factor graphs, the former can also be used to encode, among others, Hidden Markov Chains (HMC), Borne machines, quantum circuits or satisfiability formulas. In many of these there exist algorithms for computing various quantities of interest, sampling, parameter learning and approximating probability distributions, see (Glasser et al., 2018), (Rams et al., 2021) and (Bañuls, 2023).

- *Modelling with continuous variables:*
  It is possible to generalize tensor networks to the setting where the variables are continuous variables described by exponential probability density functions (like Gaussian, Exponential, Beta, etc.) and the factors are no longer arrays of numbers but *functions*, or rather transformations on the random variables. This is known in the literature as *factor graphs* (tensor networks and discrete factor graphs are the same thing). While continuous factor graphs are much more expressive probabilistic models than tensor networks or Bayesian networks, inference on FG is much more complicated. There exist several implementations of continuous factor graphs, such as the Julia-based library RxInfer[5], or the C++-based GTSAM[6].

2.5.3 Chosen reasoning formalism

In this section we have considered two reasoning formalisms for diagnostics, Bayesian networks and tensor networks. We have seen that the latter include the former and that, although many of the limitations of Bayesian networks identified in the ADiA project have changed (see Section 2.5.2.1.1) some challenges remain for which tensor networks are the more appropriate formalism. We have also discussed the unique features of tensor networks and their limitations. In light of this discussion, we have decided to use tensor networks as our primary reasoning formalism, using the python library *Quimb* as our implementation of tensor networks. Nonetheless, we have decided to continue supporting Bayesian networks

[5] https://github.com/ReactiveBayes/RxInfer.jl

[6] https://gtsam.org/

for the time being, implemented using the python library *PyAgrum,* given the familiarity of some of the authors with the software and the minimal overhead that this incurs.

2.6 Iterative diagnosis

In this section, we will describe the *iterative* diagnostic process that forms a core part of our methodology as well as theoretical learnings about the diagnostic process. In a nutshell: by performing service actions on the system and feeding their results back into the diagnostic system, we can recommend service actions that are perfectly aligned with the needs of the diagnoser.

The following will be our running example:

> *Your cherished CD Radio player, a relic from the golden 90s, suddenly falls silent. The music that once filled your room is no more. You're determined to breathe life back into this piece of nostalgia. So, what's your first move? Do you dive into settings, tweaking and turning knobs in hopes of hitting the right combination? Or do you opt for a fresh power source, replacing the old battery with a new surge of energy?*

Approaching this conundrum systematically, you might be tempted to *take the most insightful service action*, assuming, of course, you have a measure for that. After all, the more insights you have on the status of the system, the quicker you can make it work again. But what if it turns out that the most insightful diagnostic action you could take on the CD Radio player is a comprehensive measurement of its main board? For this task you would need a specialist measuring device that costs a significant amount of money and will also take a few days to be shipped to you. So, before going down that road, it may be much more sensible to consider a *less insightful but more affordable* diagnostic test, such as trying different settings to determine whether it's only the CD that stopped working or also the radio receiver.

This example illustrates the common thread of this section: striking the right balance between diagnostic insight and diagnostic cost to find the best next diagnostic action. To get there, we need to take a few steps. First, we will discuss the operational diagnosis loop in more detail. We then move towards the workings of the diagnoser and discuss both criteria – *diagnostic information gain* and *cost of diagnostic actions* – in isolation. Finally, we will describe our methodology for combining those criteria into a single value that expresses the value of the diagnostic action in the current context.

2.6.1 Operational diagnosis loop

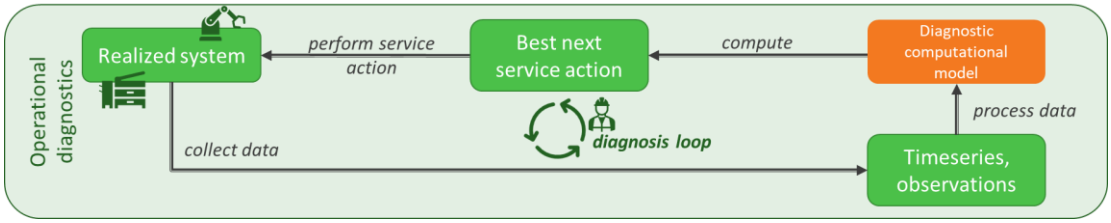


Figure 2.27: The operational diagnosis loop consists of a realized system, observations, a diagnostic computational model and service actions.

The operational diagnosis loop, as depicted in Figure 2.27, is an iterative process for root cause analysis in malfunctioning high-tech systems. This loop consists of four essential components:

- the realized system,

- a diagnostic computational model,
- timeseries and other observations,
- service actions.

The operational diagnosis loop starts from a realized system in its operational context, i.e. the system is already built and running in its environment.

The design process of the system is (usually) concluded. Hence, a diagnostic action may help us find the problem but will not change the system design to make it easier to find the problem.[7]

The realized system produces operational data in the form of timeseries and observations. For example, this could be error or event logs, logs of measurements that the system collects during operation, or observations such as a warning indicators. In the context of the SD2Act project, we also call these *direct observables* as the information can be obtained from the system directly, without any additional means (see also Section 2.4.1).

The direct observables (timeseries and other observations) are then fed into the diagnostic computational model. For details about the computational model, we refer to the previous chapter. Crucial for the operational diagnosis loop is that the diagnostic computational model calculates an optimal next service action that can be performed on the system.

Finally, the best next service actions are performed on the realized system and their results are fed back into the diagnostic computational model. This closes the diagnosis loop. Note that service actions in the context of SD2Act are usually considered to be measuring KPIs or *indirect observables*. They provide measurements or insight into system components for which – as opposed to direct observables – some effort has to be taken. The results of these efforts are entered as evidence into the reasoning model. We do not consider replacements of parts or trying alternative settings as service actions in the context of SD2Act, for these types of service actions see (van Gerwen et al., 2023). See Figure 2.28 for an illustration of the goal of assisted, iterative diagnostics as opposed to conventional diagnostics.

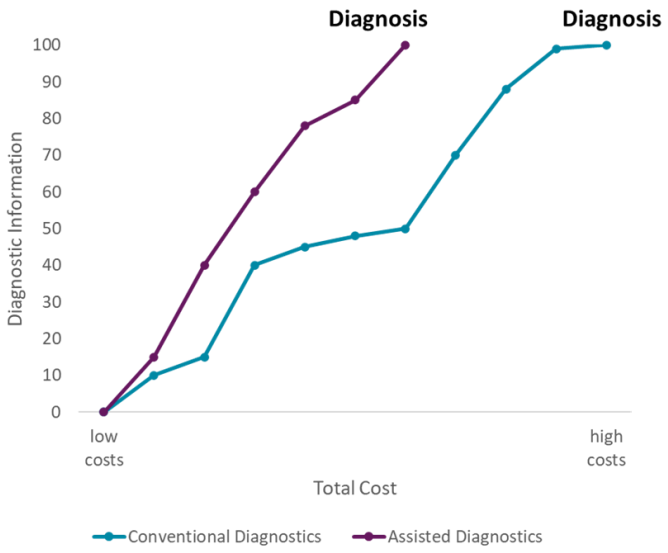


Figure 2.28: This graphic illustrates the goal of iterated diagnostics: reducing the number of service actions (the dots) as well as the total cost of the diagnostic procedure. Through iterated diagnostics we aim to push the graph from right to left.

[7] One can also consider a diagnosis loop in the context of designing systems. The goal there is to improve system design in order to facilitate diagnostics in the operational context.

2.6.2 Recommending the next best service action

In this section and its subsections, we will describe our methodology for recommending the next best service action. To begin with, there are two main criteria for choosing a service action:

- The (expected) information gain of a service action, i.e., insight the service action promises to provide on the status of the system.
- The (expected) cost of performing said service action. Cost may depend dynamically on the system's current state, and may also make reference to duration.

If one considers only cost, then one might end up performing many affordable but useless tests. On the other hand, if one considers only information gain, then one might recommend tests that are so expensive that they are not considered cost-effective to carry out. Potentially, the same information gain could be achieved by combining a few less informative tests that are jointly less expensive. These considerations illustrate that both cost and information gain should be considered when choosing the next service action.

We will now discuss both criteria in isolation and then move on to approaches for combining them.

2.6.2.1 Diagnostic information gain

The first criterion for recommending service actions is diagnostic information gain, i.e., we will now discuss a measure for how much we learn about the health of the system by performing a service action.

2.6.2.1.1 Technical preliminaries and scope of the methodology.

The minimal assumptions for applying the techniques discussed in this section are the existence of a diagnostic model for a complex system satisfying the following assumptions:

A random variable T_j is associated to every test j of the system.

A random variable X_i is associated to every hardware component i of the system such that we can retrieve the probability $P_E(X_i = \text{"Ok"})$, potentially dependent on some evidence E, i.e. a set of observed variables with the corresponding observations.

Given sets of random variables X and Y, we can retrieve the entropy of X, $H(X)$, and the entropy of X conditioned *on* Y, $H(X|Y)$.

Note that we need not assume the random variables to be discrete: the assumptions above allow for reformulations in the continuous setting.

2.6.2.1.2 Methodology

Our methodology for recommending a diagnostic test solely based on information gain is to perform the diagnostic test T_i that minimizes the conditional entropy of the set of hardware nodes $X = \{X_0, \dots, X_n\}$, given some evidence E:

$$\text{argmin}_i \, \text{H}_E(X|T_i) \; = \; \text{argmin}_i \, \text{H}_E(X_0, \dots, X_n|T_i).$$

The rationale behind this methodology is to select the diagnostic test that is expected to reduce the overall uncertainty about the system's health the most.

2.6.2.1.3 Scalability of the methodology

Calculating conditional entropy is an extremely expensive operation because it requires the computation of a joint probability table involving all variables involved. For example, calculating the conditional entropy for a test in a system with just 100 relevant hardware components with 2 states will be practically unfeasible as it requires computing a joint

probability table with 2^{101} rows. In practice, we therefore limit the entropy calculation to approximately 10 hardware components that are most likely to be faulty. We will now describe this approach formally.

Let the ranking $r: \{0, \dots, n\} \to \{0, \dots, \mathrm{n}\}$ be a bijection such that $i < j$ implies

$$P\left(X_{r(i)} = Ok \middle| E\right) \leq P\left(X_{r(j)} = Ok \middle| E\right),$$

where E is some evidence consisting of the measuring results of our direct and indirect observables. Our recommendation is to minimize $\mathrm{H_E}\left(X_{r(0)}, \dots, X_{r(m)} \middle| T_i\right)$ for a computationally feasible $m \leq n$. In other words, the next recommended diagnostic test is:

$$\mathrm{argmin}_i \, \mathrm{H_E}\left(X_{r(0)}, \dots, X_{r(m)} \middle| T_i\right).$$

While this approach makes the computations tractable, it has two main drawbacks. To avoid cumbersome notation, we will now write H when also H_E could apply.

First, given our current approach for scalability of the methodology, it is possible that we undervalue the information gain of diagnostic tests by ignoring their impact on lesser suspected hardware components. A test T_i that does not influence the probability $P\left(X_{r(k)} = Ok \middle| E\right)$ for $k \leq m$,[8] will have worst-case conditional entropy in that $\mathrm{H}\left(X_{r(0)}, \dots, X_{r(m)} \middle| T_i\right) \geq \mathrm{H}\left(X_{r(0)}, \dots, X_{r(m)} \middle| T_j\right)$ for every diagnostic test T_j. This is problematic as it is simultaneously possible that $\mathrm{H}(X|T_i) \leq \mathrm{H}\left(X \middle| T_j\right)$ for some diagnostic test T_j.

A second, less significant drawback is that the recommendation of diagnostic tests at the beginning of the process may happen at random. This happens, in particular, when there are more than m hardware components with equal least priors[9] and corresponding diagnostic tests with equal rates for false positive and false negative. In such situations, all these diagnostic tests will receive the same conditional entropy value with respect to the hardware components $X_{j(0)}, \dots, X_{j(m)}$ selected as relevant.

Despite these drawbacks, our approach for scalability works sufficiently well across our test cases. Further investigation of other approaches for scalability may still be fruitful. One may consider, for example:

1. Recommend the diagnostic test T_j that minimizes the average entropy $\mathrm{H}\left(X_i \middle| T_j\right)$ for all i.

2. Partition the set of diagnostic tests X into l buckets of maximal size m, X^k, such that $\mathrm{H}\left(X^k \middle| T_j\right)$ becomes computationally feasible. Then minimize the average of all these bucket entropies. Note that the buckets could be randomly generated as well as created according to some heuristic, e.g., by bucketing functionally related components.

Another way to achieve above points is to investigate estimations such as Shearer's inequality and work generalizing it, see (Madiman & Tetali, 2007).

It is crucial to note that every such approach will make slightly different selections and optimizes for slightly different goals. Therefore, extensive testing on practical examples is unavoidable.

An alternative approach might be to investigate efficient algorithms for calculating (approximate) entropy when large numbers of random variables are involved.

[8] For example, this may happen in a Bayesian Network if $X_{j(k)}$ and T_i are d-separated for all $k \leq m$ given the evidence E.

[9] Mathematically speaking, in this case there are several maps $j, k, l, \dots$ satisfying the conditions of the second paragraph of this section such that $j(i) \neq k(i) \neq l(i) \neq j(i)$, for all $i < i_0 < m$, where i_0 is large enough.

2.6.2.1.4 *The effect of prior probabilities*

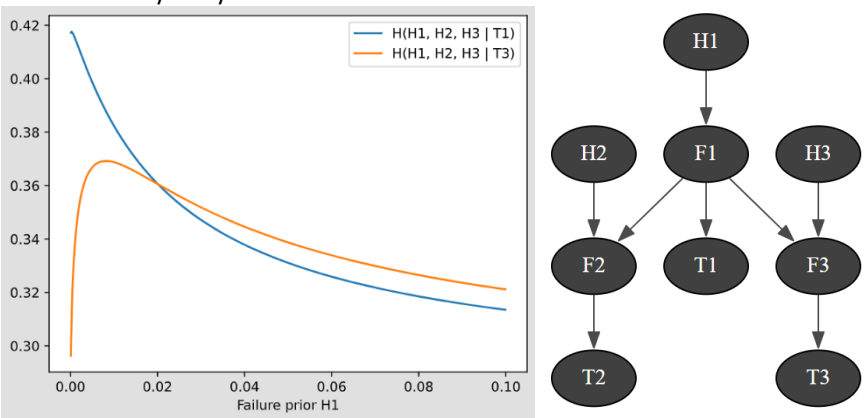


Figure 2.29: A toy example illustrating the dependence of entropy-based recommendations of next diagnostic actions on prior probabilities on the health of the corresponding hardware nodes. The model has evidence inserted that T2 indicates a failure ("NOk"). In the graph, we vary the prior for health H1 between 0.00 and 0.10 while keeping everything else fixed (failure probabilities of H2 and H3 are 0.01). We can see that there is a clear dependence on the failure prior in H1 on whether test T1 or test T3 has lower entropy (on the y-axis) and is, therefore, recommended.

As a final remark on diagnostic information gain, we would like to point out that the prior probabilities for failures of hardware component crucially impact which diagnostic test will be recommended. For illustration, consider Figure 2.29 where we compare the conditional entropy of all hardware components with respect to two different diagnostic tests. We vary the prior failure probability of one test and keep everything else fixed. In conclusion, it crucially depends on the prior probabilities which diagnostic test is expected to decrease entropy the most. In other words: the order in which tests are recommended depends on the prior probabilities (among other things). Hence, choosing at least approximately correct prior probabilities is crucial in the process of diagnostic modelling.

2.6.2.2 Cost of diagnostic actions

We will now move on towards our modelling of costs of diagnostic actions. Returning to our CD radio player example, recall the diagnostic test of performing a comprehensive analysis of the CD radio player's main board with a dedicated measuring device. The cost of this test depends not only on the time it takes the engineer to perform the measurement (and potentially their salary, etc.) but also on whether or not the measuring device is available, whether or not the CD radio player has already been disassembled, and so on. To model the cost of a diagnostic test, we therefore consider fixed costs and dynamic cost that depend on the state of the system that is currently being diagnosed. We will now describe our methodology of dynamically computing cost of diagnostic tests.

2.6.2.2.1 *Types of costs and fixed costs*

To allow for insightful information as well as potential optimization later, we assume costs to be broken down into various types that are fixed for each test T_i. These fixed cost in various types are not dependent on the overall state of the system to be diagnosed. This breakdown in types is particularly helpful when the decision about the next diagnostic action depends not only on its total cost but must also be optimised with respect to other constraints, such as time, workforce available, or similar.

For improved readability, we restrict ourselves to diagnostic costs of a single type in this report. An adaptation to multiple types of cost is straightforward.

2.6.2.2.2 *System state and dynamic costs*

Informally, we can think of the system state as a finite collection of Boolean conditions, $c_0, \ldots, c_n$, describing a set of changing properties of the system that are relevant for the costs of diagnostic tests. In our CD radio play example, such a condition could encode, for example, whether the case of the CD Radio player is open ($c_0 = 1$) or closed ($c_0 = 0$), or whether the measuring device is directly available for inspection ($c_1 = 1$) or not ($c_1 = 0$). Dynamic costs are accrued whenever these conditions must be changed for a diagnostic test. For example, a diagnostic test might require the case to be opened while it's currently closed, or a measuring device to be ordered that is not locally available.

Formally, a system state is a function $s: \{0, \ldots, n\} \to \{0, 1\}$. We will also use the notation $c_i^s = s(i)$ to denote the value of system condition i in state s. Informally, $c_i^s = s(i) = 1$ means that condition i is true in state s, and $c_i^s = s(i) = 0$ means that condition i is false in state s. A dynamic cost function is a map $c: \{0, \ldots, n\} \times \{0,1\} \to \mathbb{R}^{>0}$ where $c(i, 0)$ is the cost of setting condition i to 0, and $c(i, 1)$ is the cost of setting condition i to 1.

To every diagnostic test T_i, we assign a (potentially empty) set C_i of tuples (j, b), where $j \leq n$, and $b \in \{0, 1\}$. The intended meaning is that $(j, 0) \in C_i$ if and only if test i requires condition j to be false, and $(j, 1) \in C_i$ if and only if test i requires condition j to be true. If a condition j does not occur in any tuple in C_i, then the intended meaning is that test i depends neither on c_j being true nor on c_j being false.[10]

Finally, given a system state s, a dynamic cost function c, and a set of conditions C_i for a diagnostic test T_i, we can compute its dynamic cost as follows:

$$c_{dyn}^s(T_i) = \sum_{(j,b) \in C_i \text{ with } c_j^s \neq b} c(j, b)$$

The total cost of performing a diagnostic test consists of (the sum of) its fixed costs, denoted by $c_{\text{fix}}(T_i)$, as well as its dynamic costs:

$$c_{total}^s(T_i) = c_{fix}(T_i) + c_{dyn}^s(T_i)$$

Note that we left the system state implicit in the previous equation; if necessary, it may be denoted by adding a superscript s to any of the cost functions that depend on it. When a diagnostic test T_i is performed, the system state s is updated to s_{new} according to the conditions required by the diagnostic test:

$$s_{new}(j) = \begin{cases} b, & \text{if } (j, b) \text{ in } C_i, \\ s(j), & \text{otherwise.} \end{cases}$$

2.6.2.3 Combination of information gain an cost

Our approach for combining information gain and cost is to combine them into a single *loss value* for aggregating recommendations that take both criteria into account.

Recall from previous section that we want to minimise the conditional entropy $\mathrm{H}(X|T_i)$, where X is a suitable set of random variables describing the system health, and T_i is the diagnostic test under consideration. Moreover, we denote the total cost of performing test T_i by $c_{total}(T_i)$.[11]

[10] In this situation the test can thus be performed independently of the value of c_i. If, however, the costs of performing the test are different depending on values of c_i, then one may need to introduce a helper condition c_j that models those costs. If required by the diagnostic context, one could, of course, make an easy modification to move the dynamic costs per condition from system-wide to test-specific (or add an option for test-specific costs).

[11] While the total cost may depend on system state s if the cost of the diagnostic test has a dynamic component, we usually leave it implicit to avoid cumbersome notation.

To start our discussion, it is helpful to consider the following informal formulations of our two minimalization criteria: by minimising the conditional entropy $\mathrm{H}(X|T_i)$, *we minimise the total number of diagnostic steps.* On the other hand, by minimising the cost $c_{total}(T_i)$, *we minimise the cost of the next diagnostic action*. We would like to minimise both at the same time. Of course, this is not always possible as sometimes the most insightful diagnostic action will also be the most expensive (see the CD radio player example from the introduction of this chapter). Hence, we need to balance the two appropriately to achieve a reduction of total diagnostic costs by aiming for a balanced reduction of both the total number of steps as well as the cost of the next action.

Formally, we propose the following formula for aggregating the loss $l(T_i)$ of a diagnostic action T_i:

$$l_b(T_i) = b\frac{\mathrm{H}(X|T_i)}{\max_i \mathrm{H}(X|T_i)} + (1-b)\frac{c_{\mathrm{total}}(T_i)}{\max_i c_{\mathrm{total}}(T_i)},$$

where the balance $b \in [0,1]$. It is easy to see that minimising l_0 is equivalent to minimising c_{total} and minimising l_1 is equivalent to minimising $\mathrm{H}(X|T_i)$. Any value strictly between 0 and 1 minimises a combination of cost and entropy. Note that we normalise both entropy and cost with respect to their maximal values (at the current stage); the range of l_b is thus contained in $[0,1]$. Normalisation is necessary to overcome imbalances resulting from the very different scales of entropy and cost (the difference can easily span several orders of magnitude). The latter point is also an advantage we see of our loss function over other approaches, such as minimising the information gain per cost.

The recommended next service action is the one that has least loss value. The results of simulations, see Figure 2.30, show that choosing a good balance value b can dramatically decrease the overall diagnostic costs. The simulation also shows that the optimal balance value depends not only on the structure of the diagnostic model but also on how the costs are distributed. It is therefore vital future work to develop an optimisation method for balance values. Brute forcing the balance value might be feasible if the models are not too big and do not change too often but other means of optimisation are worthy of investigation. First experiments with reinforcement learning seem promising[12].

[12] Reinforcement learning for this optimisation process is expected to be further investigated through a student internship in 2024.

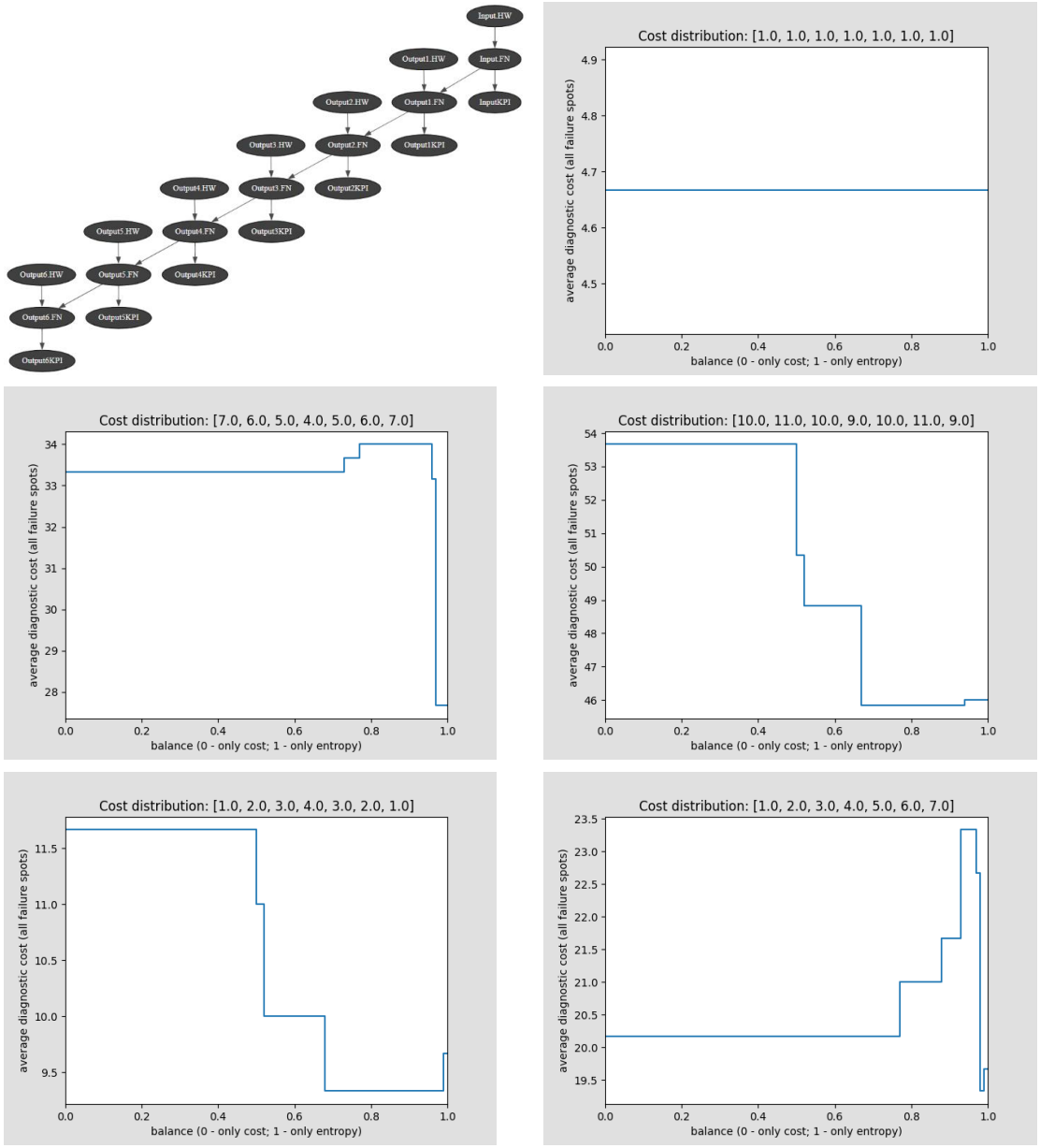


Figure 2.30: Results of simulating the diagnostic process in the diagnostic model shown in the upper left corner. The balance values b were simulated in steps of 0.05 between 0 and 1, and at each step the diagnostic test with the least loss value is chosen. The costs mentioned at the top of each graph are assigned, in order, as the costs of measuring the corresponding OutputKPI. We worked with the single fault assumption (i.e. we inserted only one fault in the system) and the procedure stops once the faulty component is identified with a probability of half the second least healthy component.

3 Applications

This section describes applications of the methodology on case studies. Each case study serves one or more purposes, which are highlighted in the introduction of the case.

3.1 CD-radio player

A CD-radio player is selected as toy example for experimenting with the methodology. The system is inspired on a semi-portable CD player that also has a radio receiver to play radio.

The system can thus play audio received by either the CD reader, or the radio receiver. Only one source can be selected by switching a mechanical switch. The system is powered by connecting a cable to an AC power socket, or by a set of batteries. The CD-radio player consists of four main modules

1. The power system, which provides power to the other modules. The main power comes from either an AC wall socket or inserted batteries.
2. The CD reader module, which reads audio from the CD and transforms it into a digital audio signal.
3. The radio receiver, which transforms radio waves to a digital audio stream.
4. The audio system, which transforms an incoming digital audio stream into sound waves.

The CD-radio player is modelled according to the Arcadia methodology (Voirin, 2017) as a Capella model. On the highest level, two main functions are fulfilled by the system, being *PlayCD* and *PlayRadio*. To achieve this, the four subsystems as described above are included. These four subsystems as well as their main functions are shown in Figure 3.1. For each of these subsystems' main functions, a functional breakdown can be made. These functional breakdowns are shown in Figure 3.2. Also, a physical breakdown of the system in different modules or components is created, which is shown in Figure 3.3. The deployment of the functions on the components is shown in Figure 3.4.

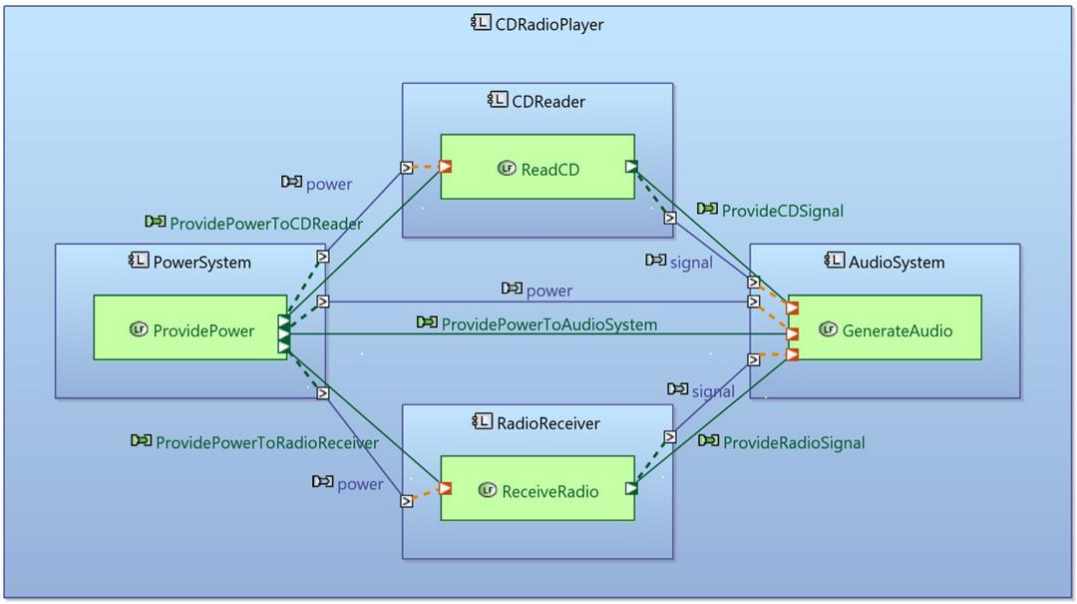


Figure 3.1: High-level functional flow of the CD-radio player. Functions are shown in green boxes and their deployment on components is represented by the blue box in which the functions are located. For example, *ProvidePower* is a function deployed on the *PowerSystem*.

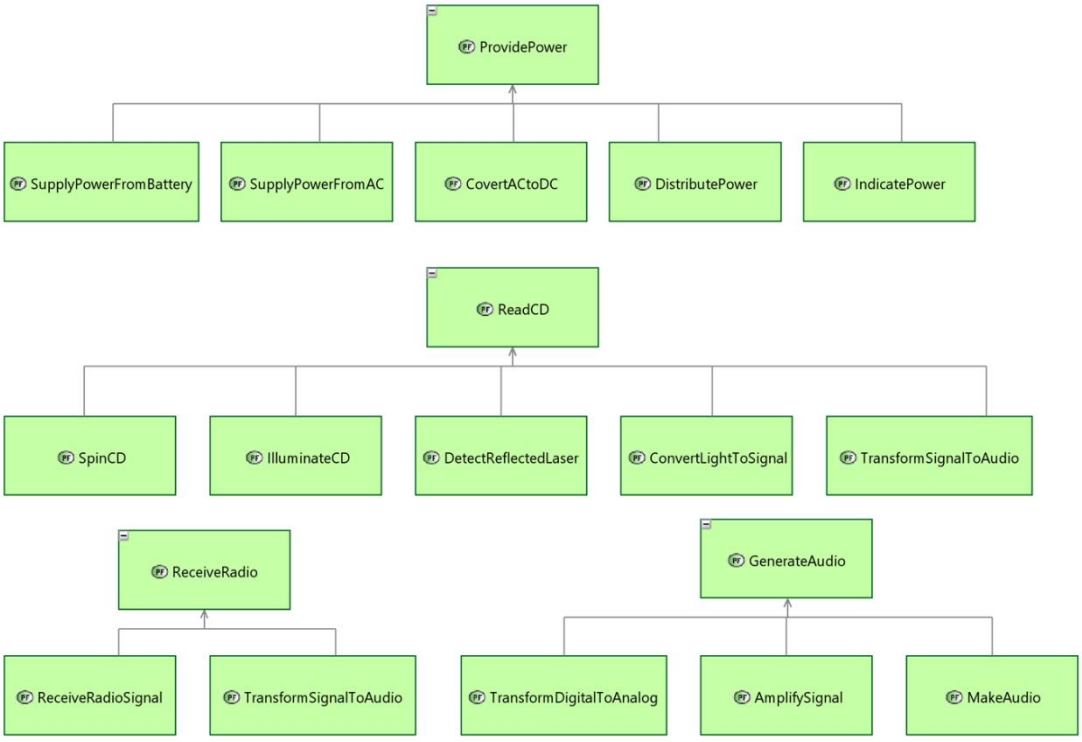


Figure 3.2: Functional breakdowns of the main functions of the four subsystems: *ProvidePower*, *ReadCD*, *ReceiveRadio* and *GenerateAudio*.

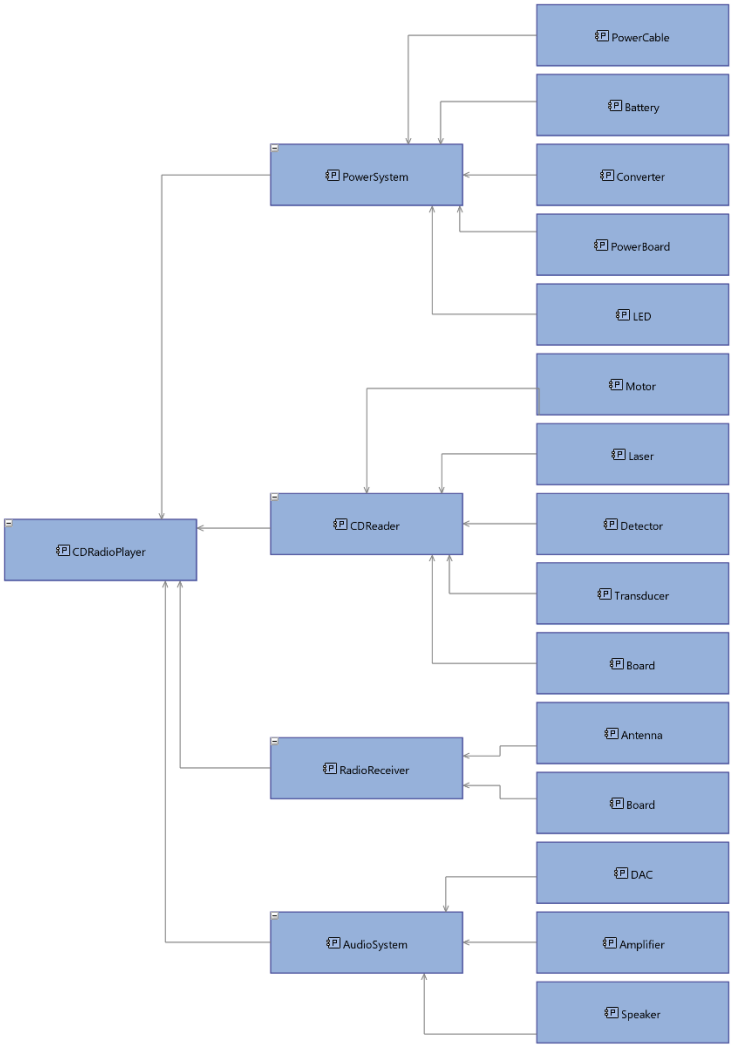


Figure 3.3: Physical breakdown of the system, down to the lowest level for which the Capella model of the CD-radio player was created.

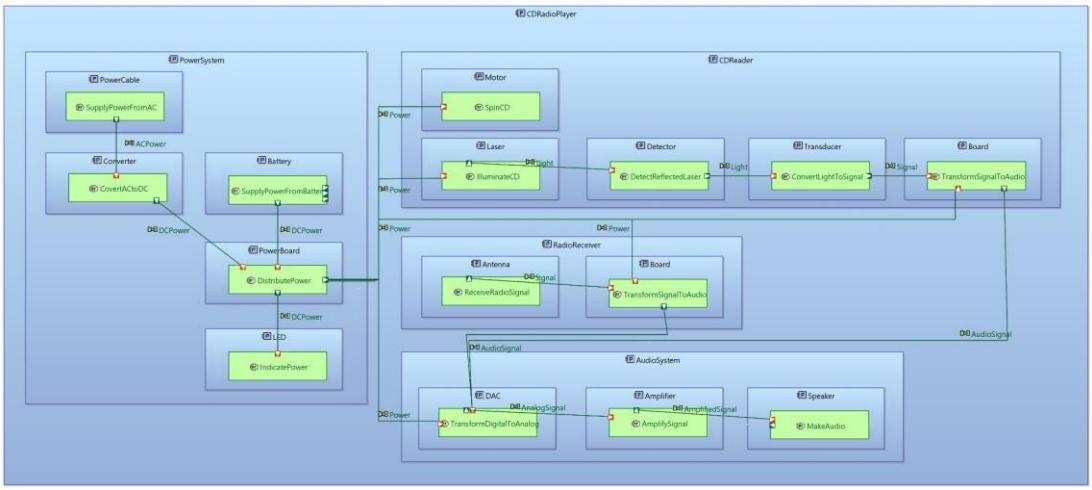


Figure 3.4: The physical architecture includes the deployed functions and the functional exchanges between all functions.

Finally, functional chains can be specified in Capella to indicate specific uses of series of functional exchanges, tying together different functions needed to fulfil a specific task. For every task, one may create a specific functional chain. Figure 3.5 shows four functional chains representing different uses, though partly overlapping, of the CD-radio player. The following functional chains are created:

- Power provision from AC power.
- Power provision from batteries.
- Functional exchanges for playing a CD, without power provision.
- Functional exchanges for playing radio, without power provision.

The rationale for adding these four functional chains is that in order to have a functioning CD-radio player, one of the two power provisioning chains and one of the two functional exchanges regarding information flow should be selected. One of the first group and one of the second group together to form a good combined functional chain for fulfilling one of the two main functions of the system.

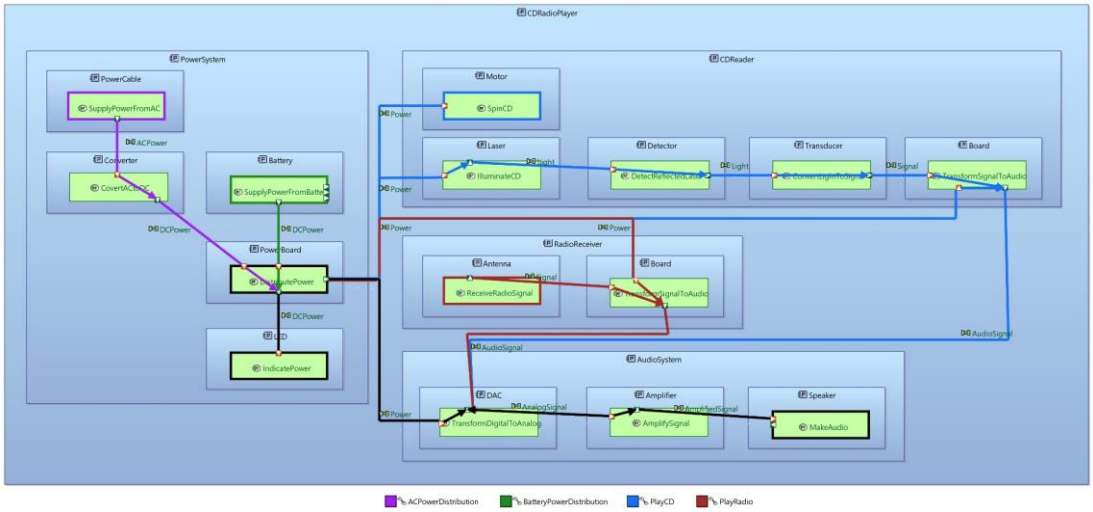


Figure 3.5: Physical architecture showing also four functional chains: power provision via either battery or AC socket, and the information exchange to play either CD or radio.

Note that the CD-radio player is just an example system used for illustration purposes. There are many more ways in which a similar system can be split up into subsystems and in which the functional breakdown can be done.

3.1.1 Functional dependencies

Based on the specification of the CD-radio player in Capella, one can put the same information in the Excel template as introduced in Section 2.4. To do this, the functions and components are transformed into functions and hardware pieces in the template. The component breakdown shown in Figure 3.3 serves as a basis for the clusters to be created: the main cluster is called ‘CDRadioPlayer’. This main cluster contains four clusters: ‘PowerSystem’, ‘CDReader’, ‘RadioReceiver’, and ‘AudioSystem’.

			Functions						Observables			
			CDRadioPlayer		PowerSystem	CDReader	RadioReceiver	AudioSystem	Direct	Direct	DiagnosticTest	
Cluster	Type	Name	PlayCD	PlayRadio	ProvidePower	ReadCD	ReceiveRadio	GenerateAudio	CD_Error	Radio_Error	ListenRadio	ListenCD
PowerSystem	Function	ProvidePower										
PowerSystem	Hardware	PowerSystem										
CDReader	Function	ReadCD										
CDReader	Hardware	CDReader										
RadioReceiver	Function	ReceiveRadio										
RadioReceiver	Hardware	RadioReceiver										
AudioSystem	Function	GenerateAudio										
AudioSystem	Hardware	AudioSystem										
Cost		Time									5	5

Figure 3.6: Specification of the CD-radio player in the SD2Act Excel template.

The main system specification sheet 'CDRadioPlayer' consists the two main functions as well as the main functions of each of the contained clusters. Figure 3.6 shows the Excel table as it is manually defined based on the Capella model. Note that this table contains the abstraction of what is shown in Figure 3.5. Since the observables are not specified in the Capella model, these were added based on a less formal description of the system. While this first step is a manual one, the Capella model itself is formally specified and would allow for potential automatic model-to-model transformation to ensure future scalability of the method. For this, the formal semantics of, for example, a functional chain need to be specified properly, as this is now subject to interpretation from the modeler. The following transformations are now applied to the Capella model to come to the Excel-based models:

- Every component in the component breakdown, except for the top-level one, is transformed into a hardware components. The top-level component is transformed into the main cluster.
- If the component has subcomponents according to the component breakdown, a cluster will be created for the component. The hardware node created will be put inside the cluster, if applicable. For example, 'PowerSystem' has subcomponents, thus will be a cluster. However, it will also be a hardware type that is put inside the cluster. Hence, there is a 'PowerSystem' hardware piece inside the 'PowerSystem' cluster.
- Every function is transformed into a function in the Excel template. The function will be put in the same cluster as where the component on which the function is deployed is. For example, 'ProvidePower' is a function that is deployed on the 'PowerSystem' component. Because the 'PowerSystem' component is in the 'PowerSystem' cluster, the function will also be put inside this cluster.
- Top-level main functions will be created as function nodes inside the top-level system.
- The functional deployment on components as specified in Capella determines which hardware-to-function relation is marked green in the Excel table. For example, the 'PowerSystem' component deploys the 'ProvidePower' function (according to Figure 3.1), thus the cell mapping the respective hardware to the function is marked green.
- The functional exchanges shown in the system architecture diagrams indicate the cells to be marked for function-to-function relations. For example, 'ProvidePower' provides its function to all three other main functions, as shown in Figure 3.1, thus gets those three cells marked in the table.
- The same transformations are applied on each level in the hierarchy, where the respective levels of granularity should also match the diagrams and levels in the breakdowns in Capella.

Each of the subsystems of the CD-radio player is also detailed out in the Excel template. An example of one of those subsystem specifications is shown in Figure 3.7.

			Functions				Observables	
			AudioSystem	DAC	Amplifier	Speaker	DiagnosticTest	
Cluster	Type	Name	GenerateAudio	TransformDigitalToAnalog	AmplifySignal	MakeAudio	Measure_Raw_Signal	Measure_Amplified_Signal
DAC	Function	TransformDigitalToAnalog						
DAC	Hardware	DAC						
Amplifier	Function	AmplifySignal						
Amplifier	Hardware	Amplifier						
Speaker	Function	MakeAudio						
Speaker	Hardware	Speaker						
Cost		Time					10	10

Figure 3.7: AudioSystem cluster specification in the SD2Act Excel template.

3.1.2 Diagnostic network

With the specification of the system completed in Excel, one can automatically create a diagnostic network and visualize it as a graph. The parser will ask for some interfaces (as shown earlier in Figure 2.13) to be specified, as the model combines different granularity levels. However, note that these questions are asked because the Excel template does not provide this information. In principle, this interface information is available in Capella at the lowest levels of granularity. If some model-to-model transformation will be developed in the future, this might eliminate the need for asking these questions completely. The resulting model is shown in Figure 3.8.

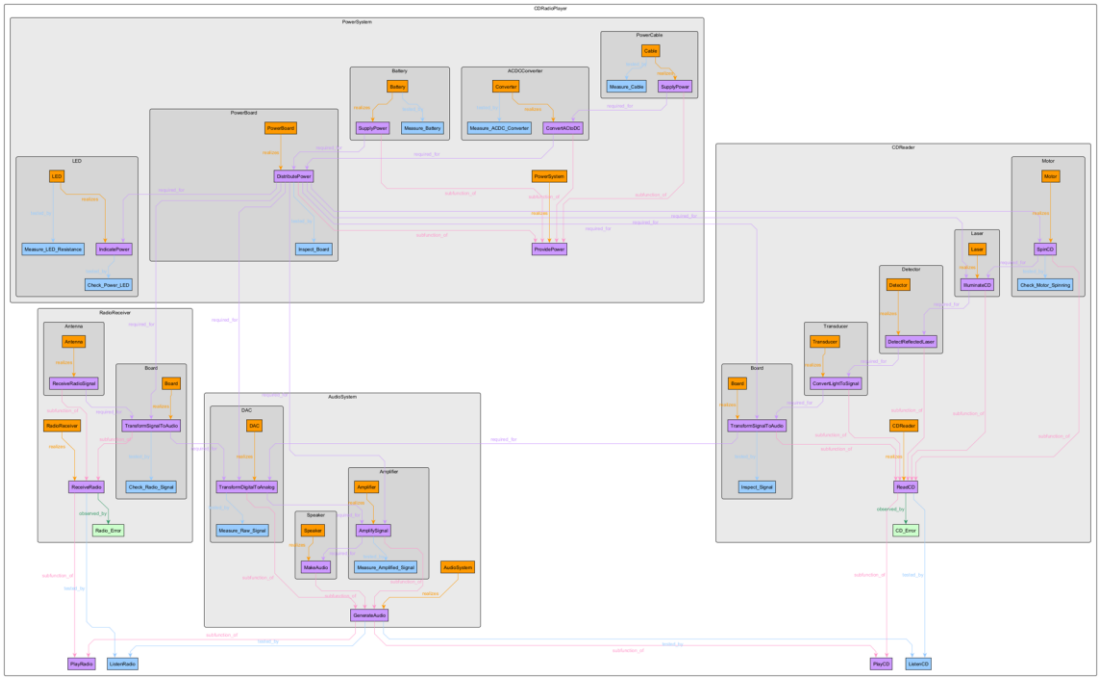


Figure 3.8: Visualization of the diagnostic network of the CD-radio player.

It is also possible to just load the top-level of the system specification to start out with a smaller diagnostic network. This is also very useful for visualization purposes. A visualization of the high-level model, without loading in all the details on the individual subsystems, is shown in Figure 3.9.

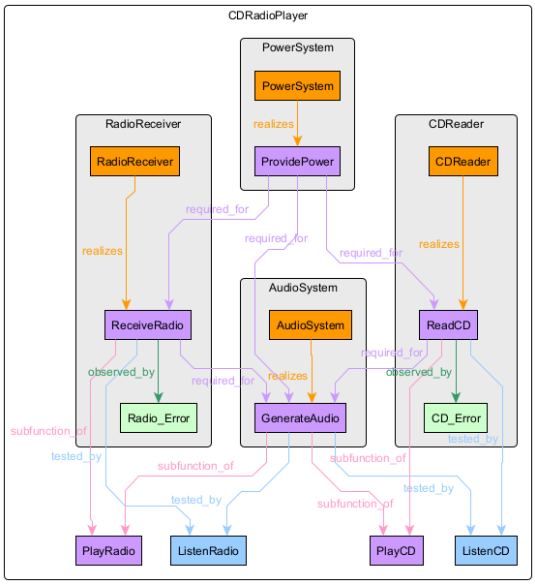


Figure 3.9: High-level visualization of the diagnostic network of the CD-radio player.

The diagnostic network can also be written to a Neo4j DB to enable querying and loading the network from there. A visualization of this model is shown in Figure 3.10.

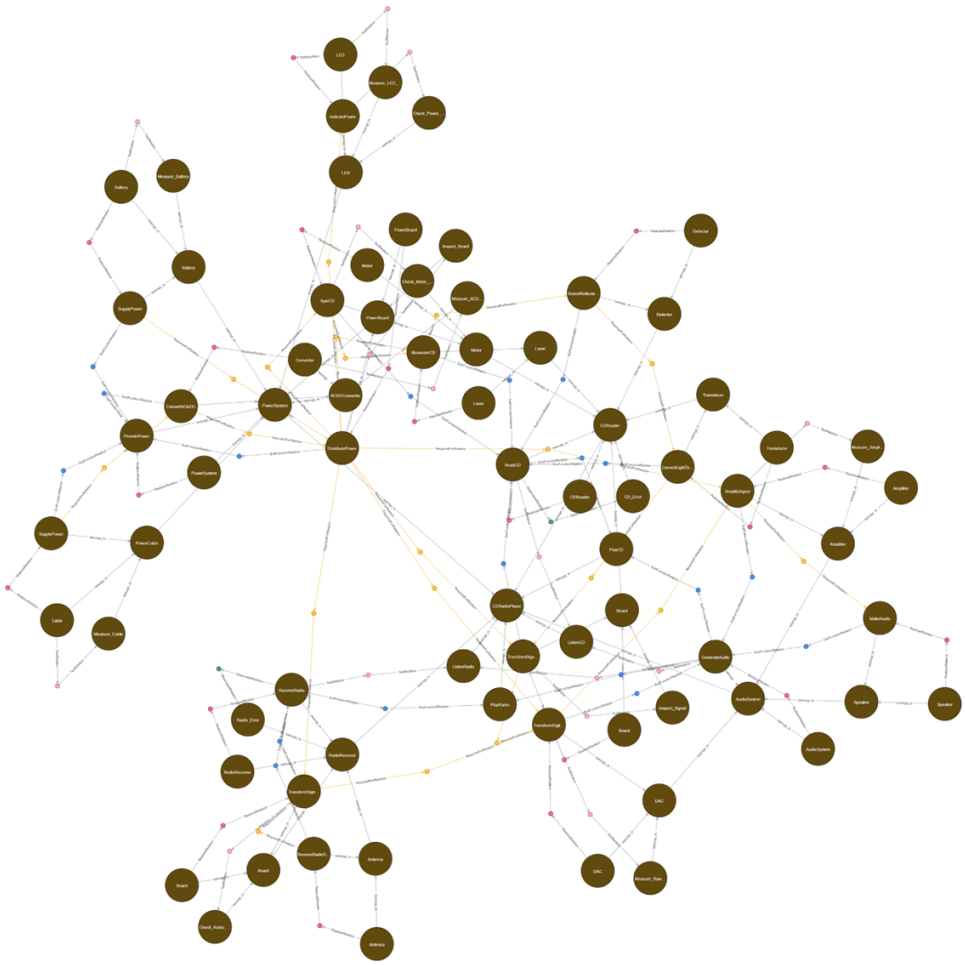


Figure 3.10: Visualization of the diagnostic network of the CD-radio player in Neo4j.

3.1.3 Reasoning model

The diagnostic network can be transformed automatically into a reasoning model, being either a Bayesian network or a Tensor network, as it does not contain any loops. The visualization of such a reasoning model is very similar to the pictures in the previous section.

However, the intended functionality of the CD-radio player cannot be captured accurately given the current modelling approach and extension of the methodology is needed to be capable of doing so. In the current approach, all parent function or hardware nodes of a given function node must be *Ok* or *Healthy*. The CD-radio player, however, has a physical switch that selects either the audio to come from the CD or from the radio. Also, there are two complementary power sources of which only one of them has to work in order to fulfil the system's functions. These kind of behaviours require the functional dependencies to be modelled as OR-gates rather than AND-gates, or may even require more extensive logical formulae to deal with the intended functional dependencies. Because of this limitation of the current methodology, the reasoning model that is created does not accurately reflect the behaviour of the CD-radio player and consequently cannot reliably be used for diagnostic reasoning.

To overcome this limitation, another type of relation needs to be added to the model to be able to cope with these more complex dependency relations in the system, as these are not uncommon in real-life systems. Possible solutions have been proposed, but none of them has been implemented yet. The plan is to address this limitation in 2024.

3.1.4 Diagnostic scenarios

Since the generated reasoning model currently does not reflect the intended behaviour of the CD-radio player, example diagnostic scenarios will be added to the report once the methodology can cope with more complex functional dependencies.

4 Future work

In this section we detail future work items that we envision for the proposed methodology. On top of items introduced in the above sections here we detailed out new items.

In the previous sections we identified the following items for future work:

- Automate the transformation of Capella models into diagnostic models according to the SD2Act methodology. [Sections 2.4 and 3.1]
- Improve the specification of the interface behaviour (functional) across different hierarchical levels. [Section 2.4.3]
- Optimize the computation of the conditional entropy for a large number of health nodes. [Section 2.6.2.1.3]
- Optimize the balance of cost and information gain for computation of best next service action. [Section 2.6.2.3]
- Specify functional behaviour via (complex) logic formulae, as an extension of the simple AND gates supported with the current methodology. [Sections 3.1.3 and 3.1.4]

The new items for future work introduced in this section are:

- Dynamic model creation, for scoping of the diagnostic root cause analysis for each given case.
- Design for diagnostics, for having at design time a functional overview of the diagnostic coverage of a given system, identification of diagnostic limitations and consequent improvement of the system design.
- Connection of Hardware and Functional diagnostic models, for having a unified diagnostic approach for the same system.

These items are described in more detail below.

4.1 Dynamic model creation

In Section 2.4.1 we presented the SD2Act approach to model system's functional dependencies for diagnostics. We showed how to create a diagnostic network, Section 2.4.3, and how to translate this network into a reasoning model, Section 2.5. In the proposed approach, the diagnostic network is hierarchical, while the reasoning model does not contain hierarchy, i.e. it is a static and flattened version of the diagnostic network. A large model is created up to the lowest level of hierarchy for all modules. While this may be a correct model, it is also sensitive for conflicting findings being inserted as evidence. An example for this is the situation in which some hardware components report problems in their KPIs, while the functions realized by these hardware components do not report any issues, due to for example redundancy. To overcome such issues, we identify two improvements to the current approach which should allow to dynamically create diagnostic reasoning models.

Firstly, investigate ways to achieve hierarchical diagnosis in such a way to allow a human-like reasoning that 'drills-down' through the system's functional hierarchy. In hierarchical

diagnosis the reasoning model is dynamic as it grows in size in the direction of the subsystem of interest. The direction of growth is guided by the evidence gathered from running diagnostic tests, see Figure 4.1 for a toy example. This approach will decrease uncertainty in the diagnosis: by excluding from the model (minor) issues reported by low hierarchical levels, in the figure these are all the orange triangles in the transparent boxes. These are not relevant for the diagnostic reasoning at the higher hierarchical level.

Furthermore, this will allow for an easier explanation of the diagnostic reasoning executed by the model, as it resembles the approach adopted by humans and the models are inherently smaller. In order to achieve this hierarchical diagnosis, we need an approach to determine which hierarchical level(s) to focus on during the diagnostic process. Similarly to the selection of the next diagnostic test, determining which level to focus on should be based both on expected added diagnostic information and cost of this action. First experiments with an algorithm by Nakakuki et al. (Nakakuki et al., 1992) seem worthy of further investigation (algorithm for measuring whether and which box to open).

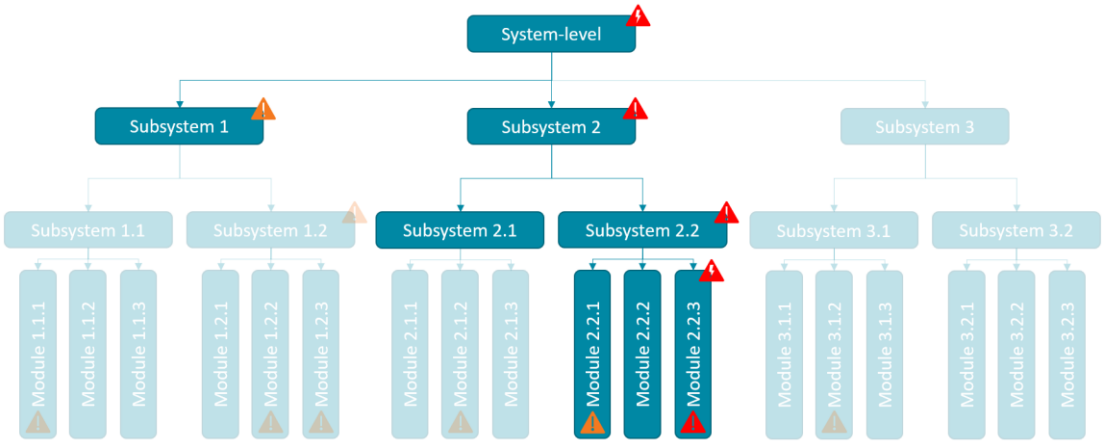


Figure 4.1: Hierarchical diagnosis. Transparent boxes are not detailed out in the hierarchical diagnosis, while they are considered in a flat reasoning model.

Secondly, investigate ways to dynamically create models 'traversing' the hierarchy of the system as described in the diagnostic network. This is would take hierarchical diagnosis a step further, as it will lead to reasoning models that are not directly derivable from the transformation of a diagnostic network at a given hierarchy to a reasoning model. The advantage of this approach is that the dynamically created reasoning models are even smaller than in hierarchical diagnosis and more scoped to the diagnostic problem at hand, to quickly drive the diagnostic process. We propose the architecture shown in Figure 4.2 to achieve this.

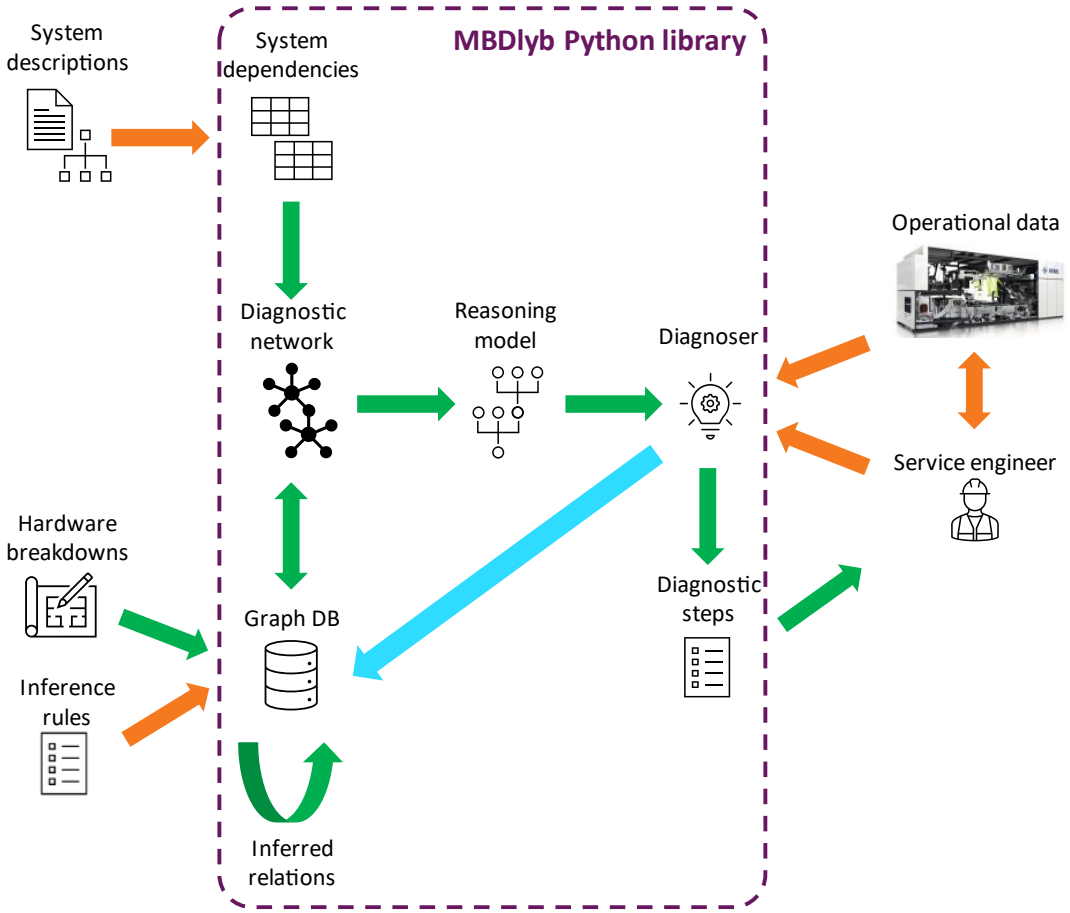


Figure 4.2: Architecture for the dynamic creation of reasoning models. In blue the difference with the current SD2Act architecture: a graph DB is used to store and query, based on diagnostic evidence, the diagnostic network.

The difference with the current SD2Act architecture of Figure 2.5 is the presence of a an edge between the Graph DB and the Diagnoser. The meaning of this edge is to query the DB to return the relevant subgraphs for the given diagnostic problem.

The central idea is that these queries, based on the evidence gathered by performing diagnostics tests, will return the most relevant diagnostic network for the given diagnostic problem. This diagnostic network will then be transformed into a reasoning model on which diagnostic inference can be performed, in such a way to have a reasoning model scoped to the diagnostic problem at hand.

4.2 Design for diagnostics

In Section 1.2.1 we detailed the relation of SD2Act with the ADiA project, the scope of the latter being design for diagnostics based on HW diagnostic models. Future investigations should allow for design for diagnostic for functional diagnostic models, as developed in SD2Act.

Such design for diagnostic analysis should compute which functions, at different hierarchical levels, cannot be performed as a consequence of HW failures. This information can then be used to spot observability limitations, i.e. different functions cannot be performed if the same HW is failing. Subsequently, to mitigate such observability gaps designers can decide to add additional tests to the system. Interestingly, these tests could require the attempt to execute a different function with the system, thus connecting to the work done with

reasoning with interventions in the Carefree project, see (van Gerwen et al., 2023) for more details.

4.3 Connection to HW models

The diagnostic models created with the SD2Act approach and those created with the CareFree and ADiA ones, see Section 1.2.1, contain the concept of HW components. In the diagnostic models these are represented as hidden random variables with discrete states, e.g. *Broken, Healthy,*. Despite the HW node being a shared type among these different models, there is a conceptual difference.

Specifically, in the functional models of SD2Act HW nodes determine the possibility of a function to be fulfilled or not; while in the physics based models of ADiA and CareFree HW nodes are related to the correct or incorrect propagation of physical quantities through the system. Therefore, further investigations are needed to fully understand if and how these models can be related to each other, or even integrated into a single model. Possibly, the HW nodes represent the linking pin of these two fundamentally different type of modelling.

5 Conclusions

The goal of the SD2Act project is to aid in failure analysis by computer-based diagnostic reasoning. First, system design information is transformed into a diagnostic network. This network can be seen as a graph with clearly defined types of nodes and types of relations between the nodes. The transformation can be done in two ways. Firstly, it can be done (fully) automatic if the design is documented in a (highly) structured manner, e.g. as part of a MBSE design approach. Secondly, it can be done manually, possibly assisted by tools, for example by using the Excel templated presented in this report. Next, a reasoning model can be generated from the diagnostic network. Bayesian Networks and Tensor Networks are evaluated in this report as potential reasoning formalisms on which the reasoning model can be based. Based on both system and human observations, the reasoning model iteratively suggests the next diagnostic test to execute to a service engineer, to efficiently come to the fault in the system.

5.1 Lessons learnt

Below we discuss the lesson learnt by developing the described methodology and applying it to the high-tech domain, like an ASML system in scope of the SD2Act project. We present the lessons learnt by splitting them up in a number of categories.

1. Transforming functional design information into a diagnostic model
    a. Functional breakdowns at the level of detail that is needed to create diagnostic models are hard to find. Most of the time, parts of relevant information need to be identified manually in a large set of unstructured or semi-structured documents and diagrams.
    b. Even though it is hard to find the necessary information, the information is available in the organization, documented or not. Once the information is found, a structured approach like the proposed Excel template proves to be a useful method to store this information.
    c. The main advantage of the approach based on an Excel template is its simplicity, as it was pointed out to those we presented the approach to. Most people would be able to grasp the concept and start discussing within minutes. Another advantage is the flexibility of not having to define the system to the fullest detail, as it allows for a relatively light-weight start of using the proposed methodology.

2. Recommending diagnostic tests and service actions
    a. It is possible to incorporate costs of performing diagnostic tests into the diagnostic model and combine it with the expected information gain to come to a recommended diagnostic test. Combining these two potentially brings an advantage in making decisions in the diagnostic process.
    b. Case studies show potential for selecting more cost-efficient paths by taking test costs into account compared to only considering the expected information gain when selecting a test. Comparing this to solved cases reveals that there is also potential for eliminating some of the conducted tests while using the model. It must be noted, however, that the model does not cover the full system and

may be biased because of that. More case studies need to be conducted to confirm it is really more cost-efficient.

c. Balancing cost and expected information gain throughout the diagnosis is not trivial. Depending on many non-technicalities, such as contractual obligations or agreements, severity of the escalation, warranty or duration of the escalation, a different balance between the properties may be desired. Optimizing this is not trivial and requires additional research. It may also be left unoptimized, in which the freedom of choice is left with the organization.

3. Inference of non-functional dependencies
    a. It is possible to derive undocumented dependency relations by applying physics laws onto design knowledge as inference rules on a knowledge graph. This approach requires rather specific design information to be stored in the knowledge graph for the inference to work.
    b. While the concept of deriving new dependencies from a knowledge base via rule-based querying is relatively straight-forward, it can be quite challenging to ensure the necessary information is in the knowledge base. In our case study, the notion of materials of hardware components, the material's sensitivity to temperature and the distances between hardware components must be known. Gathering or deriving all such information turned out to be a major endeavour. This can be solved by having all knowledge sources be connected (or connectable) to each other, for example by using extensive model-based approaches.

4. Reasoning formalism
    a. The tensor network formalism replaces the Bayesian networks as reasoning formalism. Tensor networks offer identical functionality and provide opportunities Bayesian networks do not. They can deal with cyclic networks, which will quickly be found at higher levels of abstraction of the proposed functional modelling.
    b. While tensor networks are somewhat slower than Bayesian networks for computing posterior probabilities, this reduction of computation speed does not seem to be a problem for the size of networks currently being built. Additionally, there are still some improvements that can be done on the tensor network algorithms to speed up computation.

5.2 Recommendations

1. As long as the required information for the diagnostic models is not captured in a single model, the information needs to be acquired from other documents and models. While there may be many (semi-)structured sources available from which one may extract bits and pieces of information needed for the diagnostic models, it turned out to be a challenging task to bring all information together. To facilitate this it is recommended to assign an identifier to every instance of every component and to ensure all references and mentions of this instance use the same identifier. This would greatly benefit the construction of a model even in the absence of a full-fledged model based engineering approach. Note that this identifier goes a step further compared to being able to identify spare parts, as the unique location of the part should also be included.

2. Extending upon recommendation 1, it would be good to use well-structured formats to store knowledge. This would benefit not only the creation of diagnostic models, but also

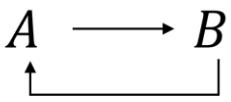
the capability of retrieving information from the knowledge base (traceability) and reusability of the information. While models are very much suited for this task, following up upon recommendation 1 together with using well-defined spreadsheets or databases may also improve upon the current situation.

5.3 Acknowledgements

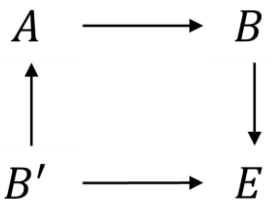
The research is carried out as part of the SD2Act program under the responsibility of TNO-ESI in cooperation with ASML. The research activities are supported by the Netherlands Ministry of Economic Affairs and Climate, and TKI-HTSM.

Appendix A

In Section 2.5.2.1.2 we described a procedure for computing inference in Bayesian networks with loops. We will prove now that simple tensor contraction in tensor networks with loops is equivalent to that procedure.



Consider the simple Bayesian Network above, and let $P_1(A|B)$ and $P_1(B|A)$ be the conditional probability tables defining A and B according to this causality structure. The joint probability distribution implied by this network, $P_1(A,B) = P_1(A|B) \cdot P_1(B|A)$ is not well defined according to standard rules of BN construction because it need not sum up to one, although one could normalize it. Moreover, belief propagation could not be used in this network. This joint unnormalized "probability distribution" is the same that one would obtain by translating this BN into a TN. And therefore, we can use all the TN tools introduced before.



Consider now the modified BN one obtains by applying the procedure of Section 2.5.2.1.2. where the CPT $P_2(B|A)$ is the same as before, i.e. $P_2(B|A) = P_1(B|A)$, the CPT for A in this network is the same as that for A in the previous network just swapping B for B', i.e. $P_2(A|B') = P_1(A|B)$, B' has uniform priors and E is the equality node whose CPT is $P_2(E = True|B, B' = b, b') = \delta_{b,b'}$ and $P_2(E = False|B, B' = b, b') = 1 - \delta_{b,b'}$.

This is now a well-formed Bayesian network whose joint probability distribution is:

$$P_2(A, B, B', E) = P_2(A|B') \cdot P_2(B|A) \cdot P_2(B')P_2(E|B, B').$$

Using Bayes rule after conditioning on $E = True$ we obtain:

$$P_2(A, B, B'|E = True) \propto P_2(E = True|A, B, B') \cdot P_2(A, B, B')$$

And marginalizing over B' yields:

$$P_2(A, B = a, b|E = True) \propto \sum_{b'} P_2(A = a|B' = b) \cdot P_2(B = b|A = a) \cdot \frac{1}{\dim(B')} \cdot \delta_{b,b'}$$
$$\propto P_2(A = a|B' = b) \cdot P_2(B = b|A = a) = P_1(A, B = a, b)$$

Extending this proof to loops with three or more variables is a straightforward exercise, all one needs to do is define a partition a network with a loop into two sets that cut across the loop and consider the Cartesian product of all variables in each set as a new variable, after which the argument above would apply.

References

Adén, S., & Stjernström, K. (2013). *Guidelines for modeling hydraulic components and model based diagnostics of hydraulic applications*.

Bañuls, M. C. (2023). Tensor network algorithms: A route map. *Annual Review of Condensed Matter Physics*, *14*, 173–191.

Barbini, L., Bratosin, C., & Nägele, T. (2021). Embedding Diagnosability of Complex Industrial Systems Into the Design Process Using a Model-Based Methodology. *PHM Society European Conference*, *6*(1), 9.

Deb, S., Pattipati, K. R., Raghavan, V., Shakeri, M., & Shrestha, R. (1995). Multi-Signal Flow Graphs: A Novel Approach for System Testability Analysis and Fault Diagnosis. *IEEE Aerospace and Electronic Systems Magazine*, *10*(5), 14–25. https://doi.org/10.1109/62.373993

Glasser, I., Pancotti, N., & Cirac, J. I. (2018). Supervised learning with generalized tensor networks. *ArXiv Preprint ArXiv:1806.05964*, 50.

Glasser, I., Sweke, R., Pancotti, N., Eisert, J., & Cirac, I. (2019). Expressive power of tensor-network factorizations for probabilistic modeling. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d Alché-Buc, E. Fox, & R. Garnett (Eds.), *Advances in Neural Information Processing Systems* (Vol. 32). Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2019/file/b86e8d03fe992d1b0e19656875ee557c-Paper.pdf

Hess, A., Stecki, J. S., & Rudov-Clark, S. D. (2008). The Maintenance Aware Design environment: Development of an Aerospace PHM Software Tool. *Proc. PHM08*, *16*, 17.

Koller, D., & Friedman, N. (2009). *Probabilistic graphical models: principles and techniques*. MIT press.

Lind, M. (2011). An introduction to multilevel flow modeling. *Nuclear Safety and Simulation*, *2*(1), 22–32.

Madiman, M., & Tetali, P. (2007). Sandwich bounds for joint entropy. *2007 IEEE International Symposium on Information Theory*, 511–515.

Nakakuki, Y., Koseki, Y., & Tanaka, M. (1992). Adaptive model-based diagnostic mechanism using a hierarchical model scheme. *Proceedings of the Tenth National Conference on Artificial Intelligence*, 564–569.

Rams, M. M., Mohseni, M., Eppens, D., Jałowiecki, K., & Gardas, B. (2021). Approximate optimization, sampling, and spin-glass droplet discovery with tensor networks. *Physical Review E*, *104*(2), 25308.

Robeva, E., & Seigal, A. (2019). Duality of graphical models and tensor networks. *Information and Inference*, *8*(2), 273–288. https://doi.org/10.1093/imaiai/iay009

Roques, P. (2017). Systems Architecture Modeling with the Arcadia Method: A Practical Guide to Capella. In *Systems Architecture Modeling with the Arcadia Method: A Practical Guide to Capella*. https://doi.org/10.1016/C2016-0-00854-9

van Gerwen, E. (2024). *Guided root cause analysis of machine failures - Status 2023*. https://repository.tno.nl/SingleDoc?docId=58178

van Gerwen, E., Barbini, L., & Borth, M. (2023). Differential Diagnosis with Active Testing. *PHM Society Asia-Pacific Conference*, *4*(1).

van Gerwen, E., Barbini, L., & Nägele, T. (2022). Integrating System Failure Diagnostics Into Model-based System Engineering. *INSIGHT*, *25*(4), 51–57.

Voirin, J. L. (2017). Model-based system and architecture engineering with the Arcadia method. In *Model-based System and Architecture Engineering with the Arcadia Method*. https://doi.org/10.1016/c2016-0-00862-8

ICT, Strategy & Policy

High Tech Campus 25
5656 AE Eindhoven
www.tno.nl

TNO innovation for life