Model-Based System Engineering for Diagnostics

Using a system model to diagnose hardware faults

TNO Public ⟩ TNO 2025 R13089
2 March 2026

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

TNO 2025 R13089 – 2 March 2026

Model-Based System Engineering for Diagnostics

Using a system model to diagnose hardware faults

Author(s)	Pierre America, Thomas Nägele
Classification report	TNO Public
Title	TNO Public
Report text	TNO Public
Number of pages	49 (excl. front and back cover)
Number of appendices	0
Project name	CareFree 2025
Project number	060.62910/01.01

Acknowledgements
The research is carried out as part of the CareFree project under the responsibility of TNO-ESI with Canon Production Printing as the carrying industrial partner. The research activities are co-funded by TKI HTSM via the PPP Innovation Scheme (PPP-I) for public-private partnerships.

Summary

One of the goals of the CareFree 2025 project has been the development of a methodology that connects Model-Based Systems Engineering (MBSE) to diagnostics, which is the process of identifying faulty hardware components in case of a system failure. For this purpose, we started with a methodology developed earlier in de SD2Act project, and applied it to a large and complex module in a realistic industrial system: a high-volume printer developed by Canon Production Printing. We identified various opportunities for improvement, found appropriate solutions, and accordingly enhanced the methodology and its supporting tooling, the open-source library MBDlyb. The resulting methodology takes as a starting point a system model expressed using a common MBSE tool, such as Capella. Such a model describes the component hierarchy in the system, the functions that the various components implement, and the dependencies between these functions. This system model can be extended with diagnostic information, such as observables, error messages, diagnostic tests, and prior hardware fault rates. MBDlyb can then import this extended system model, construct a probabilistic model for the system, and use that to generate diagnostic procedures at design time, indicating potential observability limitations of the design. To diagnose a troubled system, observations and error messages are entered into the tool, which then calculates which components are most likely to be faulty and which tests are most useful to do next.

By applying the methodology to a realistic system, we learned that it is possible to construct a sufficiently detailed system model, complete with diagnostic information, with a reasonable effort. The system model only needs to contain direct interactions among components and functions, because the more remote interactions are calculated by evaluating the probabilistic model. It is also relatively easy to adapt the model when the system design changes or when new insights are gained. In this sense, we conclude that our methodology is promising as a replacement for more traditional approaches, such as FMEA and FMECA, which require systematic manual analysis of text-based information, and are therefore laborious and error-prone.

Besides the conclusions of our research, this report also contains guidelines for system developers who want to apply our methodology.

Contents

Summary 3

Contents 4

1 Introduction 5
1.1 Research question 5
1.2 Main findings 6
1.3 Organization of this document 6

2 Background 7
2.1 State of the art 9
2.2 State of practice 9

3 Approach 11
3.1 Research method 11

4 Extended MBSE for diagnostics 14
4.1 Frequent model updates 14
4.2 Observables 15
4.3 Error reporting chains 17
4.4 Diagnostic tests - inspections 25
4.5 Diagnostic tests - singular 27
4.6 Diagnostic tests - composite 29
4.7 Control loops 36
4.8 Hardware priors 39
4.9 Complete composites 40

5 Observations 42
5.1 Modelling effort 42
5.2 Modelling tools 43
5.3 Output 43

6 Conclusions 45
6.1 Answers to research question 45
6.2 Further work 46

References 47

1 Introduction

With the ever-increasing complexity of high-tech systems, it also becomes increasingly difficult to diagnose them upon failure. It is common practice to write a service manual to describe how to diagnose and maintain the system at hand, but this has become a demanding effort. Furthermore, due to the system complexity, ensuring proper observability of failures for the purpose of diagnosis has become a complex task. In a highly competitive business, time-to-market is often prioritized over full completion of the service-related deliverables, such as service manuals and diagnostic procedures.

Assessment of system failure observability and the description of diagnostic procedures is traditionally a manual effort that involves several brainstorms with domain experts. As a consequence, it is hard to ensure completeness and correctness of the procedures. Moreover, most of this process needs to be redone after redesign, as it is hard to reuse the information due to the loose or even absent coupling to other system design artifacts. Altogether, delivering proper diagnostics of a designed system is a costly, but necessary, activity.

The SD2Act project [1] introduced a methodology for model-based diagnostics that connects to Model-Based Systems Engineering (MBSE) approaches, such as the Arcadia method [2]. Once formalized, the methodology's model enables operational diagnostics, in which a service engineer is guided through the diagnostic process, as well as design for diagnostics [3], in which diagnostic observability gaps are identified to improve the (diagnostic) design of the system. While both use cases are supported, the construction of the model still is a challenge. This is caused by the type of knowledge captured in typical MBSE: many system functional dependencies, mappings and decompositions are present, but there is no notion of observability or testability when deployed in the field. Hence, a more structured approach for connecting the system architecting domain and the diagnostic domain is needed. This report describes an approach to connect these domains.

1.1 Research question

The work described in this report has been conducted as part of the CareFree 2025 project, a project that develops new methodologies to address, amongst other, the aforementioned challenges in the high-tech industry. The project is a collaborative effort between TNO-ESI and Canon Production Printing (CPP) and is structured in two parts: One focused on model-based system engineering (MBSE) and another one focused on core performance diagnostics. This report is about the MBSE part of the project, for which the following research question is formulated:

RQ1 How to leverage the (potential) interplay between functional and structural description-based methodologies for hardware (HW) diagnosis?

The main goal to be achieved by answering this research question is to investigate whether the modelling approaches could be practically used for the diagnostic design of a system. For this investigation, the research question can be broken down into the following sub-questions:

RQ1.1 How does the aforementioned function-based diagnostic methodology scale to component-level?

RQ1.2 How to ensure proper integration of the diagnostic-relevant (observability) information in the MBSE models?

This document reports about the efforts and methodology extensions realized to answer these questions. [4] elaborates on the second part of the project that focuses on performance diagnostics.

1.2 Main findings

In the MBSE-focused part of the CareFree project, we focussed on the 'How', i.e., RQ1.2. Therefore, a large part of this report is devoted to improvements and extensions to the methodology, together with guidelines extracted from practice. With these improvements and extensions we found that it is possible to build a model of a complex system based on structure and functions, and to transform that model into a probabilistic graphical model. The model was used to obtain realistic output from the latter model, in the form of (simulated) diagnostic procedures and support for design for diagnostics. Although the methodology was aimed at starting from an *existing* structural and functional model, it even appears worthwhile to construct such a model from scratch, as a replacement for a laborious Failure Mode, Effects and Criticality Analysis (FMECA) [5] procedure.

1.3 Organization of this document

The remainder of this report is structured as follows. First, Chapter 2 introduces some background information on the function-based methodology, state-of-the-art and industry practice. Then, Chapter 3 describes the research approach used to extend the methodology and answer the research questions by means of practical use cases. Chapter 4 provides the methodology extensions and guidelines on how to model common concepts, after which Chapter 5 discusses several further observations that we made during our activities. Chapter 6 concludes the report by reflecting on the research questions and discussing future work.

2 Background

Research on diagnostics at TNO-ESI as well as the resulting methodologies are structured in three main use cases, as described in [6]. These use cases, as well as their relations are shown in Figure 2.1. Central in all use cases is the diagnostic model, which formalizes all knowledge and learnings related to diagnostics for a specific system.

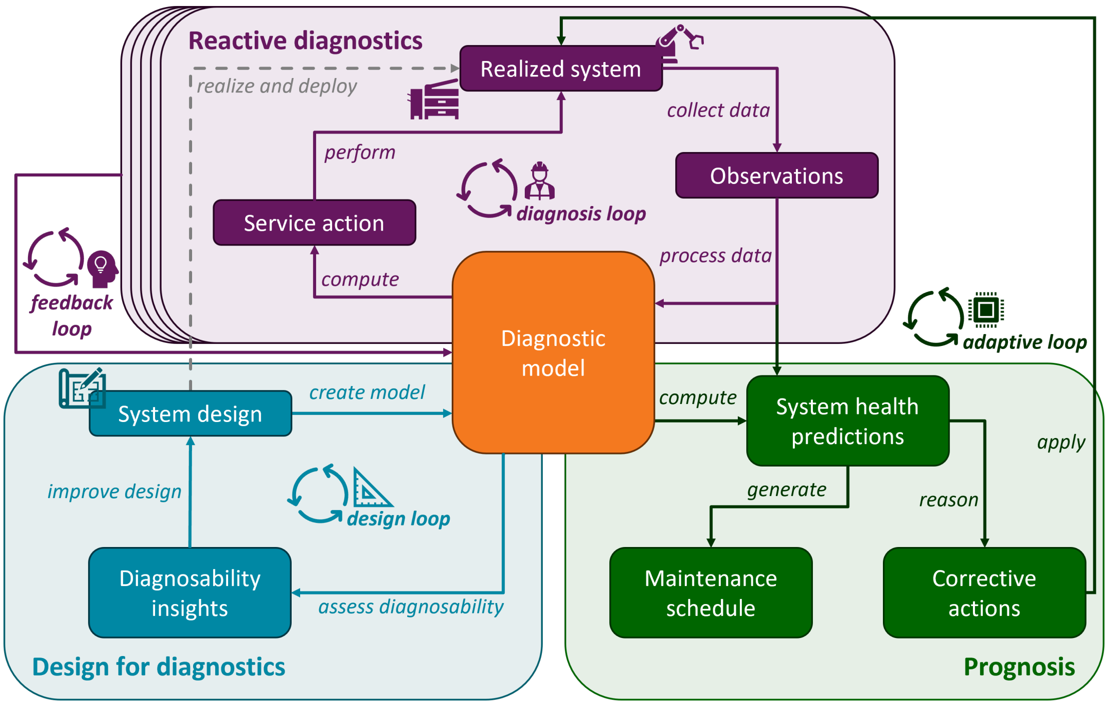


Figure 2.1: Three main diagnostic use cases in scope at TNO-ESI. More information about the use cases can be found in [6].

The use cases can be explained as follows:

- **Design for diagnostics** (D4D) is about assessing the diagnosability of a system design and designing the system in such a way to make it diagnosable after deployment in the field. D4D assists in design space exploration for sensor placement and the specification of diagnostic inspections.
- **Reactive diagnostics** is a form of operational diagnostics, focussed on assisting a service engineer in the field to diagnose a troubled system. It is about interpreting observations – both automatic and manual – to explain what has happened to the system to find a root cause for the problem at hand.
- **Prognosis** is about continuously monitoring field data and predicting the future health of the system. These predictions could be used to schedule maintenance proactively, or to (automatically) apply corrective actions to the system to prolong its lifetime without requiring maintenance.

In the SD2Act project [1], the function-based diagnostic methodology was developed. The methodology focusses on the diagnosis of equipment failures causing a hard-down situation of the system. High-level system architecture information, such as system functional decomposition, structural decomposition, functional deployment on hardware and inter-functional relations, forms the backbone of the diagnostic model. Since all of this information is available in typical MBSE models, the methodology connects well to already existing MBSE methodologies. Figure 2.2 depicts an overview of the methodology.

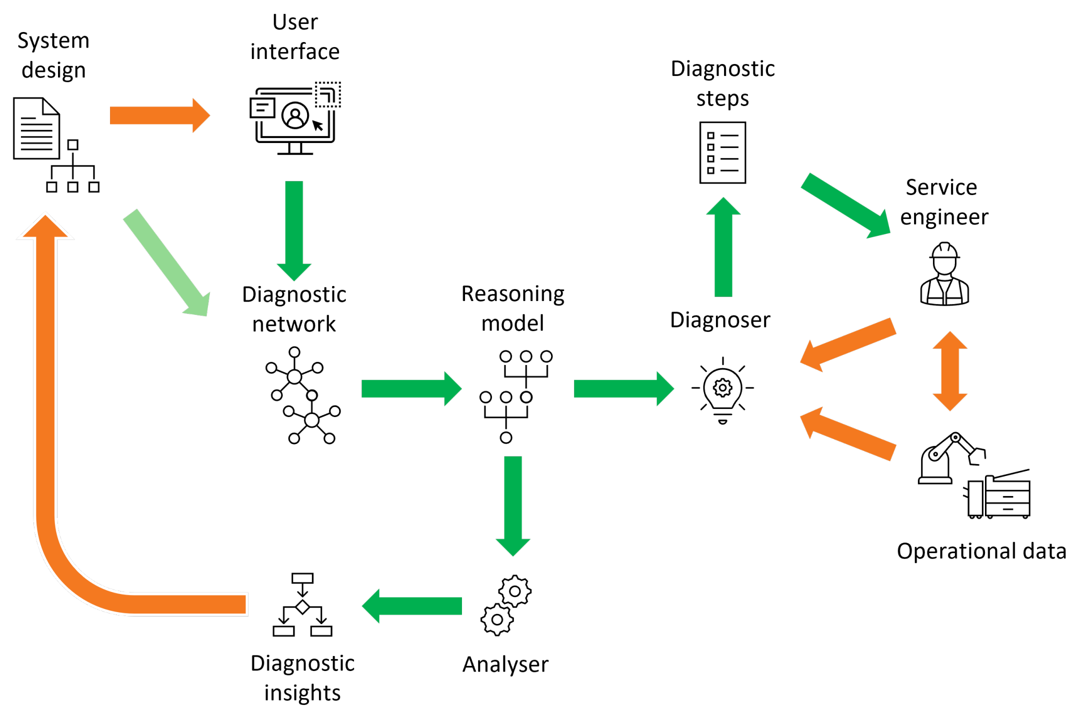


Figure 2.2: Overview of the function-based diagnostic methodology. Green edges represent fully automatic transformations, while orange edges are manual. The light-green edge is automatic, but results in a partial model that needs to be completed manually.

A short description for each of the elements is given below. For more details we refer to [1].

- **System design** represents all knowledge – both structured and unstructured – available about the system. As such, MBSE models are also included in this knowledge, as are field learnings and tacit knowledge.
- The **user interface** is a web-based front-end to create and modify the diagnostic network, and allows for importing MBSE models as diagnostic models.
- The **diagnostic network** is knowledge base in which all diagnostic-relevant information for the methodology is stored. It is a stored as a knowledge graph in a graph database, following a defined ontology. When referring to the *diagnostic model*, this is what is meant.
- The (diagnostic) **reasoning model** is a formal, mathematical model that allows for diagnostic reasoning. The methodology transforms the diagnostic network to a probabilistic graphical model, such as a Bayesian network or Markov random field, to reason based on system observations and to compute diagnoses.

- The **analyser** implements the D4D algorithms to assess the diagnosability of the designed system and to generate the initial diagnostic procedures. The **diagnostic insights** derived from these analyses can be used to improve the **system design** by adding or moving sensors, or by designing additional diagnostic tests.
- Reactive diagnosis is implemented by the **diagnoser**. Here, operational data from the machine is injected in the reasoning model and algorithms are used to compute a diagnosis. The diagnoser also produces a list of **diagnostic steps** – usually tests – that can be conducted by the **service engineer** to acquire more information from the system to proceed in the diagnostic process.

This methodology is the starting point to answer the questions posed in Section 1.1. Chapters 3 and 4 further describe how the methodology was applied and extended.

2.1 State of the art

Stemming from the late 80s [7], model-based diagnostics (MBD) [8] is by no means a new concept. In this field, formal models are created to provide assistance for troubleshooting issues as they could arise in the field, what we refer to as operational diagnostics. Both scalability [9] and the uncertain nature of the systems [10] to diagnose included probability in diagnostic reasoning. To ensure diagnosability of a designed system after deployment, methods were developed to compute diagnosability and compare sensor placement configurations for making diagnostics-informed design decisions [11, 12, 13].

An effort to connect different techniques for risk assessment, such as Failure Modes and Effects Analysis (FMEA) [14] and Fault Tree Analysis (FTA) [15], solidified in the Risk Analysis and Assessment Modeling Language (RAAML) [16, 17]. RAAML is compatible with SysML [18, 19], and as such connects well to model-based systems engineering (MBSE) methods to deal with the complexity and scale of modern-day high-tech systems. With the development of the succeeding SysML v2 standard [20, 21], efforts are made to adapt the concepts of RAAML in it [22].

The rise of (generative) AI has given an impulse in the diagnostic domain as well, though most advancements to date lie in the medical field [23, 24]. Furthermore, AI has proven to be useful to detect anomalies in time series [25, 26], and to learn patterns for quick issue resolution [27]. Typical AI-based approaches are less suitable for addressing D4D, as these rely on the availability of logged system data to be trained. Approaches to generate synthetic data for training AI with simulation models [28] or surrogate models [29] could work, but require additional effort to be able to synthesize the sufficient accurate data.

2.2 State of practice

Many high-tech companies rely on FMEA-like processes for assessing the diagnosability of their systems. Such analyses are often centred around a spreadsheet that needs to be filled in, listing all the faults or failure modes and their observability. The assessment is then used to design new observables, such as error messages, and diagnostic procedures. A well-defined process dictates the participants of the brainstorm sessions to complete the spreadsheets, but there is little to no guarantee the results are correct or complete. The analysis is typically done when most of the design is fixed and there is very little room for incorporating design changes.

For operational diagnostics, many companies rely on some sort of failure detection and identification via pattern matching, linking certain combinations of observations to known causes. Troubleshooting experts should then follow a predefined manual, or rely on their experience to identify the root cause. Also, many companies create several dashboards in which system data is aggregated and presented in a useful way to be used to diagnose a specific type of issue. It takes years before the organization has built up sufficient experience to represent a broad range of failures, and it requires structured sharing of experiences and knowledge to efficiently make use of the collective knowledge.

Tools like QSI's TEAMS [30] and PHM Technology's MADe [31] have been developed and commercialized to facilitate model-based (design for) diagnosis. These tools require an engineer to build models dedicated for diagnostics, which is in practice hard to justify in an industrial context. As such, they are mostly used for safety- or mission-critical systems, such as aerospace and military, where the diagnosability of the system needs to be quantified and proven up to a certain level. Besides commercial offerings, some companies have developed in-house solutions for assisted diagnostics.

3 Approach

Since our research question (see Section 1.1), and more specifically subquestion RQ1.2, is about leveraging our methodology at industrial scale, it was necessary to do our work in an industrial context and with a real-life system in mind. We found a realistic case from our industrial partner, Canon Production Printing, that we could validate the methodology with. This involved a large production printer under development, with a special focus on one complex module that involves various mechanical, electrical, aerodynamic, and thermal subsystems. We strove to model our focus module completely from the top level down to the level of field-replaceable parts (since that was the desired level of diagnostics). Other modules where covered in much less detail, covering only the components and functions needed for diagnosing the focus module (e.g., power supply and error communication).

The diagnostic methodology that we wish to apply (Figure 2.2) is supported by an open-source Python library called MBDlyb [32], which originated from an earlier project [1]. MBDlyb offers a web-based user interface that allows importing or constructing a system model and augmenting it with diagnostic information. Then its diagnoser page can simulate the various steps in a diagnostic procedure, by accepting observable information about a system and the results of diagnostic tests, where after each step it reports which system components are most probable to be broken, and it suggests diagnostic tests to perform next. It also provides a page to support design for diagnostics, where diagnostic procedures are summarized and gaps in observability can be identified. Because of this close connection to our methodology, we considered MBDlyb as a core artefact in our research: We applied it in practice and strove to improve and extend it to embody any new insights.

As an MBSE tool we used Capella [33], for the simple reason that it is readily available and a Capella input module for MBDlyb had already been developed earlier [34]. If a suitable MBSE model had already been available for our industrial system in a different tool or formalism, we would have considered developing an additional input module, but that was not the case.

3.1 Research method

To answer our research question, as mentioned in Section 1.1, we followed these steps:

1. Apply the method to an industrial system.
   This means the following activities:
   - Gather information about the design of the system
   - Build a model or modify an existing model in Capella
   - Import the Capella model into MBDlyb
   - Observe the output of MBDlyb, for example by running several scenarios using the diagnoser and considering the ranking of components that may be broken and the tests that are proposed
   - Compare this output of MBDlyb with researchers' and experts' expectations

   These activities were executed by ESI researchers together with the industrial partner.

2. Find a pattern that offers an opportunity for improvement.
   This could be any of the following:

   - The modelling effort is larger than expected
   - The modelling work is very repetitive
   - The Capella model is not a good representation of system design
   - The effort of importing the model into MBDlyb is too large or repetitive
   - The operation of MBDlyb presents unjustified difficulties
   - The output from MBDlyb differs from what it should be
   - The performance of MBDlyb is too slow

   Again, these activities were executed by ESI researchers together with the industrial partner.

3. Check for general applicability.
   This step is typically performed by ESI researchers discussing whether the pattern is likely to occur in other domains as well. If that is the case, it is considered worthwhile to adapt the methodology to better deal with the pattern, since the resulting solution can then be reused in other domains. If not, it appears to be a situation specific to our industrial partner, or even the individual system that we modelled, and in that case we would spend no further effort to optimize the methodology for this case.

4. Manually create the desired probabilistic model that could serve as a reasoning model (see Figure 2.2).
   This step is done whenever the MBDlyb output is not as desired. We work backwards from the desired output towards adapting the input and the transformation tools. Typically, this step involves ESI researchers creating, for example, Bayesian networks in a tool such as Netica [35]. The responses of the Bayesian network can be evaluated, and the network can be adapted until it accurately reflects the underlying mechanism that we want to model.

5. Reproduce the desired probabilistic model as a reasoning model in MBDlyb.
   Working backwards again from the desired probabilistic network, we (typically ESI researchers) can work to create, using the MBDlyb user interface, a diagnostic network that translates (see Figure 2.2) into a reasoning model that has such a probabilistic network. In several cases, MBDlyb can already support the desired network, otherwise Step 6 applies. After creating a suitable model in MBDlyb, we can check MBDlyb's output, but we can also export the resulting Bayesian network and inspect whether it has indeed the desired structure.

6. If necessary, extend MBDlyb.
   In some cases, the desired probabilistic network cannot be generated in MBDlyb as it is. Then it may be necessary to extend or otherwise modify MBDlyb. This is a task for ESI researchers or developers, which often involved extending or modifying MBDlyb's internal data model that describes the structure of the data it stores in its graph database. After MBDlyb has been updated, we can create a diagnostic network that results in the desired probabilistic network. Again, we can check this by observing the output from MBDlyb or by exporting the Bayesian network and inspecting it separately.

7. Find a natural modelling pattern in the MBSE tool.
   The goal of this step is to decrease as much as possible the manual efforts required to create a good diagnostic model by means of the MBDlyb user interface — as shown in Figure 2.2 — and to maximize the diagnostic content creation in an integrated MBSE

tool, such as Capella. Now that we know what kind of MBDlyb model we want to use for the pattern under consideration, we (ESI researchers) also need to find a way to represent this in Capella. Ideally, we do this in a way that fits naturally in Capella and its associated method, Arcadia, because that would allow system modellers to create models that are easy to write, understand, and maintain. If necessary, we can use an existing plug-in for Capella, but we want to avoid building a new plug-in unless we are sure that this would be very valuable by itself. In some cases we may even decide that the model information we want to edit does not fit well in Capella so that it is better to store it separately.

8. If necessary, extend the MBDlyb input module.
   When we know how we want to represent the relevant information in a Capella model, we can check if MBDlyb can already import that information for further use. If not, the MBDlyb input module for Capella must be updated, which is again a task for ESI researchers or programmers.

9. Apply the updated method to the industrial system and check the outcome.
   This means that we repeat Step 1 again.

10. Repeat.
    This is an iterative approach, which means that steps 1 to 8 are repeated over and over. Of course, it is a good idea to check now and then whether the whole approach is adequate for achieving our goals.

4 Extended MBSE for diagnostics

In this chapter, we discuss the extensions to the methodology we developed. This discussion will be based on patterns. With a pattern, we mean a phenomenon that we observe repeatedly in the design and modelling of real-life systems. This could be a phenomenon related to the structure of a model, but it could also be related to the development process. If such a pattern occurs frequently and across multiple systems and domains, we want to support in the model-based diagnostics methodology.

4.1 Frequent model updates

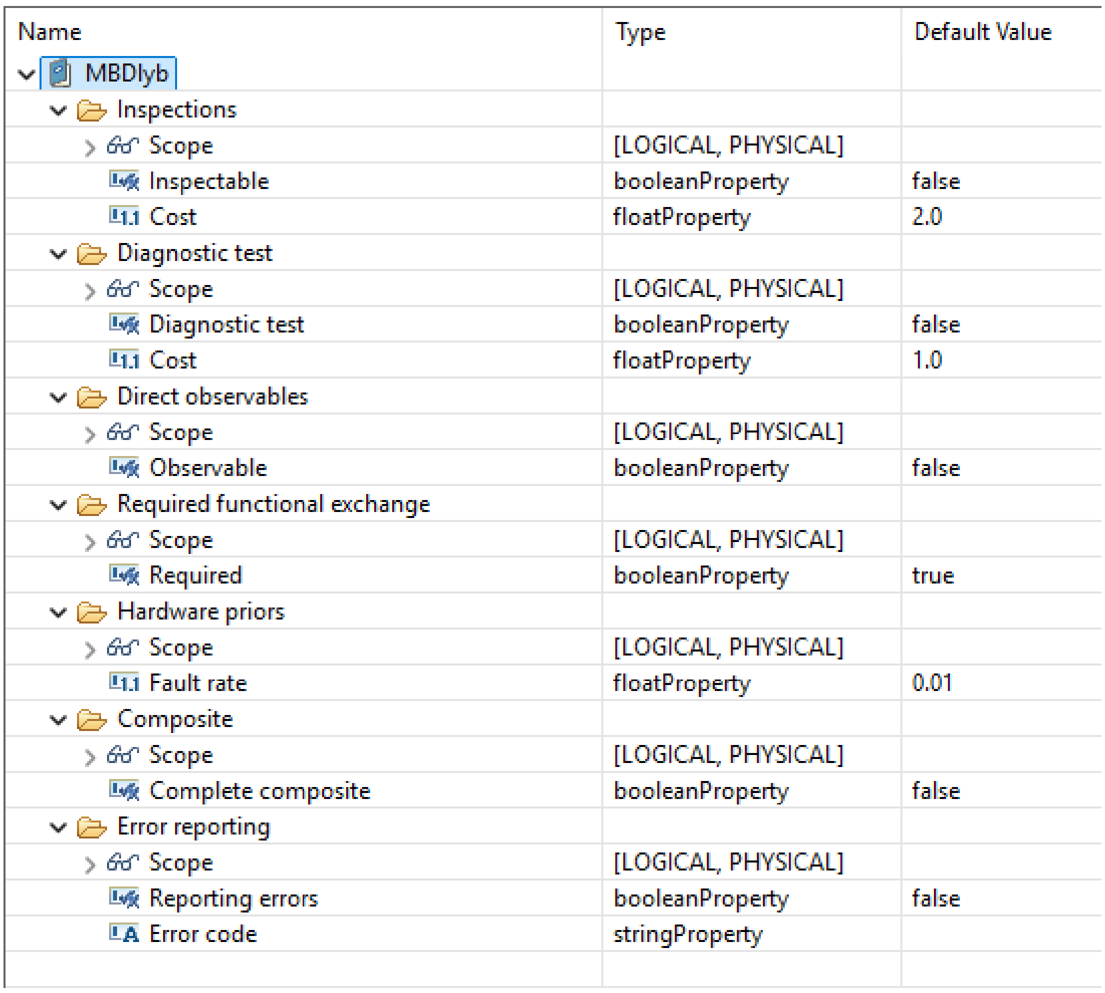
The original approach assumed that there is a relatively stable MBSE system model available for components and functions, represented in Capella, for example. Then this model could be imported into MBDlyb and the additional information needed for diagnostics, such as observables and tests, would be added using the web interface. After that the tool's functionality could be applied, including the diagnoser and the design for diagnostics overview.

However, in practice we observe that the underlying structural and functional model is constantly being updated during development. This is not just a consequence of our approach of developing a model as a research vehicle, but we also observe frequent changes in actual industrial development artefacts, such as mechanical and electrical CAD models. In modern industrial development it is normal that designs at all levels are regularly updated to incorporate new insights, either from the ongoing development process or from a changing environment.

Now this would mean that we would have to import the updated structural and functional model very frequently, and every time we would have to manually add the diagnostic information again. This is clearly undesirable and for a realistically-sized model even infeasible. Therefore, we decided to adapt our methodology.

The solution that we adopted was the inclusion of *all* information in the Capella model, not just the structural and functional information, but also the diagnostic info. Typically, this diagnostic information does not fit in a standard Capella model, and therefore we chose to represent this information using the Capella plug-in PVMT (Property Values Management Tool) [36]. This allows us to add extra annotations to various kinds of Capella elements, which capture the necessary diagnostic info. Figure 4.1 shows a screenshot of the PVMT editor showing the properties that we defined for this purpose. How to use these PVMT annotations to represent diagnostic information is discussed in the following subsections.

When importing a Capella model, MBDlyb offers the options to add default observables and diagnostic tests automatically, but since there is no guarantee that these observables and test are the same as the ones available in a real system, we leave these options deselected. Rather we want to add the relevant diagnostic information in the Capella model. This approach allows us to edit all the relevant information in a single modelling tool – Capella – ensuring consistency of the model and enabling importing it into MBDlyb. After importing, no



Name	Type	Default Value
MBDlyb		
Inspections		
Scope	[LOGICAL, PHYSICAL]	
Inspectable	booleanProperty	false
Cost	floatProperty	2.0
Diagnostic test		
Scope	[LOGICAL, PHYSICAL]	
Diagnostic test	booleanProperty	false
Cost	floatProperty	1.0
Direct observables		
Scope	[LOGICAL, PHYSICAL]	
Observable	booleanProperty	false
Required functional exchange		
Scope	[LOGICAL, PHYSICAL]	
Required	booleanProperty	true
Hardware priors		
Scope	[LOGICAL, PHYSICAL]	
Fault rate	floatProperty	0.01
Composite		
Scope	[LOGICAL, PHYSICAL]	
Complete composite	booleanProperty	false
Error reporting		
Scope	[LOGICAL, PHYSICAL]	
Reporting errors	booleanProperty	false
Error code	stringProperty	

Figure 4.1: PVMT editor showing the various properties that can be used to add diagnostic information to elements of a Capella model

more additions and adjustments are needed.

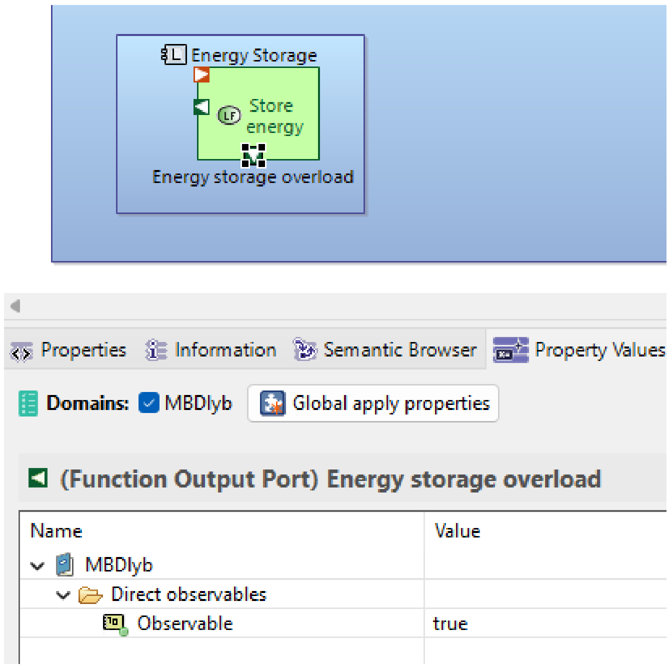
4.2 Observables

Typically, diagnostic activity around a system starts with observing that something went wrong. The basic form of such an observation is when the correct operation of a function is directly observable in the system. This can be indicated in Capella as follows:

- Add an output port to the function and give it a meaningful name
- Set the property `Observable` of the output port to true in PVMT

For each function only one such observable output is meaningful, since it only flags whether the function works correctly.

Figure 4.2 shows how this can be done in Capella and Figure 4.3 shows what the result looks like after import into MBDlyb. Although all this information is collected in Capella, you typically need to look at multiple diagrams and tables to see how everything is connected, so the MBDlyb diagram can be useful to see these connections in a single view.



(Function Output Port) Energy storage overload

Name	Value
MBDlyb	
Direct observables	
Observable	true

Figure 4.2: In Capella, a diagram (top) shows function *Store energy* of component *Energy Storage* and its output port *Energy storage overload*. In PVMT (bottom) we see that the output port is labelled as observable

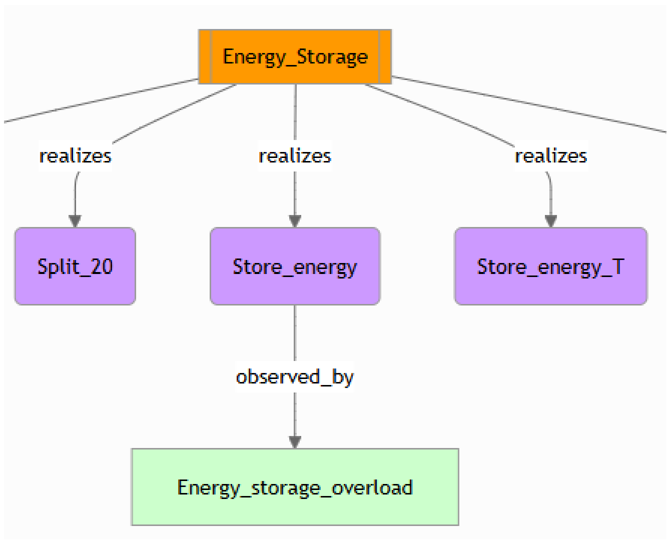


Figure 4.3: After import from Capella, MBDlyb shows that component *Energy_Storage* realizes function *Store_energy* and that the correct operation of this function can be observed by observable *Energy_storage_overload*

Guidelines

As we will see in Section 4.3, MBDlyb offers specific support for extensive error reporting chains. Therefore, this mechanism for observables is meant for the simpler cases, such as:

- An obvious mechanical failure. Here are some examples from the automotive domain for familiarity:
  - A vehicle that does not move when it should
  - A steering wheel that cannot turn
  - A brake pedal that goes all the way down without slowing down the vehicle
- An indicator light on a power supply
- A leak in a container that spills liquid or gas where you can see it

4.3 Error reporting chains

In a large system, the component that detects an error is often not the same as the component that reports the error to a user or service engineer. Rather there is a chain of digital components that transmit the error message from the detecting component to the reporting one. This also means that when we see an error message, this is not only bad news: The good news is that the components in the error reporting chain are actually working. This is because the probability is extremely low that a *digital* reporting chain reports an error incorrectly because of a *hardware* failure.

To model this in Capella, we distinguish between the detecting component and the reporting one. We can label a function output as detecting (yielding) an error as follows:

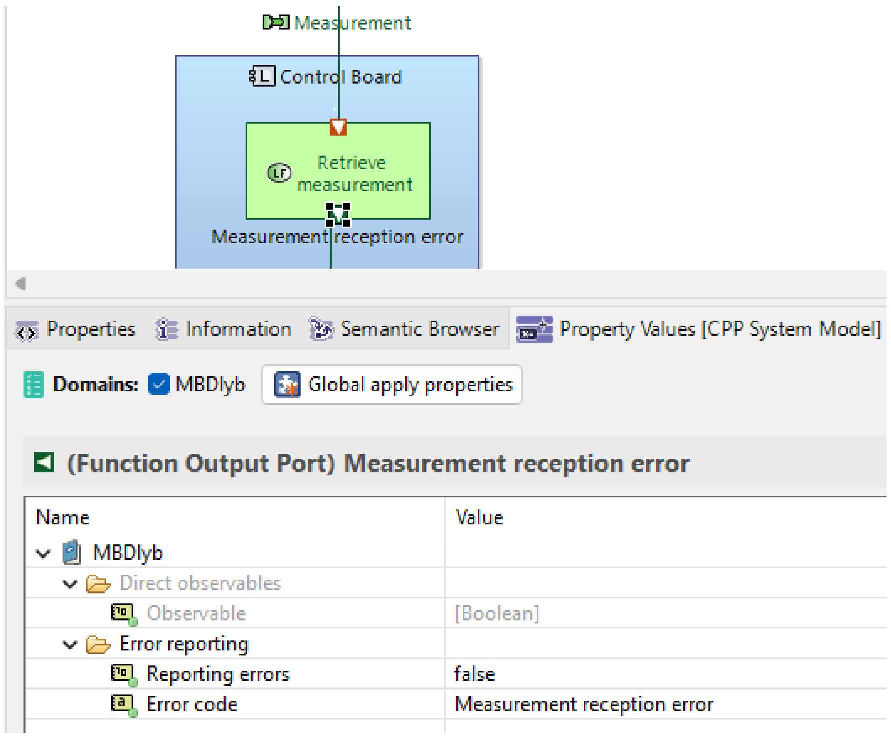
- Add an output port to the function, optionally giving it a meaningful name.
- Set the property `Error code` in the group `Error reporting` to the desired name for the error message. Note that the name of the output port does not matter, apart from the fact that this can be made visible in a diagram.
- The property `Reporting errors` can be left to the default value, false (unless the same function reports the error to the user, in which case it would be easier to label it as a direct observable).

Figure 4.4 shows the way to do this in Capella.

After importing into MBDlyb, the error is shown as a yellow box, as Figure 4.5 illustrates.

Now that we can indicate in Capella where an error is detected, it is also possible to label where it is reported:

- Make sure there is a path of functional exchanges between each function that yields an error to an error reporting function. We call this the error chain.
- Add an output port to the reporting function, optionally giving it a meaningful name.
- Set the property `Reporting errors` in the group `Error reporting` to true.
- The property `Error codes` can be left empty, in which case all incoming errors are reported here. Alternatively, it can contain a list of error codes.



(Function Output Port) Measurement reception error

Name	Value
MBDlyb	
Direct observables	
Observable	[Boolean]
Error reporting	
Reporting errors	false
Error code	Measurement reception error

Figure 4.4: In Capella, a diagram (top) shows function *Retrieve measurement* of component *Control Board* and its output port *Measurement reception error*. In PVMT (bottom), the output port is labelled as yielding but not reporting an error

This is shown in Figure 4.6.

After importing into MBDlyb, the error is again shown as a yellow box, with a *reports* arrow from the reporting function, as shown in Figure 4.7. In MBDlyb's diagnostic network, each error occurs only once, so when an MBDlyb diagram's scope includes both the yielding and the reporting function (which is rare for typical systems), they would be shown as linked to the same yellow box.

When this is set up properly, the MBDlyb diagnoser will show such errors in the list of direct observables, with result values *Absent* and *Present*. When *Present* is selected, the function that yields the error is considered as not working correctly, but the functions in the error reporting chain are marked as working correctly.

Guidelines

First, let us stress that this mechanism for modelling error reporting chains is meant only for digital reporting chains, where an error code is injected at the start and transmitted to the component where it can be reported. Moreover, it is only suitable for diagnostics on hardware failure. If the reporting chain would be analogue, e.g., realized by one cable per error, a shortcut in such a cable could lead to an incorrect report of the associated error. Also, a software error could potentially lead to an incorrect error report. Only for a digital chain in the context of hardware diagnostics can we be sure that an error being reported indicates that the reporting chain is working correctly.

Note that the function yielding the error is not part of the error reporting chain. Now we have two possibilities for connecting the error output port:

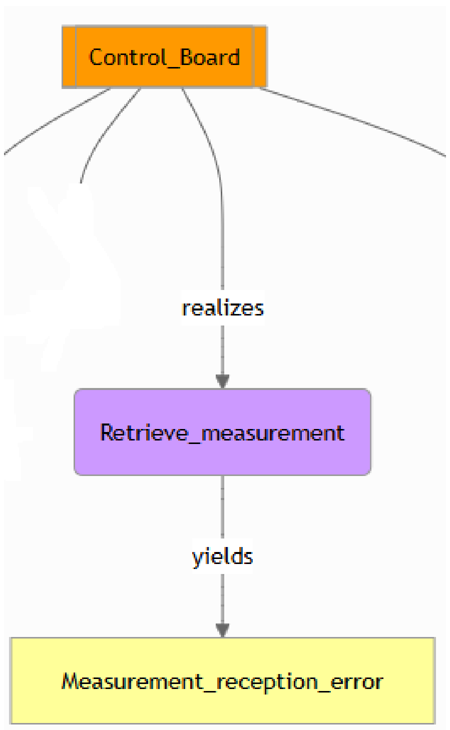


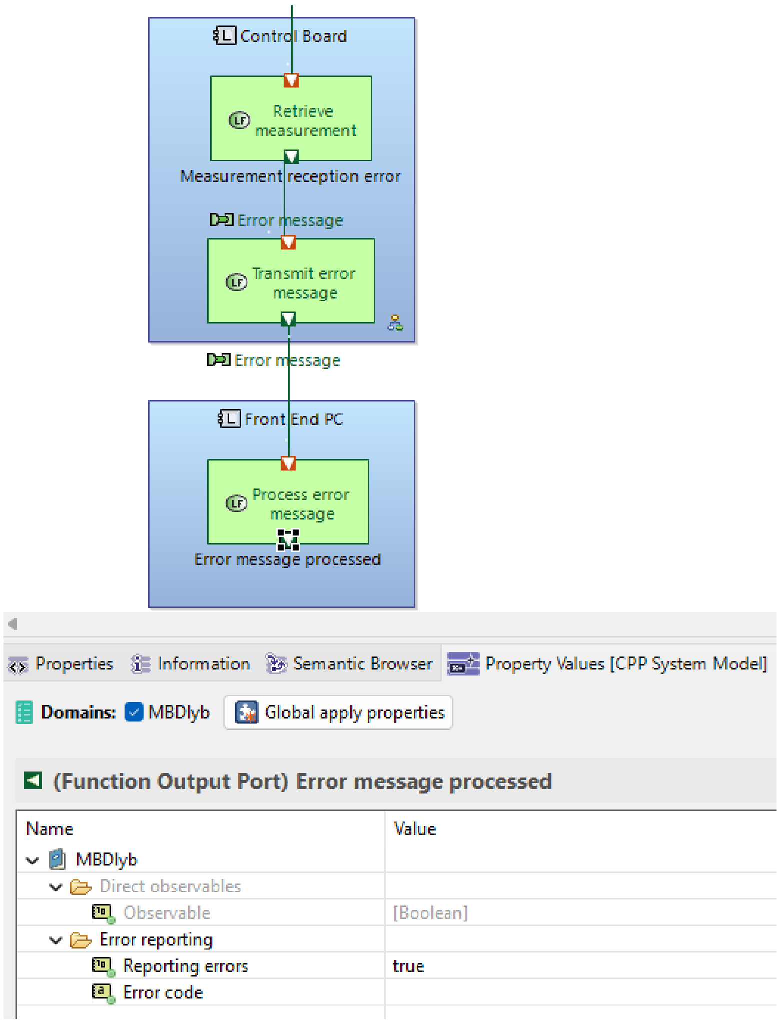
Figure 4.5: After import from Capella, MBDlyb shows that component *Control_Board* realizes function *Retrieve_measurement* and that this function yields error *Measurement_reception_error*

- The error is passed on to a transmitting function in the same component, as in Figure 4.6. Now when the error is reported, this component is considered to be healthy, because the transmitting function is working correctly. Instead, the error will be blamed on a function and a component *before* the yielding function. Therefore, we should use this pattern when the yielding function detects that there is an error, but is not involved in causing the error.
- The error is passed on to a function in a different component (not shown in a figure). In this case the yielding function is considered to be a possible suspect in causing the error. We should therefore use this pattern when this is indeed a possibility.

It is possible to use the same error reporting chain for several different error messages: There is no need to define a separate reporting chain for each message if in reality the same functions and components are involved. Rather, multiple errors can be injected into the same function at the beginning of the chain. An example of this is shown in Figure 4.8. MBDlyb has been designed in a way that such reuse of an error reporting chain does not cause a lot of duplicated nodes in the probabilistic network (see Figure 4.9), so this is an efficient use of human and computing resources.

It is also possible to inject errors at different points in the error reporting chain, as illustrated by Figure 4.10. Effectively this means that there are different error reporting chains, and MBDlyb will create a suitable set of nodes in the probabilistic network for each of them (see Figure 4.11).

Finally, we should mention it is possible to have different *ends* of an error reporting chain. In this case, it is necessary to list the names of the errors in the PVMT property `Error codes` of



(Function Output Port) Error message processed

Name	Value
MBDlyb	
Direct observables	
Observable	[Boolean]
Error reporting	
Reporting errors	true
Error code	

Figure 4.6: In Capella, we see in the diagram (top) that function *Process error message* of component *Front End PC* has an output port *Error message processed*. PVMT (bottom) shows that this function output port is labelled as reporting errors, and there is no restriction on the error codes that it can report. The diagram also illustrates that at least one of these errors is *Measurement reception error*, yielded by function *Retrieve measurement* in component *Control board* and transmitted by function *Transmit error message*

the respective reporting output ports. Of course, this mainly makes sense if the reporting of the various errors is done by different components.

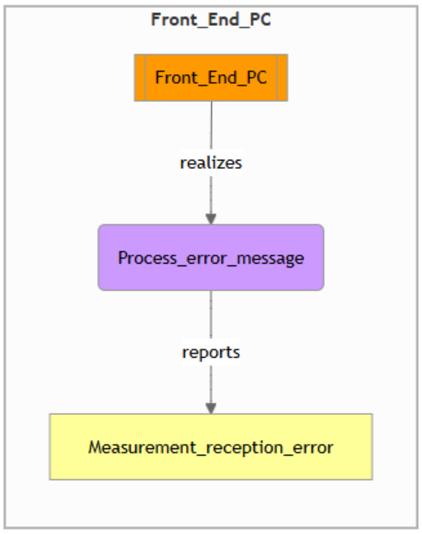


Figure 4.7: After import from Capella, we see in MBDlyb that function *Process_error_message* of component *Front_End_PC* reports the error *Measurement_reception_error*

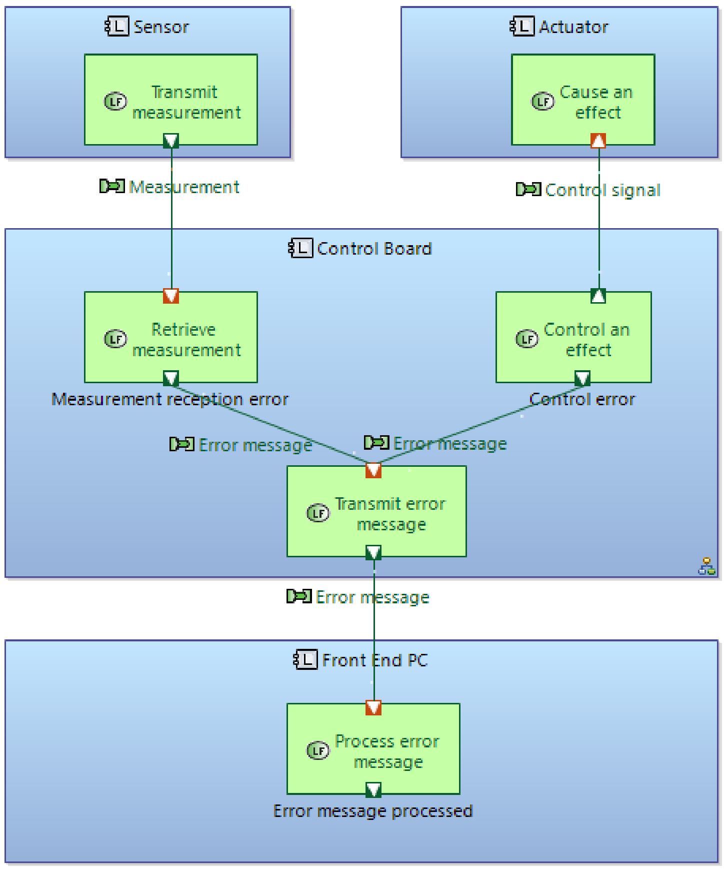


Figure 4.8: In this Capella diagram we see that both *Measurement reception error* and *Control error* are injected into the same reporting chain. This chain starts at function *Transmit error message* in component *Control Board* and ends at function *Process error message* of component *Front End PC*

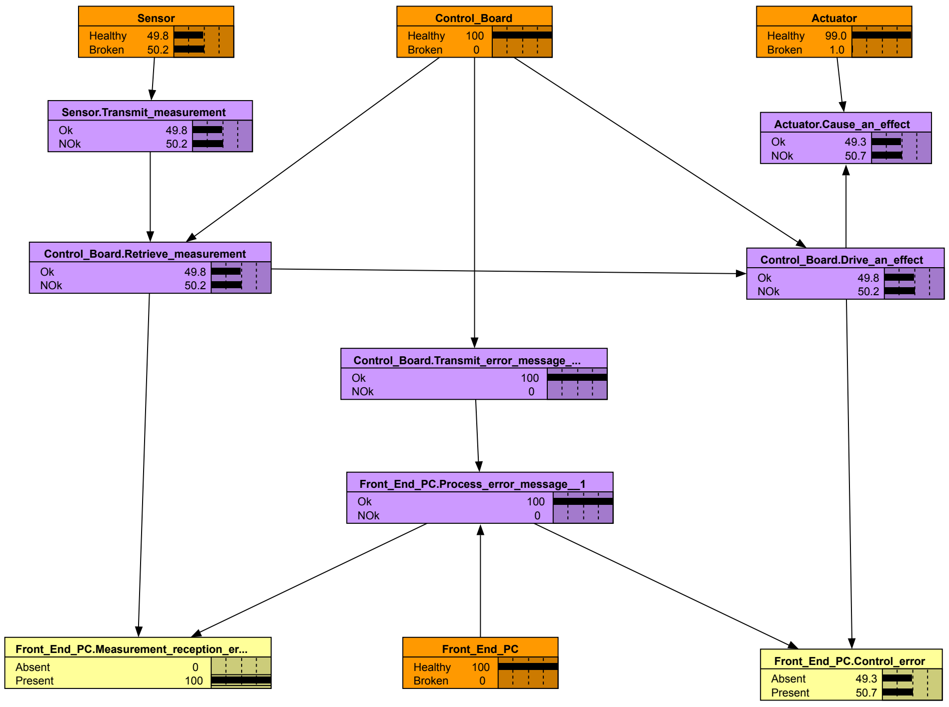


Figure 4.9: This screenshot from Netica shows the (slightly simplified) Bayesian network exported from MBDlyb after importing the Capella model shown in Figure 4.8. We see components in orange, functions in purple, and errors in yellow. Dependencies are indicated by arrows: each error depends on the function that originally yields it and on the reporting chain, consisting on the functions *Transmit_error_message* and *Process_error_message*. The conditional probability tables (not shown here) are such that an error can be present only if the reporting chain is working correctly and its probability to be present is 99.9% if the yielding function is not working. In the case shown, we have entered that the error *Measurement_reception_error* is *Present*, and evaluating the Bayesian network tells us that most components are healthy, except for *Sensor*, which is 50.2% probable to be broken, very different from 1%, the prior value.

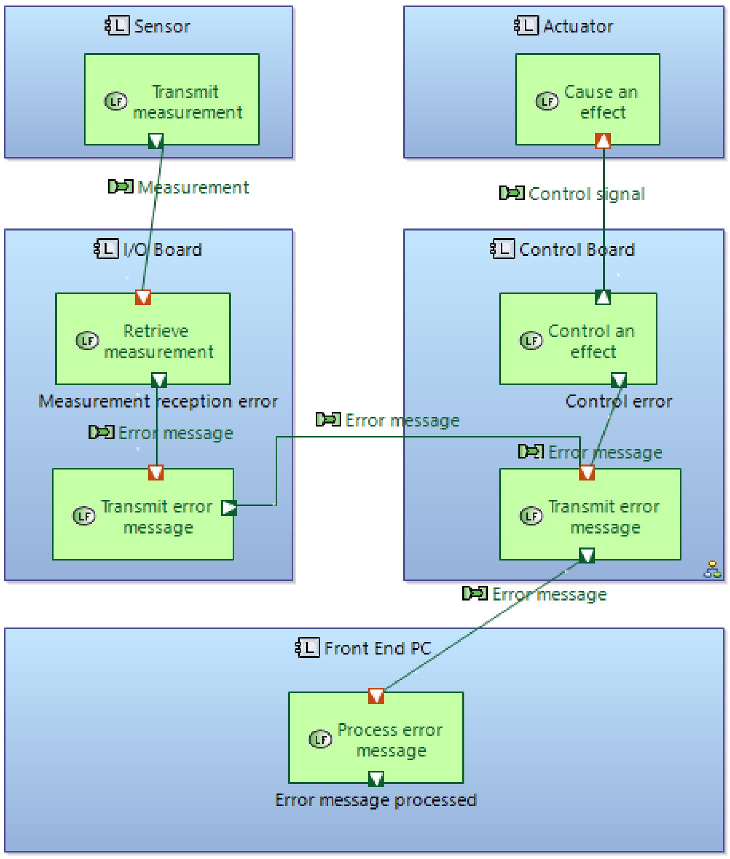


Figure 4.10: This Capella diagram shows a longer error reporting chain, starting from function *Transmit error message* of component *I/O Board* and ending at function *Process error message* of component *Front End PC*. The error *Measurement reception error* is injected at the beginning of the chain, but the error *Control error* is injected into a later point of the chain.

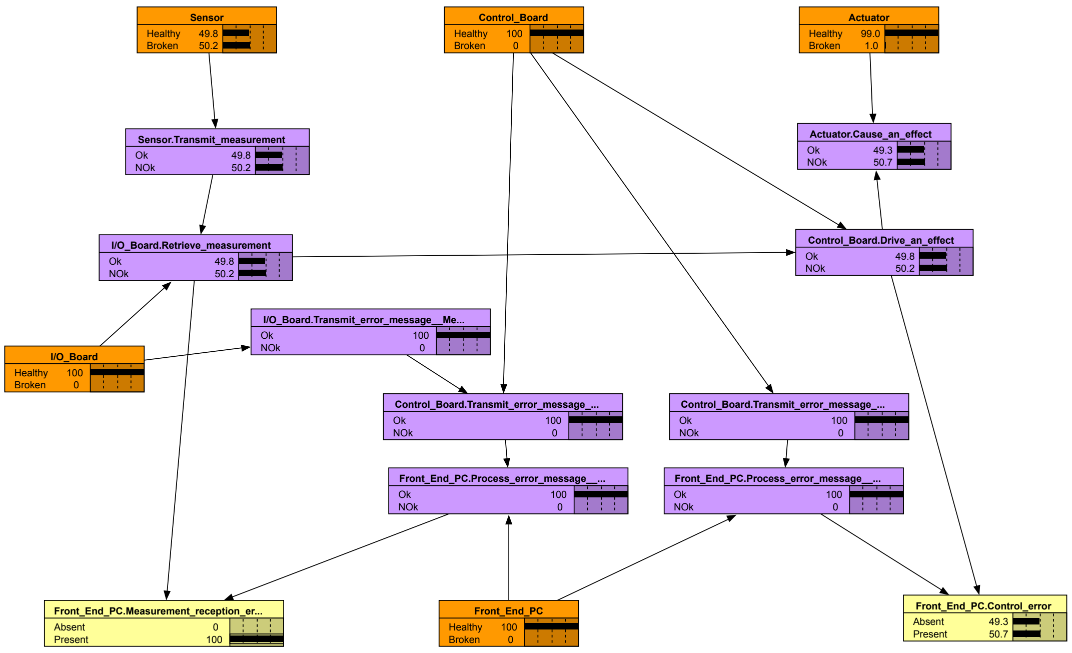
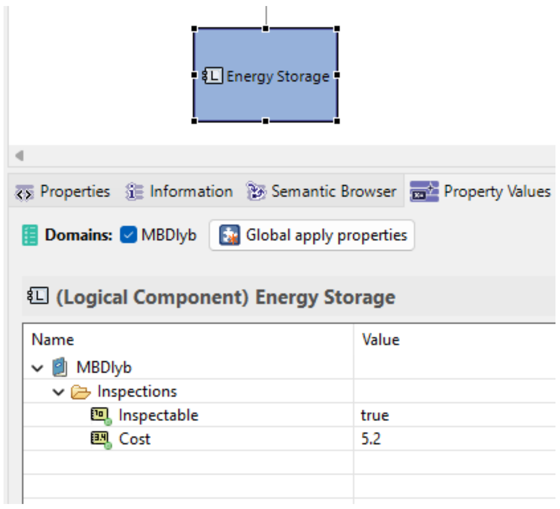

Sensor
Healthy 49.8
Broken 50.2
Control_Board
Healthy 100
Broken 0
Actuator
Healthy 99.0
Broken 1.0
Sensor.Transmit_measurement
Ok 49.8
NOk 50.2
Actuator.Cause_an_effect
Ok 49.3
NOk 50.7
I/O_Board.Retrieve_measurement
Ok 49.8
NOk 50.2
Control_Board.Drive_an_effect
Ok 49.8
NOk 50.2
I/O_Board.Transmit_error_message__Me...
Ok 100
NOk 0
I/O_Board
Healthy 100
Broken 0
Control_Board.Transmit_error_message_...
Ok 100
NOk 0
Control_Board.Transmit_error_message_...
Ok 100
NOk 0
Front_End_PC.Process_error_message__...
Ok 100
NOk 0
Front_End_PC.Process_error_message__...
Ok 100
NOk 0
Front_End_PC.Measurement_reception_er...
Absent 0
Present 100
Front_End_PC
Healthy 100
Broken 0
Front_End_PC.Control_error
Absent 49.3
Present 50.7

Figure 4.11: This Netica screenshot shows the Bayesian network (again slightly simplified) that MBDlyb produces after importing the Capella model shown in Figure 4.10. Compared to Figure 4.9, we see that now we have two error reporting chains in the middle of the diagram.

4.4 Diagnostic tests - inspections

In this section and the following ones we will discuss diagnostic test. The simplest kind of diagnostic test to represent in Capella is inspections. Suppose we have a hardware component in our system that can be directly inspected by a service engineer to check whether it is broken. The component that represents this hardware in Capella can be labelled as such by setting the value of the property `Inspectable` to true. The cost of the inspection can be filled into the `Cost` property. Figure 4.12 shows what this looks like in Capella.



Name	Value
MBDlyb	
Inspections	
Inspectable	true
Cost	5.2

Figure 4.12: The Capella diagram (top) shows a hardware component, and in PVMT (bottom) this is labelled as inspectable

When the Capella model is imported into MBDlyb, even if the option *Default tests on hardware* is off, a test with its result is added for this component, as shown in Figure 4.13.

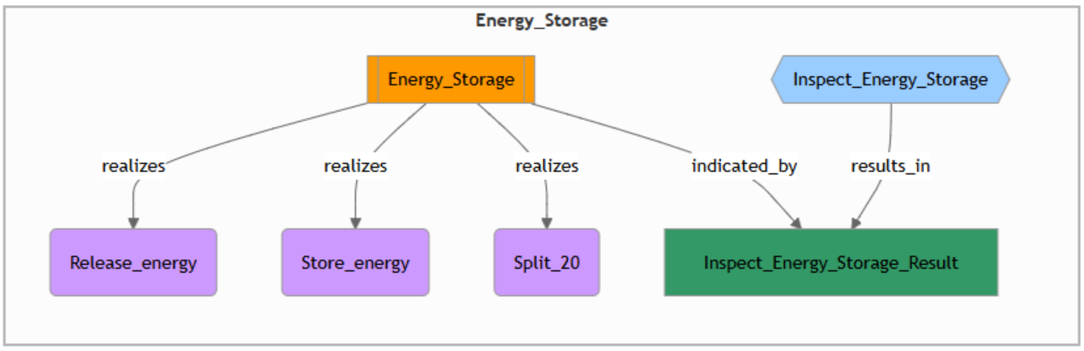


Figure 4.13: In MBDlyb we see component *Energy_Storage* (orange), realizing a number of functions (purple). There is also the diagnostic test *Inspect_Energy_Storage* (blue hexagon), which results in *Inspect_Energy_Storage_Result* (green rectangle). An arrow shows that the test result indicates the health of the component.

Guidelines

As mentioned above, a hardware component can be labelled as inspectable when it can be directly inspected by a service engineer. Examples are:

- checking whether a part is dirty, bent, or otherwise damaged

- using a multimeter to check the resistance between two points of a part, e.g., a cable
- checking whether a part, such as an LED, works by applying external power to it

This means that the inspection does not depend on any other components in the system. Here are some examples of such dependencies:

- The inspection requires power to be applied to the component and this power comes from a regular power supply in the system, e.g., checking the status of an LED or measuring a voltage or current using a multimeter.
- The inspection involves exchanging data with the component and this is done using a communication channel that is a regular part of the system, e.g., reading out digital status codes using built-in communication channels
- The inspection involves a measurement in the component and this is done using a mechanism that is a regular part of the system, e.g., checking a temperature using a built-in sensor

If any such condition applies, a more general mechanism, as described in Section 4.5, should be used to specify the test in Capella. The reason for this is, of course, that the dependency on the other component should be taken into account in the diagnostic model, which is not possible with a simple inspection. This also means that the dependency can be removed by using special test equipment for power, communication, or measurements, where this test equipment is not a regular part of the system being diagnosed.

How to specify the cost of a diagnostic test?

- Costs of tests should be specified in the same way throughout the Capella model, because all the tests will be ranked based on cost (and information gain) in the diagnoser
- Since in most cases labour is the most important contributor to the cost of diagnostics, it may be most natural to specify the cost of a test in terms of the time it takes the service engineer to perform the test. Typically, a minute is a good unit to express this
- It may also be possible to express the cost in monetary units, such as euros or dollars. Labour time can be converted to monetary units by applying an hourly or “minutely” labour rate
- If a test involves consumables, such as materials or energy, the cost of these can be added to the labour cost. If the labour cost is expressed in time units, consumable cost can be converted to time by dividing by the labour rate
- In general we recommend using labour minutes when the service engineer’s time is dominant in most tests and monetary units when consumables often play an important role

Although inspections are the simplest kind of diagnostic tests to model in Capella, this does not mean that every inspection is easy or cheap to perform. In some cases, an inspection requires removing the component from the system, which can take considerable time and effort. This should be reflected in the cost of the inspection.

4.5 Diagnostic tests - singular

A standard diagnostic test in our methodology is a function of the system that can be invoked explicitly by a service engineer and then either succeeds or fails. The way to specify a diagnostic test in Capella is as follows:

- Define a function (in the logical or physical architecture) that represents the test
- Set the `Diagnostic test` property to true and fill in a value for the cost of the test (see the guidelines in Section 4.4 about how to specify the cost)
- Define subfunctions of the test and allocate them to the components that are covered by the test
- Several test functions can be grouped as subfunctions of higher-level functions. Whenever such a higher-level function contains *only* tests, it will not be imported to MBDlyb.

Figure 4.14 shows what this looks like in Capella, whereas Figure 4.15 shows the page describing the diagnostic test in MBDlyb.

Guidelines

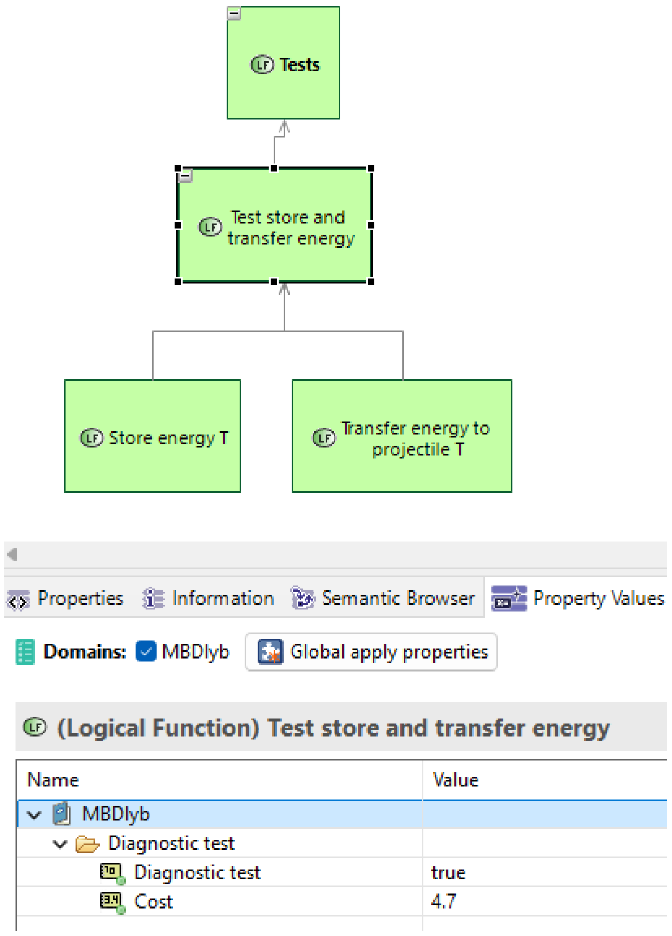
Often, a diagnostic test uses the regular functionality of some components. Since in Capella functions are organized in a strict tree structure, it is not possible to use the existing functions as regular functionality and as subfunctions of tests. Still, the relationship can be indicated clearly by subfunctions of the test having the same names as regular functions of their components. If it is desirable to distinguish the regular functions and their equivalent subfunctions of tests, you can make small modifications to the name. We often add a ‘T’ or even ‘T2’, ‘T3’, etc. to the names of test subfunctions that are equivalent to regular functions.

Another way to express a dependency of a diagnostic test on regular functionality is by a functional exchange. This could be done, for example, when a function in a test needs power (electric, hydraulic, pneumatic, mechanical, etc.), as illustrated in Figure 4.16.

Here we see a few test functions, which are subfunctions of *TestAudioSystem* and some of them are getting power from the regular function *DistributePower.* The idea here is that the test functions, such as *AmplifySignal T*, are invoked specifically for this test, but the regular functions on which they depend, such as *DistributePower* are just active as in normal operation. In practice, this distinction is not always clear, so it is good to know that for the diagnostics it does not matter: *TestAudioSystem* can only succeed as a test when both it subfunctions and all the regular functions on which they depend work correctly. A similar pattern can be used if a test subfunction depends on a regular function to ensure a certain condition (e.g., pressure, open door, etc.) or to provide some materials needed to perform the test.

Note that in Figure 4.16 we did not add functional exchanges *between* the functions in the test. There are two reasons for this:

- Functional exchanges between test subfunctions may help to illustrate how the test works, but they are not needed for diagnostics: with or without functional exchanges, the test will only succeed when all its subfunctions work correctly.



(Logical Function) Test store and transfer energy

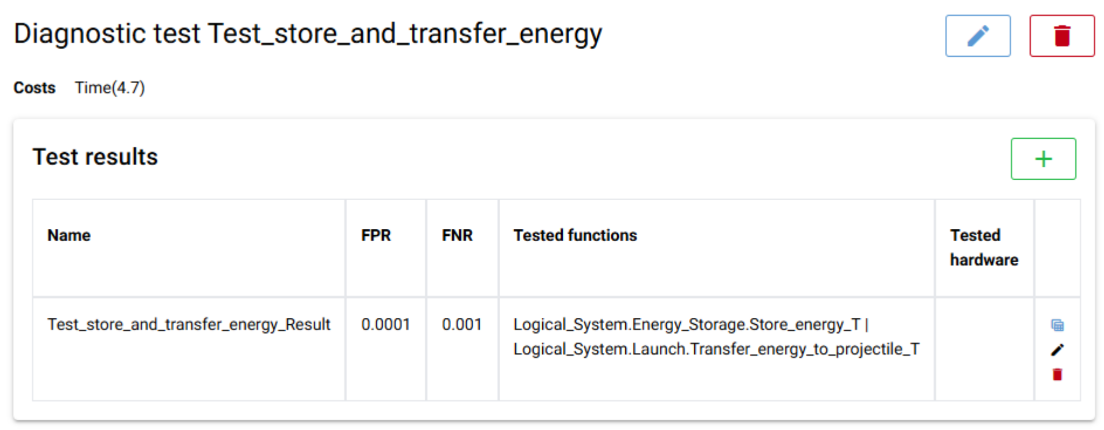
Name	Value
MBDlyb	
Diagnostic test	
Diagnostic test	true
Cost	4.7

Figure 4.14: In the Capella diagram (top), function *Test store and transfer energy* is selected and in PVMT (bottom) it is labelled as a diagnostic test. The diagram also shows subfunctions *Store energy T* and *Transfer energy to projectile T*, which would typically be allocated to different components that are covered by the test. (This allocation would be shown in a different kind of diagram in Capella.) In the diagram we also see parent function *Tests*, which will not be imported into MBDlyb.

- In certain circumstances (see Section 4.6) functional exchanges between test subfunctions may lead to unintended consequences.

Therefore, we recommend not to add functional exchanges *between* test subfunctions and to use them only from (regular) functions outside a test to test subfunctions.

It is also undesirable to add functional exchanges *from* test subfunctions *to* regular functions: this would mean that the regular functionality of the system would depend on test functions, which are not executed during regular use of the system. For that reason, in Figure 4.16, we have added specific test subfunctions *AmplifySignal T* and *Make Audio T*, instead of feeding the output of function *GenerateTestSignal* to the regular functions of the respective components.


Diagnostic test Test_store_and_transfer_energy

Costs Time(4.7)

Test results

Name	FPR	FNR	Tested functions	Tested hardware	
Test_store_and_transfer_energy_Result	0.0001	0.001	Logical_System.Energy_Storage.Store_energy_T | Logical_System.Launch.Transfer_energy_to_projectile_T		

Figure 4.15: This page in MBDlyb shows information about diagnostic test *Test_store_and_transfer_energy* after import from Capella

4.6 Diagnostic tests - composite

Performing an inspection (Section 4.4) or a singular test (Section 4.5) will give only a single bit of information: Either the test succeeds, in which all components involved are considered to be healthy, or it fails, in which case at least one of the components involved is broken. In practice, it is also possible that performing a test procedure will give us more detailed information, effectively telling us that some parts of the test have succeeded, while others have failed or have not even been performed. To model this in Capella, it is possible to label any subfunction of a test again as a test. This is illustrated in Figure 4.17.

Figure 4.18 shows what this looks like in MBDlyb.

When we open the diagnoser in MBDlyb, we see this test displayed as shown in Figure 4.19.

For such a composite test, we have several possible outcomes:

- The complete test succeeded. In the diagnoser we can fill in *Ok* for the top-level test result, and we do not need to fill in anything else.
- The test did not succeed completely, but some of the subtests did succeed. In the diagnoser we can fill in *NOk* for the top-level test result and *Ok* for the subtests that succeeded. If we know which subtests failed, we can fill that in as well, but if we do not have that information, we leave it open.
- The test was not performed completely, but some subtests were done. In the diagnoser we can leave the top-level test result open, just as the results for any subtests that were not done. For the subtests that were done, we can fill in the individual results.

MBDlyb will not stop us from entering inconsistent results, such as a successful overall test with failed subtests, but such inconsistencies will never arise from a real test (provided it was designed and performed correctly).

Guidelines

When should we use a composite test? There are several typical cases for composite tests:

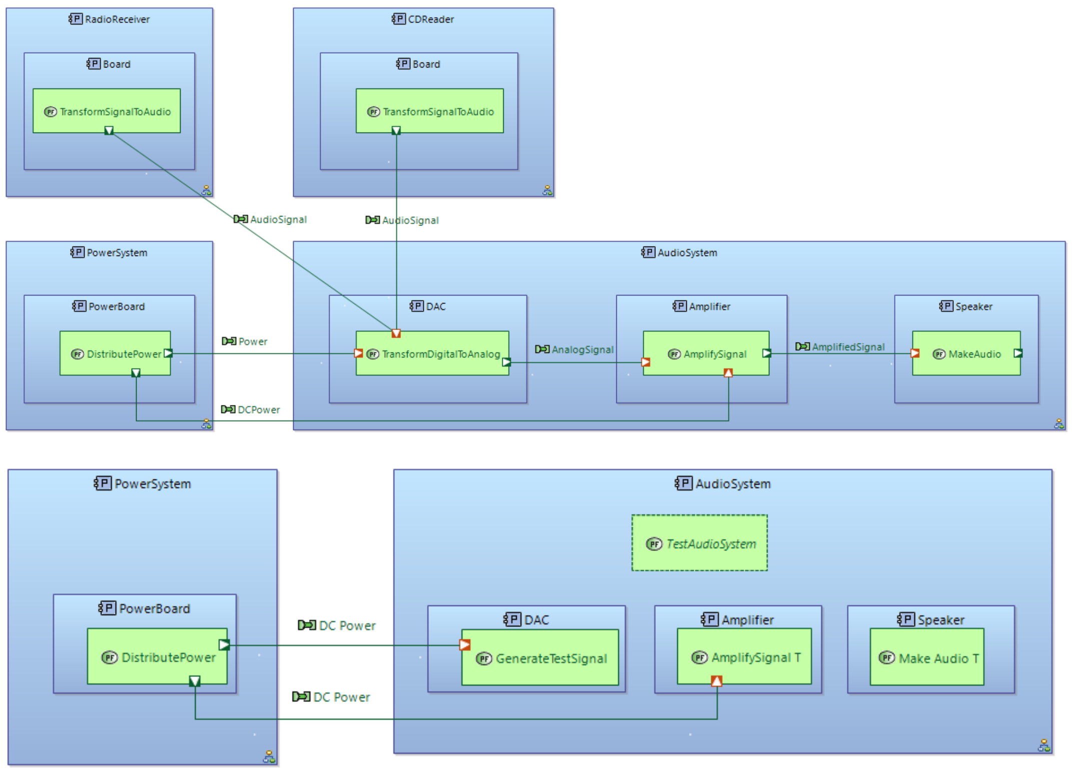
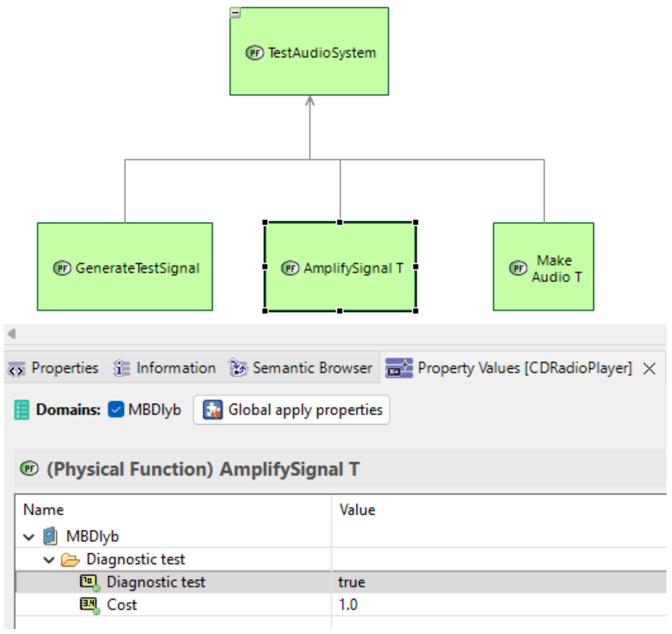


Figure 4.16: The topmost Capella diagram shows a summary of the normal operation of a radio/CD player. The bottom diagram shows the test function *TestAudioSystem* with its subfunctions *GenerateTestSignal*, *AmplifySignal T*, and *MakeAudio T*. Comparing this to the regular functionality shown in the top diagram, we see that the power for two of the subfunctions is provided by the regular function *DistributePower*. However, the subfunctions that test the *Amplifier* and *Speaker* components need a different input signal than in regular operation, which is generated by the function *GenerateTestSignal*. Therefore, we cannot use the regular functions here, so we made copies *AmplifySignal T* and *MakeAudio T*. Functional exchanges between the test functions were left out for reasons explained in the text.

- A regular test happens to be able to produce multiple results. For example, when testing a setup as in figures 4.8 and 4.10, it may be possible to detect that some sensor readings are received, in which case the function *Retrieve measurement* is working correctly. Similarly, it may be possible to detect that the actuator is causing an effect, so that we can conclude that the function *Control an effect* is working. Note that it is not necessary that every execution of the test must produce all the results: The MBDlyb diagnoser will allow us to fill in certain results and leave others open.
- There may be a group of relatively cheap tests that need a common, relatively expensive preparation. For example, for a consumer device such as a radio/CD player, it may take a bit of work to take the cover off, but when that is done, you can easily do a number of measurements using a multimeter. In this case we can model the group of tests as one overall test in Capella, where the individual measurements are modelled as subfunctions. The cost of the overall test should, of course, include the effort for the common preparation plus the individual tests. Again, modelling the group as a single overall test does not force us to perform all the individual tests: We may skip some subtests and leave their results open in the MBDlyb diagnoser.

When modelling a composite test in Capella, we need to pay special attention to its internal



Domains: MBDlyb

(Physical Function) AmplifySignal T

Name	Value
MBDlyb	
Diagnostic test	
Diagnostic test	true
Cost	1.0

Figure 4.17: In the Capella diagram (top) we see that subfunction *AmplifySignal T* of *TestAudioSystem* has been selected, and in PVMT (bottom) it has been labelled as a test. Note that the cost of the subtest is given a default value by PVMT, but it is ignored by MBDlyb: only the cost of the overall test is considered. (Of course, *TestAudioSystem* itself is also a test, and in addition *GenerateTest-Signal*, but this is not visible in the figure.)

organization: for each subtest, we need to make sure that its dependencies are exactly right. As an example, let's consider the control loop (see Figure 4.24) that we will encounter in Section 4.7. It is a slight extension of figures 4.8 and 4.10, with the omission of the error reporting chain. We could model a test for this control loop as illustrated in Figure 4.20.

In Figure 4.20 we see that there are *no* functional exchanges between subfunctions of the test (all labelled with 'T'), which means that each subfunction could have an individual test result. On the other hand, the diagram shows functional exchanges from the function *Distribute power* to some of the subfunctions, and this means that none of these subfunctions can work correctly without power. As suggested above, we have labelled the subfunctions *Retrieve measurement* and *Control an effect* as subtests, although this is not visible in the diagram. Therefore, even if the overall *Test control loop* fails, we may be able to detect that the subtest *Retrieve measurement T* went well and therefore that the I/O Board is healthy. Similarly, the subtest *Control an effect T* may have succeeded, in which case we can conclude that the control board is healthy.

Now suppose that success of the subtest *Retrieve measurement* would indicate that not only the I/O board, but also the sensor is healthy, and also that the success of the subtest *Control an effect* would guarantee the health of not only the control board, but also the actuator. There are two ways we can indicate this in Capella. The first way is by organizing the tests hierarchically as shown by figures 4.21 and 4.22. Not visible in the diagrams, we have labelled the new intermediate functions *Measurement T* and *Effect T* as subtests. Now the hierarchical relationships between the functions, best shown in Figure 4.21, ensures that success of these subtests reflects correctly on the components involved.

We also see in Figure 4.22 that we have allocated the subtests to some components, for example we allocated subtest *Measurement T* to component *I/O Board*. Officially this is dis-

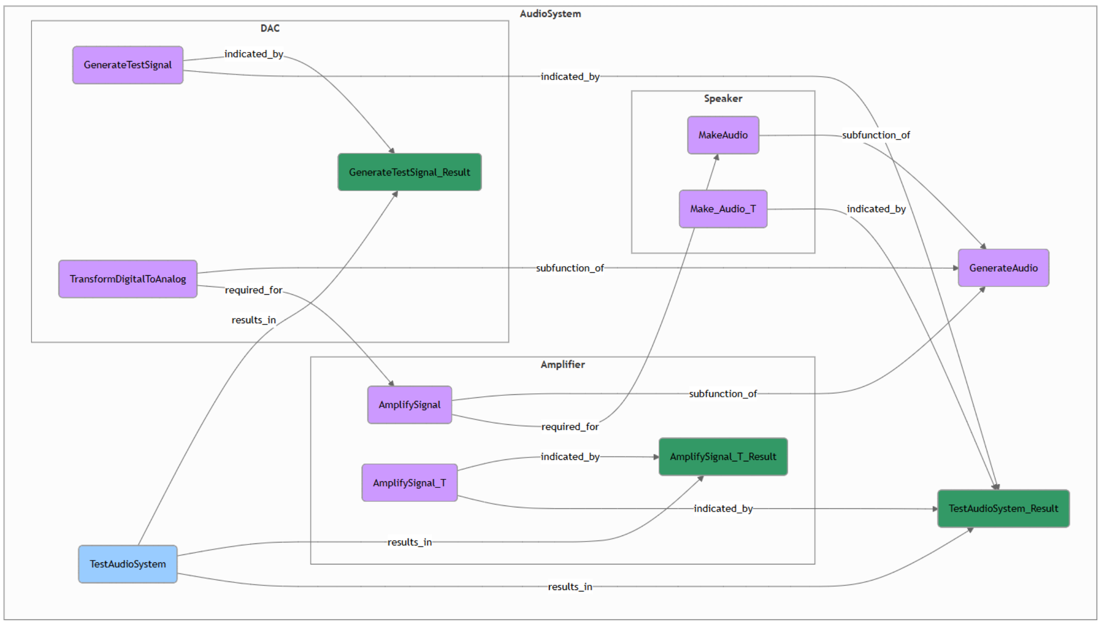
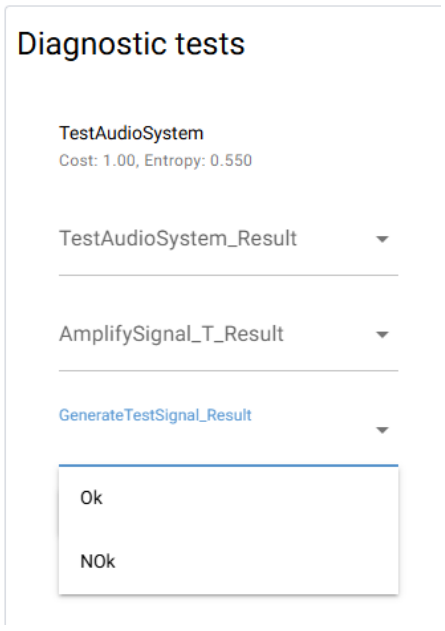


Figure 4.18: This MBDlyb diagram presents various components (white boxes) with their allocated functions (purple boxes). It also shows the diagnostic test *TestAudioSystem* as a blue box and its multiple results as green boxes. The various subtests cannot be executed independently, and therefore they are show as normal functions.

couraged in Capella, where only functions without subfunctions are supposed to be allocated to components. Nevertheless, we find it illustrative to indicate which component bears the main responsibility for each higher-level function, even though it does not matter for the diagnostics

An alternative way to indicate additional dependencies between subfunctions of a test is by using functional exchanges. This is shown in Figure 4.23. Here there is an additional functional exchange from *Transmit measurement T* to *Retrieve measurement T*, which indicates that the correct working of function *Retrieve measurement T* depends on the correct working of *Transmit measurement T*. For the same reason we added a functional exchange called *Feedback* from function *Cause an effect T* to *Control an effect T*. Alternatively we could have duplicated the original functional exchange *Control signal* from Figure 4.24 and labelled function *Cause an effect T* as a subtest. In Section 4.7 we will see that we could even add more functional exchanges, for example to illustrate the working of a test, provided we mark them properly as irrelevant for diagnostics.

For the diagnostics, these two ways of expressing dependencies between subtests are largely equivalent, but in practice it is best to reflect in the model the actual way a test works. This remark holds more in general: whenever modelling diagnostic tests, whether singular or composite, it is important to reflect the actual operation, especially the dependencies, of all the subfunctions as much as possible, while taking into account the simplifying assumptions of our model-based diagnostics methodology.


Diagnostic tests

TestAudioSystem
Cost: 1.00, Entropy: 0.550

TestAudioSystem_Result

AmplifySignal_T_Result

GenerateTestSignal_Result

Ok

NOk

Figure 4.19: This shows a composite test as it appears in MBDlyb's diagnoser. We see that we can provide outcomes for the overall test and the subtests separately. Here the result of subtest *Generate-TestSignal* is selected, so we can choose to enter the outcome of the subtest.

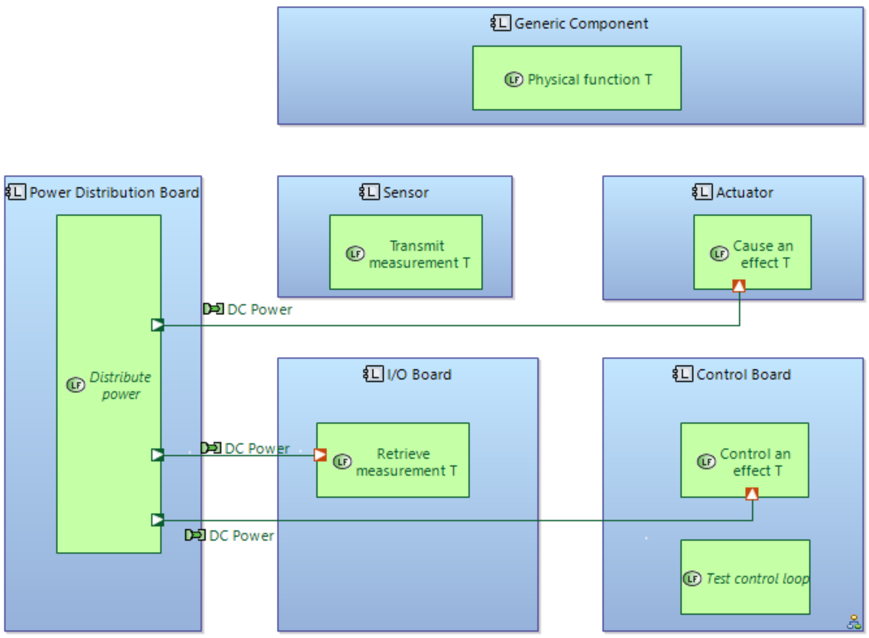


Figure 4.20: Diagnostic test for a control loop as modelled in Capella. See the text for more details.

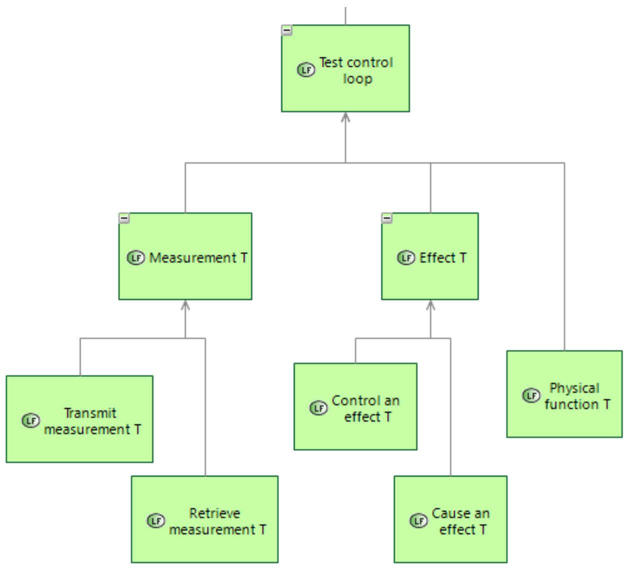


Figure 4.21: Hierarchically organized diagnostic test for a control loop in a Capella functional breakdown diagram

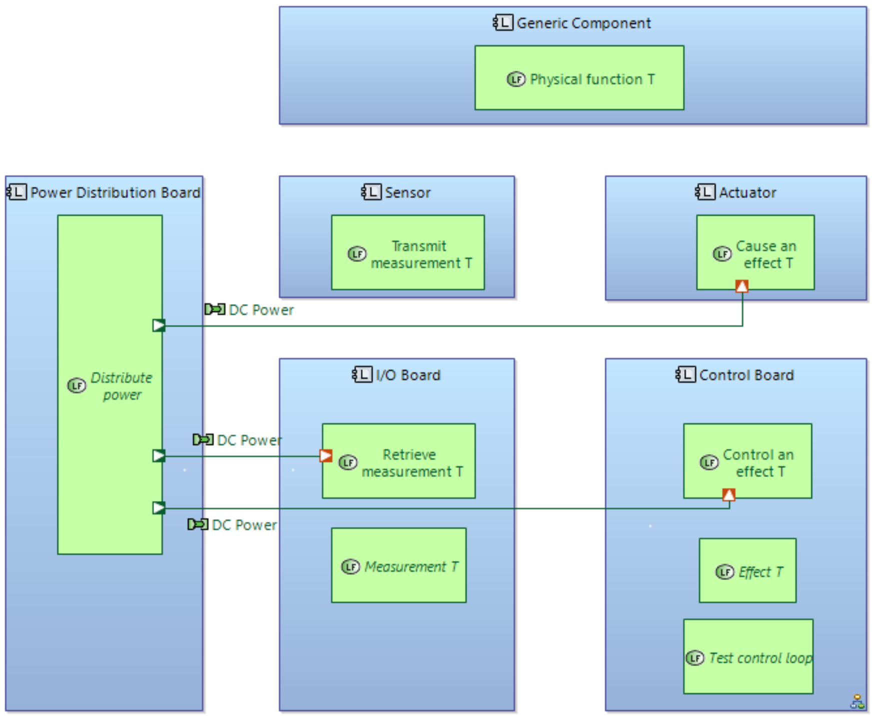


Figure 4.22: Hierarchically organized diagnostic test for a control loop in a Capella architecture diagram

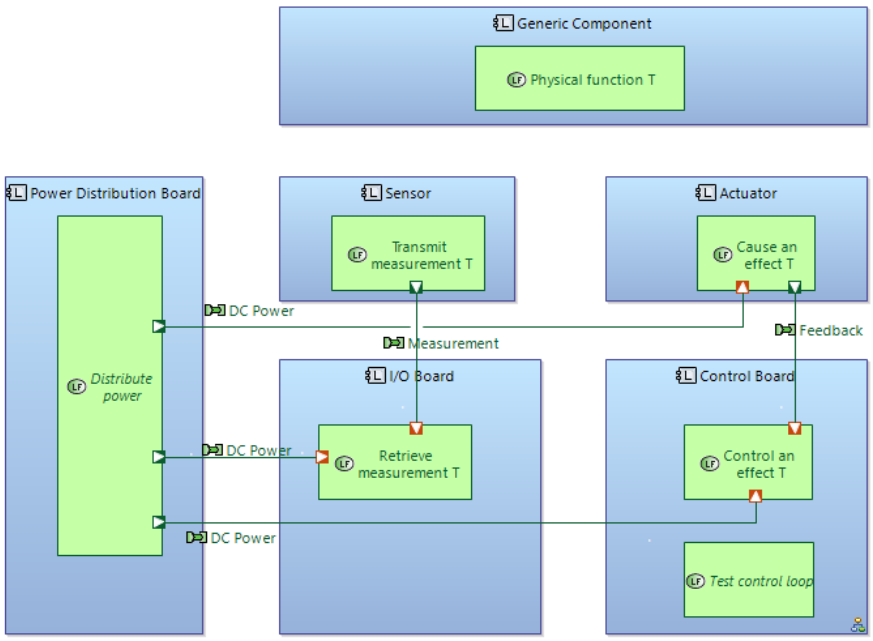


Figure 4.23: Diagnostic test with additional functional exchanges for a control loop in Capella

4.7 Control loops

It is common for a complex cyber-physical system to have several control loops. Figure 4.24 provides an example. In practice, a control loop can have fewer or more components and functions involved. Nested control and intertwined control loops can also occur.

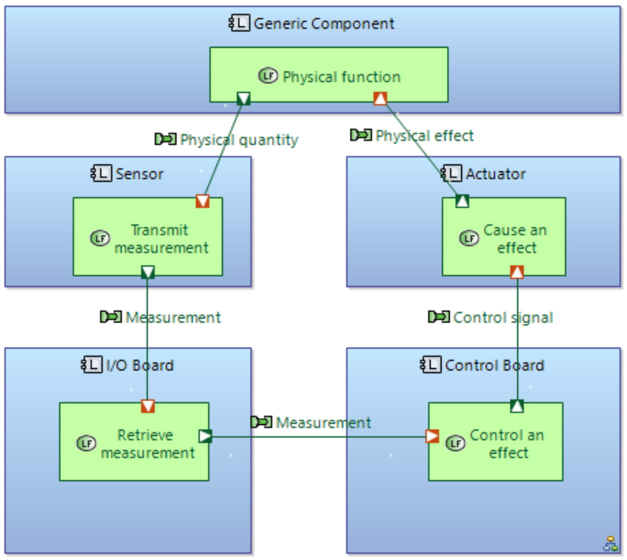


Figure 4.24: Control loop as modelled in Capella

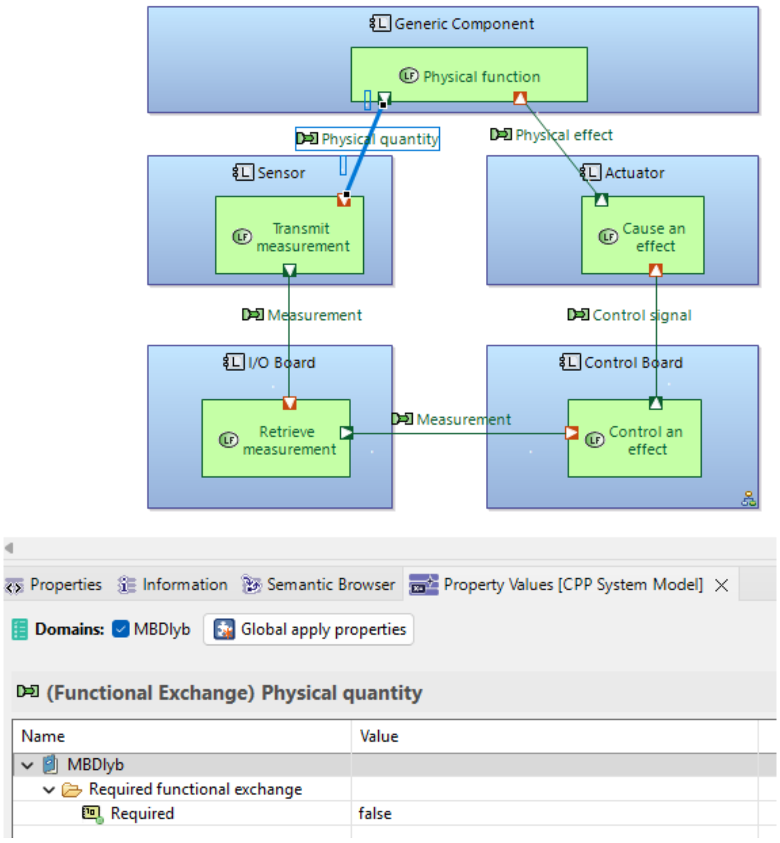
How to model control loops for diagnostic purposes has been discussed in [37]. Here we discuss how to do this in practice using Capella. Most importantly, we need to deal with the fact that Bayesian networks cannot have loops, so we need to break the loop. MBDlyb will automatically break control loops, and the *Check model* function can report on the loops that it found. In many cases it is nevertheless desirable to indicate explicitly in Capella where the loop should be broken. For this purpose, it is possible to label a functional exchange as not required by setting the `Required` property to false. This is shown in Figure 4.25.

You could interpret this as follows: There is an interaction going on between the source and target functions, but the target function (in our example: *Transmit measurement*) can work correctly even if the source function is not working as desired. Because of this the target function does not depend on the source function in our diagnostic Bayesian network.

Guidelines

Typically, a good place to break a loop is at the incoming exchange to a function that performs a measurement, since often the value of the measurement is affected by the function before it, but if that preceding function is broken, typically the measurement can still be performed accurately (provided the sensor hardware itself is not broken).

In Section 4.6 we discussed how to model tests related to control loops (among others), and we saw that it is important to model precisely the dependencies of the subfunctions of a test, either by hierarchical function structures or by functional exchanges. In order to get adequate diagnostics, it is also important to accurately model the dependencies leading to error messages that occur during normal operation (i.e. outside of tests).



Properties | Information | Semantic Browser | Property Values [CPP System Model]

Domains: MBDlyb | Global apply properties

(Functional Exchange) Physical quantity

Name	Value
MBDlyb	
Required functional exchange	
Required	false

Figure 4.25: In the Capella diagram (top) we see that functional exchange *Physical quantity* is selected, and in PVMT (bottom) it is marked as not required.

For example, if we want to provide an error message for our function *Retrieve measurement*, as shown in Figure 4.10 and similar, we need to make sure that the error message depends on the function *Transmit measurement* (and probably on the function *Distribute power*, which is not visible in Figure 4.10). Therefore, it is good that we marked the functional exchange from *Physical function* to *Transmit measurement* as not required as in Figure 4.25.

But now we *also* want to be able to generate an error message when the control loop as a whole does not work. If we added that error message to function *Control an effect* in Figure 4.24, then we would not get the correct dependencies, because the dependency chain does not reach back to *Physical function*, since we broke the loop there. To express the correct dependencies, we can use a functional hierarchy, as shown in Figure 4.26. Here we moved the function *Control an effect*, representing the complete control loop, to the top of the hierarchy, and we expressed the contribution made by the control board by the function *Drive and effect*.

The corresponding architecture diagram is shown in Figure 4.27. Here we see that the error message *Control error* is generated by the overall control loop, which means that it depends on all the functions and components involved in the loop. By contrast, the error message *Measurement reception error* only depends on the sensor and the I/O board (plus the power supply chain, not shown here). Now if any of these error messages is generated, the MBDlyb diagnoser will flag the correct set of culprits, and we can use the appropriate tests to localize the problem further.

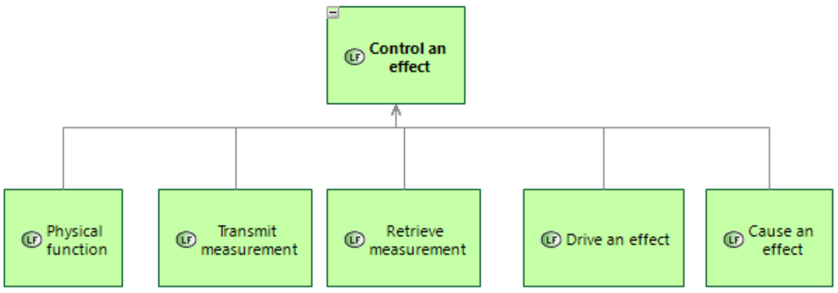


Figure 4.26: Function hierarchy for a control loop as modelled in Capella

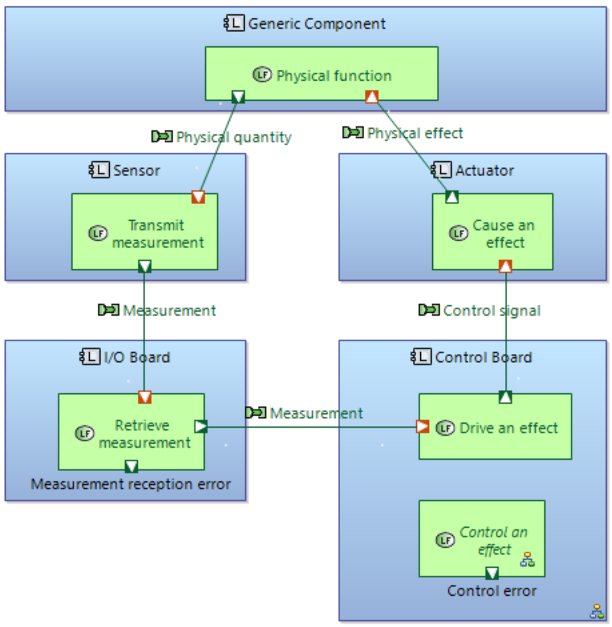
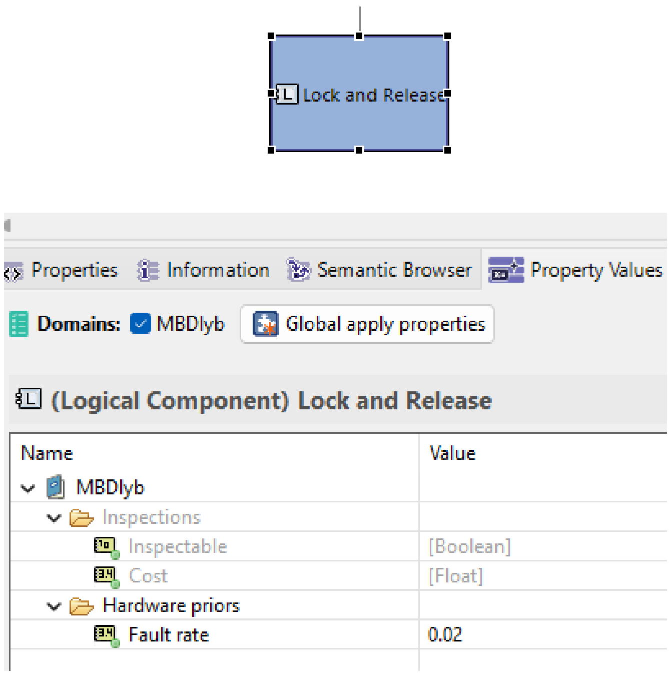


Figure 4.27: Hierarchically organized control loop in a Capella architecture diagram

4.8 Hardware priors

MBDlyb assumes a default value of 0.01 for the probability that a hardware component is faulty. (This is a prior value, which holds as long as there is no additional diagnostic information available.) In Capella, we can set a non-default value for a specific component by setting the `Fault rate` property to the desired value, as shown in Figure 4.28.

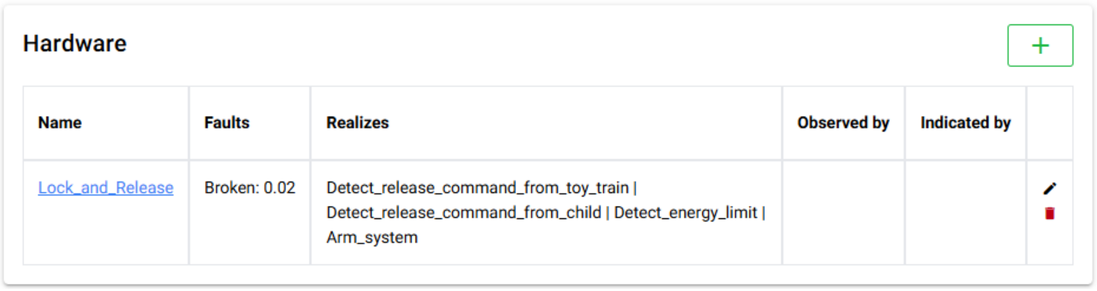


(Logical Component) Lock and Release

Name	Value
MBDlyb	
Inspections	
Inspectable	[Boolean]
Cost	[Float]
Hardware priors	
Fault rate	0.02

Figure 4.28: Hardware element with explicit fault rate, as modelled in Capella

When importing the Capella model into MBDlyb, this new fault rate will now be shown in place of the default one, see Figure 4.29.


Hardware

Name	Faults	Realizes	Observed by	Indicated by	
Lock_and_Release	Broken: 0.02	Detect_release_command_from_toy_train | Detect_release_command_from_child | Detect_energy_limit | Arm_system			

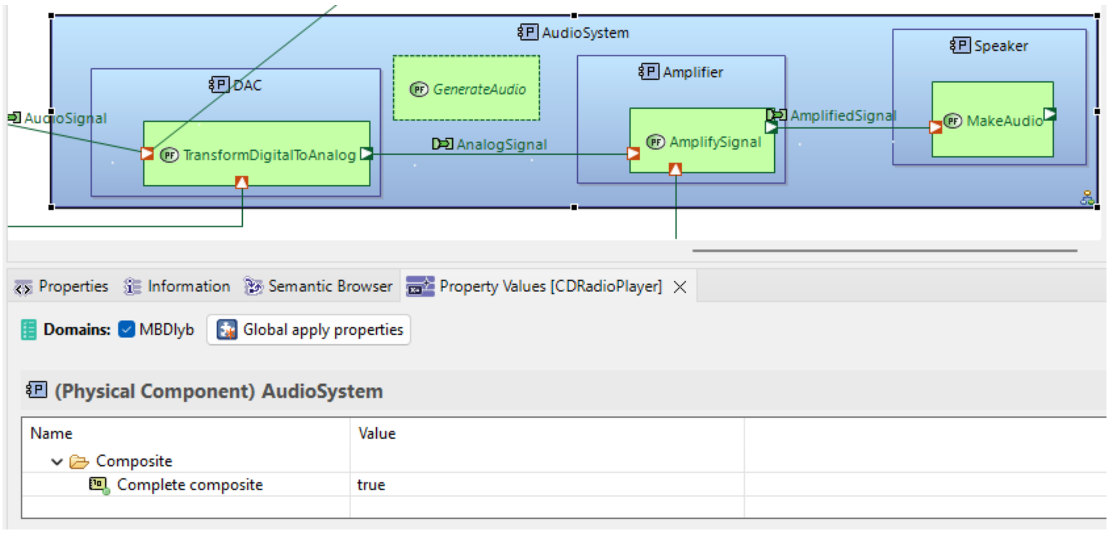
Figure 4.29: Hardware element with non-default fault rate, as shown in MBDlyb

Guidelines

It is not easy to get an exact number for the fault rate of a component. In some situations, this information can come from the field. But let's consider that the most relevant part of the results of the MBDlyb diagnoser is the ordering of the suspected culprit components and of the suggested tests, not the exact values of the probabilities. Fortunately, once some evidence (observables or test results) is available, the ordering of the MBDlyb diagnoser's results is not very sensitive for the prior fault rates. That means that we do not need very precise, absolute values for these priors, but we can still use them to emphasize or de-emphasize some components in the results: A component that we know to be more reliable than average we can give a lower fault rate, and a component that we know to break more often we can give a higher one.

4.9 Complete composites

In principle, for every Capella component that has functions allocated to it, MBDlyb creates a hardware node, so that the component can be diagnosed as healthy or broken. However, there can be situations where this is not desired. In that case you can label the component as a complete composite, as Figure 4.30 shows.



Name	Value
Composite	
Complete composite	true

Figure 4.30: Component labelled as complete composite in Capella

A component labelled as a complete composite like this will not be assigned a hardware node in MBDlyb, but its functions are still present, as shown in Figure 4.31.

Guidelines

A typical situation where this complete composite label is applicable is when you have a higher-level component that is completely described by its subcomponents in the model (i.e. there is nothing in the component that is not in the subcomponents). The high-level component can still have functions assigned to it, but they will typically have subfunctions allocated to the subcomponents.

When building the model, we can start by modelling the higher-level components and assign higher-level functions to them. Then, as we are completing the model, we may add subcomponents and subfunctions, and at some point the set of subcomponents may be complete, so we can label the higher-level component as a complete composite.

But this is not the only possibility: We can also decide to add one or more subcomponents that do not represent the complete higher-level component. In that case, the higher-level component remains a separate entity, which can be diagnosed separately from its subcomponents, and it should *not* be labelled as complete composite. An example of this is a device that needs a battery to operate: The battery could be absent or discharged, which we may want to check during diagnosis, but the device could also have faults that are not related to the battery. This is illustrated by Figure 4.32.

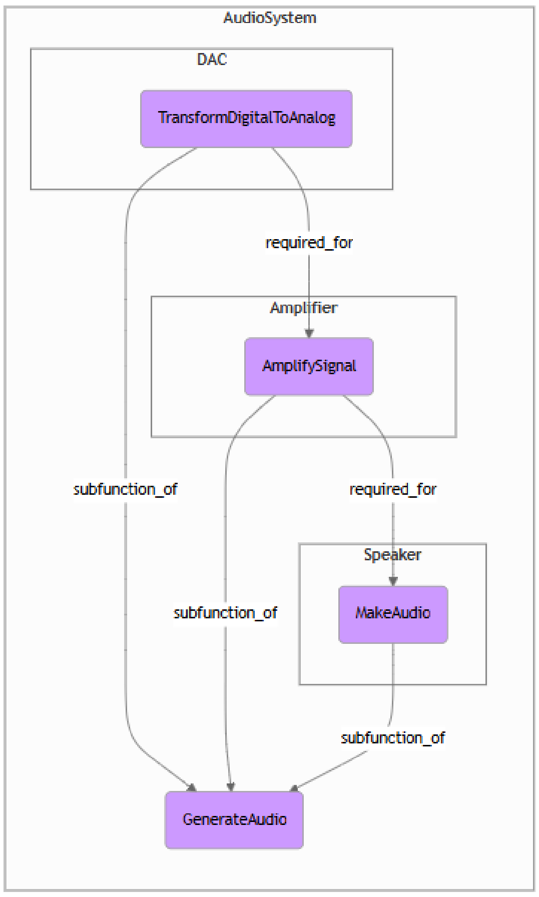


Figure 4.31: Complete composite as shown in MBDlyb

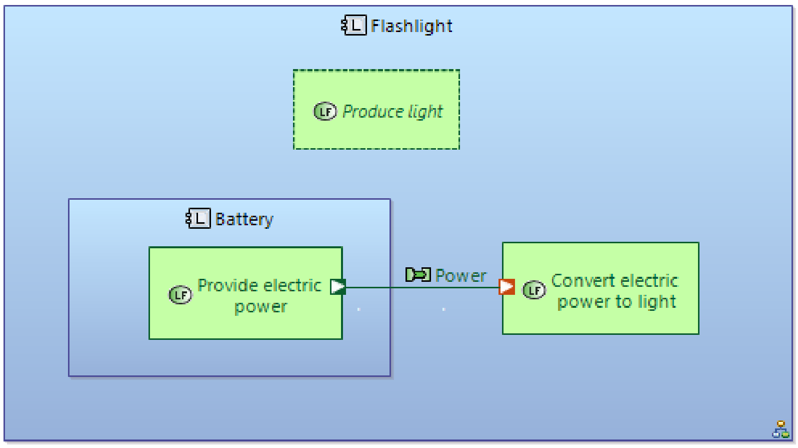


Figure 4.32: Example in Capella where complete composite is not applicable

5 Observations

Apart from the methodology extensions discussed in Chapter 4, we also made a number of further observations when we applied the methodology to an industrial system. We detail them in the following sections.

5.1 Modelling effort

As mentioned in Chapter 3, we applied our methodology to a large production printer under development, focusing on one complex module. Here are some estimated indications of the size of our Capella model for the industrial system we studied:

- 127 components (covering more than 95% of the components in our focus module)
- over 200 functions
- 65 errors and observables (about 50% of the ones relevant for our focus module)
- 24 diagnostic tests (about 30% of the relevant ones)

Since components and functions (including tests) are organized hierarchically in Capella, these numbers comprise the entities at all levels of the hierarchies. Note that the relatively low coverage of errors, observables, and tests is related to the fact that the system was still under development and many of these items were not yet known.

The total effort to construct the model was about 3–4 person-months. The actual modelling was just a fraction of this. The largest effort was spent on collecting the information about the system and understanding it.

Modelling the system structure was relatively easy, partly due to the fact that the system experts mostly agreed on the actual structure (although not necessarily on the names of the various components).

However, modelling the functions allocated to the various components was significantly more difficult. To some degree, this was due to the absence of a sufficiently detailed functional model of our industrial system shared among the experts. But we also realized that the construction of a good functional model, complete with allocation to components, needs a great deal of care. You need to ask yourself not only “What does this component do?” but also “What happens to directly related functions and components when this component is broken.” (Note that the effect on *indirectly* related functions and components will be modelled by the Bayesian network and computed by MBDlyb, so you do not need to do this end-to-end analysis by hand.) After answering these questions, you still need to consider whether it is possible to simplify the system model, while still getting diagnostics of sufficient quality.

This also means that if a standard MBSE model, with components and functions, were available, you would still expect a significant effort to check whether the functional model is suitable for diagnostic purposes and adapting it where necessary.

Modelling observables, errors, and diagnostic tests turns out to be relatively straightforward, but since tests are modelled as functions, the above considerations also apply here.

5.2 Modelling tools

As mentioned in Chapter 3, we used Capella as the MBSE tool for modelling the system. Capella has some clear advantages, such as:

- It is easily available at no cost.
- It is very mature and well documented.
- It was very stable and predictable during our modelling activities.
- Our system model could be expressed in a set of relative easy-to-understand diagrams. The standard colours for diagram elements certainly helped here. Because of this, the diagrams were helpful in discussing the system model with domain experts.
- Capella offers a sizeable set of plug-ins for additional tasks. As mentioned in Section 4.1, we made extensive use of the PVMT plug-in.
- There is a well-maintained Python library for interacting with the models, which we used in MBDlyb's Capella import module.

However, Capella also has some disadvantages:

- There are strict limitations on diagram structure. For example, you cannot show the function hierarchy in an architecture diagram, so you cannot always tell a full story in a single diagram (see figures 4.21 and 4.22).
- There is only rudimentary support for generalization, so you cannot show a generalization hierarchy in a diagram.
- There is no support for classes and instances, which makes reuse difficult (there is a reuse mechanism in Capella, but it is hard to use.) So you cannot introduce the concept of a wheel and then say that a car has four of them, each with a specify role (e.g., front left). For this reason, we did not fully model all replicated components in our system.

In view of the disadvantages of Capella, we explored SysML v2 [20, 21, 38] as well. We expect that this may mitigate some of Capella's disadvantages. Unfortunately, during the time dedicated to this, the tooling that was available was not sufficiently mature. The full range of SysML v2 diagrams was not yet supported and a round trip (import and export) of a textual SysML v2 model would often result in loss of information. Therefore, we could not use SysML v2 for all our modelling tasks. We have good hope that this will change in the near future.

5.3 Output

We compared the output of the MBDlyb diagnoser with diagnostic procedures that were hand-crafted by system developers. There were considerable differences.

- Partly these differences could be attributed to the simplifying assumptions in our approach, to which the system developers were not bound:

    - the assumption that either *all* functions of a component work correctly or *none* of its functions work
    - the fact that the diagnoser needs an explicit notification that certain errors are absent, whereas a hand-crafted procedure can rely implicitly on errors being generated in a specific order, and therefore on certain errors being absent

- Other differences could be attributed to simplifications in the model, which we applied to keep calculation times reasonable, particularly for the diagnoser.
- Finally, there is considerable freedom in designing diagnostic procedures, so some differences between manually designed and automatically generated procedures are perfectly acceptable.

In any case, system experts judged that the diagnostic procedures as produced by the diagnoser were reasonable, in the sense that they led to the identification of the actual culprit component without too much unnecessary effort. So in other words, the differences between hand-crafted and MBDlyb-generated procedures were considered largely inconsequential.

The response times of the MBDlyb were most noticeable in the diagnoser. Every time a new piece of evidence is added (an observable/error or a test result), the diagnoser has to calculate the values in the Bayesian network to determine the most likely culprit component. Moreover, it must calculate the entropy for each of the tests. On a modern (laptop) computer, this can take about 10–30 seconds, which is longer than some of the diagnostic tests in our model would take in practice. The actual time depends on the size of the model and the number of tests, but also on the particular structure of the model. For a diagnoser algorithm to support a service engineer in his work, we aim to perform a diagnostic step in under 10 seconds, so that the engineer does not need to wait too long before choosing the next action (test or repair). Therefore, we would like to lower these response times considerably.

We regularly studied the Design for Diagnostics page of MBDlyb, and we found that it nicely illustrates the kind of analysis that we would like to support, but its actual contents are not yet suitable for practical use. On the one hand, this is because it assumes that the values for *all* observables are given, which is not realistic. In principle, you could add all the errors that the system generates when a single component fails, which could be more than one, but the assumption that all the errors that are not generated did not occur, is incorrect and will lead to the wrong output. This is because software may stop generating errors at some point after the first one. Moreover, some components may not be reachable any more, so their errors will never make it to the user interface. Further work is needed to improve the Design for Diagnostics page, so that it can deal more gracefully with smaller sets of observables. On the other hand, a limitation of the Design for Diagnostics page is that often in the early and middle design phases, it is not yet known what the observables of the various components are, because the mechanisms to provide these observables are still to be designed. We would like the Design for Diagnostics page to provide some support for that process, so that we could call it ‘Design for Diagnosability’.

6 Conclusions

In this chapter, we first revisit the research questions guiding our activities, and then discuss future work.

6.1 Answers to research question

Let's look again at our research question, as introduced in Section 1.1:

RQ1 How to leverage the (potential) interplay between functional and structural description-based methodologies for hardware (HW) diagnosis?

As mentioned in Section 1.1, we broke this research question down into the following sub-questions:

RQ1.1 How does the aforementioned function-based diagnostic methodology scale to component-level?

RQ1.2 How to ensure proper integration of the diagnostic-relevant (observability) information in the MBSE models?

To answer RQ1.1, we applied our methodology to a complex module in the context of a large system, a production printer developed by Canon Production Printing. Regarding the number of components and the multidisciplinary nature of their interactions, we can safely say that the above system serves as a representative example for many kinds of high-tech systems. According to our observations (Section 5), applying the methodology took a reasonable amount of effort and the results were good. Therefore, we conclude that our methodology, with the extensions discussed in Chapter 4 scales well to systems with a large number of components. We believe that the methodology can also be applied successfully in other domains than industrial printing, since our case study already involved multiple domains (mechanics, electronics, thermodynamics, etc.).

Experts at our industrial partner are satisfied by these results and expect that our methodology will provide an alternative for their current FMECA process (Failure Mode Effect and Criticality Analysis) [5] with the following advantages:

- Our methodology requires less manual analysis, which takes effort and is error-prone
- Our methodology can be applied repeatedly during the system design process with less effort
- Our methodology depends less on natural language descriptions, which can be ambiguous
- The models resulting from applying our methodology can be more easily reused when a new version or variant of a system is developed

For these reasons, our industrial partner intends to introduce our methodology into their development and service processes in a step-wise way.

When we look at RQ1.2, we see that this is addressed by the significant extensions and improvements of the methodology described in Chapter 4. There it is shown how a basic functional/structural MBSE model can be enriched with diagnostic information, including observables, errors, tests, loop handling, hardware priors, and complete composites. The fact that we can express all the diagnostic-relevant information in the Capella model makes it possible to create and maintain one single MBSE model for our system that contains all the information needed for diagnostics. This ensures that we keep all the information together, and that in turn is important to keep the information consistent.

Initially we intended the method to be used with pre-existing MBSE models, but based on the above, it appears to be worthwhile to develop a new model, even if it is only used for diagnostics at first. Replacing the laborious and error-prone FMECA process can be a good reason to start with system-level modelling, and from there the usage of the model could be expanded to other areas, such as reliability, serviceability, or testing. The system model could also be used to assess other aspects, less directly related to diagnostics, such as cost price and variability. All this could expand into a solid MBSE practice in a development organization. Of course, we are still interested in exploring how our methodology could be integrated in a such a broad MBSE approach in industry.

6.2 Further work

Although considerable progress has been made in extending and improving our diagnostic methodology, further research and development is required.

First, as mentioned above, our industrial partner is intending to introduce our methodology into its processes. This will take several steps, and we hope to remain in close contact during this change, observing and assisting. We expect to encounter a number of opportunities for improvement, which we can take up to get our methodology more mature. In this context, we also plan to explore another MBSE formalism and tool as an alternative for Capella. Most likely, this formalism will be SysML v2. We aim at discovering how we can build a suitable system model in SysML v2 and integrate all the diagnostic-relevant information in such a model along the lines of Chapter 4. If we can develop a new MBDlyb input module for this formalism, we can try it out in practice. In this way we can find out whether such an alternative formalism and tool would fit better in an industrial setting.

Another line of research we intend to follow is exploring the application of the methodology in very early design phases. In these earlier phases, much less is known about the detailed system design, even though several parts of an existing design may be reused. Nevertheless, we hope to be able to do an early assessment of diagnosability by combining high-level design sketches with earlier detailed designs.

We will also need to improve our approach to Design for Diagnostics. This includes assessing which information elements will realistically be available for analysis and what kinds of outputs the developers need. One such output we envisage is a metric for completeness, which can tell us how much progress has been made during system development towards an acceptable coverage of the diagnostic needs of a service engineer in the field.

In all these activities, we strive towards broad applicability of our methodology across many domains in the high-tech industry and beyond.

References

[1] T. C. Nägele et al. *SD2Act 2024: Guided Diagnosis of Functional Failures in Cyber-Physical Systems*. 2025-R10010. Eindhoven: TNO, 2025. URL: `https://resolver.tno.nl/uuid:52f484ea-2638-4403-88a4-65305b245697`.

[2] Jean-Luc Voirin. *Model-Based System and Architecture Engineering with the Arcadia Method*. 2017. DOI: `10.1016/c2016-0-00862-8`.

[3] Leonardo Barbini, Carmen Bratosin and Thomas Nägele. 'Embedding Diagnosability of Complex Industrial Systems Into the Design Process Using a Model-Based Methodology'. In: *PHM Society European Conference* 6.1 (29th June 2021), p. 9. ISSN: 2325-016X. DOI: `10.36001/phme.2021.v6i1.2806`.

[4] Leonardo Barbini, Alvaro Piedrafita Postigo and Micha Lipplaa. *Probabilistic graphical models for performance diagnostics. Methods and applications to a print quality case*. Report R13164. The Netherlands: TNO, 2026.

[5] Christian Spreafico, Davide Russo and Caterina Rizzi. 'A State-of-the-Art Review of FMEA/FMECA Including Patents'. In: *Computer Science Review* 25 (1st Aug. 2017), pp. 19–28. ISSN: 1574-0137. DOI: `10.1016/j.cosrev.2017.05.002`.

[6] L. Barbini, T. C. Nägele and A. Piedrafita. *Vision and Outlook on High-Tech Equipment Diagnostics*. 2024-R12778. Eindhoven: TNO, 2024. URL: `https://resolver.tno.nl/uuid:d5e21d85-b400-4e09-bcc8-27bd0ca100ca`.

[7] Randall Davis and Walter Hamscher. 'Chapter 8 - Model-based Reasoning: Troubleshooting'. In: *Exploring Artificial Intelligence*. Morgan Kaufmann, 1988, pp. 297–346. DOI: `10.1016/B978-0-934613-67-5.50012-5`.

[8] Johan de Kleer and James Kurien. 'Fundamentals of Model-Based Diagnosis'. In: *IFAC Proceedings Volumes*. 5th IFAC Symposium on Fault Detection, Supervision and Safety of Technical Processes 2003, Washington DC 36.5 (June 1997), pp. 25–36. ISSN: 1474-6670. DOI: `10.1016/S1474-6670(17)36467-4`.

[9] Amaury Vignolles, Elodie Chanthery and Pauline Ribot. 'A Holistic Advanced Diagnosis Approach For Systems Under Uncertainty'. In: *32nd International Workshop on Principle of Diagnosis – DX 2021*. Hamburg, Germany, Sept. 2021. URL: `https://laas.hal.science/hal-03282343`.

[10] Johan de Kleer. 'Focusing on Probable Diagnoses'. In: *Proceedings of the Ninth National Conference on Artificial Intelligence - Volume 2*. AAAI'91. Anaheim, California: AAAI Press, 14th July 1991, pp. 842–848. ISBN: 978-0-262-51059-2.

[11] Ethan Scarl. 'Sensor Placement for Diagnosability'. In: *Annals of Mathematics and Artificial Intelligence* 11.1 (Mar. 1994), pp. 493–509. ISSN: 1573-7470. DOI: `10.1007/BF01530756`.

[12] Matthew Daigle, Indranil Roychoudhury and Anibal Bregon. 'Diagnosability-Based Sensor Placement through Structural Model Decomposition'. In: *PHM Society European Conference* 2.1 (8th July 2014). ISSN: 2325-016X. DOI: `10.36001/phme.2014.v2i1.1545`.

[13] Amaury Vignolles, Elodie Chanthery and Pauline Ribot. 'An Overview on Diagnosability and Prognosability for System Monitoring'. In: *European Conference of the Prognostics and Health Management Society (PHM Europe)*. (Virtual conference), Italy, July 2020. URL: `https://laas.hal.science/hal-02891028`.

[14] Kapil Dev Sharma and Shobhit Srivastava. 'Failure Mode and Effect Analysis (FMEA) Implementation: A Literature Review'. In: *Journal of Advance Research in Aeronautics and Space Science* 5.1 (2018), pp. 1–17.

[15] Enno Ruijters and Mariëlle Stoelinga. ‘Fault Tree Analysis: A Survey of the State-of-the-Art in Modeling, Analysis and Tools’. In: *Computer Science Review* 15–16 (Feb. 2015), pp. 29–62. ISSN: 1574-0137. DOI: 10.1016/j.cosrev.2015.03.001.

[16] Axel Berres et al. ‘OMG RAAML Standard for Model-based Fault Tree Analysis’. In: *INCOSE International Symposium* 31.1 (July 2021), pp. 1349–1362. ISSN: 2334-5837, 2334-5837. DOI: 10.1002/j.2334-5837.2021.00905.x.

[17] *RAAML: Risk Analysis and Assessment Modeling Language*. Version 1.1 beta. URL: https://www.omg.org/spec/RAAML.

[18] Matthew Hause et al. ‘The SysML Modelling Language’. In: *Fifteenth European Systems Engineering Conference*. Vol. 9. 2006, pp. 1–12.

[19] Jon Holt and Simon Perry. *SysML for Systems Engineering*. IET, 2008. 351 pp. ISBN: 978-0-86341-825-9. Google Books: OEKtufR7spYC.

[20] Manas Bajaj, Sanford Friedenthal and Ed Seidewitz. ‘Systems Modeling Language (SysML v2) Support for Digital Engineering’. In: *Insight* 25.1 (2022), pp. 19–24. DOI: 10.1002/inst.12367.

[21] OMG Systems Modeling. *SysML v2 Release: The latest incremental release of SysML v2*. GitHub. URL: https://github.com/Systems-Modeling/SysML-v2-Release.

[22] Mec Octavio Companioni Sarmiento. ‘Adaptation of the Concepts of the RAAML Standard for Applications Based on SysML V2’. PhD thesis. University of Applied Sciences, 2025.

[23] Jordi Vallverdú. ‘Challenges and Controversies of Generative AI in Medical Diagnosis’. In: *Euphyía* 17.32 (15th Dec. 2023), pp. 88–121. ISSN: 2683-2518. DOI: 10.33064/32euph4957.

[24] Nasrullah Abbasi et al. ‘Generative AI in Healthcare: Revolutionizing Disease Diagnosis, Expanding Treatment Options, and Enhancing Patient Care’. In: *Journal of Knowledge Learning and Science Technology ISSN: 2959-6386 (online)* 3.3 (15th Aug. 2024), pp. 127–138. ISSN: 2959-6386. DOI: 10.60087/jklst.vol3.n3.p127-138.

[25] Chetana Hegde. ‘Anomaly Detection in Time Series Data Using Data-Centric AI’. In: *2022 IEEE International Conference on Electronics, Computing and Communication Technologies (CONECCT)*. 2022 IEEE International Conference on Electronics, Computing and Communication Technologies (CONECCT). July 2022, pp. 1–6. DOI: 10.1109/CONECCT55679.2022.9865824.

[26] Zahra Zamanzadeh Darban et al. ‘Deep Learning for Time Series Anomaly Detection: A Survey’. In: *ACM Comput. Surv.* 57.1 (7th Oct. 2024), 15:1–15:42. ISSN: 0360-0300. DOI: 10.1145/3691338.

[27] Min Du et al. ‘DeepLog: Anomaly Detection and Diagnosis from System Logs through Deep Learning’. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’17. New York, NY, USA: Association for Computing Machinery, 30th Oct. 2017, pp. 1285–1298. ISBN: 978-1-4503-4946-8. DOI: 10.1145/3133956.3134015.

[28] Shuang Hao et al. *Synthetic Data in AI: Challenges, Applications, and Ethical Implications*. 3rd Jan. 2024. DOI: 10.48550/arXiv.2401.01629. arXiv: 2401.01629 [cs]. Pre-published.

[29] Reza Alizadeh, Janet K. Allen and Farrokh Mistree. ‘Managing Computational Complexity Using Surrogate Models: A Critical Review’. In: *Research in Engineering Design* 31.3 (1st July 2020), pp. 275–298. ISSN: 1435-6066. DOI: 10.1007/s00163-020-00336-7.

[30] S. Deb, K.R. Pattipati and R. Shrestha. ‘QSI’s Integrated Diagnostics Toolset’. In: *1997 IEEE Autotestcon Proceedings AUTOTESTCON ’97. IEEE Systems Readiness Technology Conference. Systems Readiness Supporting Global Needs and Awareness in the 21st Century*. Sept. 1997, pp. 408–421. DOI: 10.1109/AUTEST.1997.633654.

[31] Andrew Hess, Jacek S Stecki and Shoshanna D Rudov-Clark. 'The Maintenance Aware Design Environment: Development of an Aerospace PHM Software Tool'. In: *Proc. PHM08* 16 (2008), p. 17.

[32] *MBDlyb*. ESI. URL: `https://esi.nl/research/output/tools/mbdlyb`.

[33] *Capella | Open Source MBSE Tool*. URL: `https://mbse-capella.org/`.

[34] Thomas Nägele. *Leveraging System Architecture Models for Diagnosis | TNO-ESI | Capella Days 2024*. 17th Dec. 2024. URL: `https://www.youtube.com/watch?v=TKoa1teGLRk`.

[35] *Norsys - Netica Application*. URL: `https://www.norsys.com/netica.html`.

[36] Eclipse Capella. *PVMT*. GitHub. URL: `https://github.com/eclipse-capella/capella/wiki/PVMT`.

[37] M. J. A. M. van Gerwen and L. Barbini. *Diagnosing Systems with Loops: Applying Bayesian Networks on Cyclic Graphs*. Eindhoven: TNO, 2025. URL: `https://resolver.tno.nl/uuid:a3d93793-1dc7-4638-b8de-0dbc75fe12b4`.

[38] Bram van der Sanden and Pierre America. *SysMLv2 as foundation for ESI tools?* Report R12436. Eindhoven, The Netherlands: TNO, 6th Dec. 2024. URL: `https://repository.tno.nl/SingleDoc?docId=65497`.