PCDT: Power Cabinet Diagnosis Tool

Geenen, R.

Award date:
2020

Department of Electrical Engineering
Electronic Systems Research Group

PCDT: Power Cabinet Diagnosis Tool

Master Thesis

Dhr. R. (Röbke) Geenen (0843558)
robke.geenen@prodrive-technologies.com

Supervisor Eindhoven University of Technology
Prof.dr.ir. J.P.M. (Jeroen) Voeten
J.P.M.Voeten@tue.nl

Supervisor Prodrive Technologies
Dhr. E. (Eric) van Uden
eric.van.uden@prodrive-technologies.com

Eindhoven, July 9, 2020

ID, Version, Date: D20052716881R04_Thesis, R04, 2020-07-09

Preface

The idea for the Power Cabinet Diagnosis Tool (PCDT) was initially sparked during a meeting between Prodrive and the customer some ten years ago. Our ultimate dream was to be able to apply a 'test vector' to the combination of a control rack and the power cabinet infrastructure of a lithography machine. This test vector was envisioned to enable easier testing and improved coverage at the supplier level of the power cabinet infrastructure to facilitate drop in shipment.

Unfortunately the realization of this future vision has been put on the back burner. However, due to circumstances, new interest has peaked recently at the side of Prodrive. While the idea originated to improve testability by means of a test vector, the focus in this project has shifted towards improving testability by gathering and interpreting diagnostics information.

It is a great opportunity for me to attempt to obtain the degree of Master of Science (MSc) in the Embedded Systems (ES) program. This requires the successful completion of a graduation project. In order to satisfy both this requirement and the desire for fulfillment of the dream, the execution of this project has been cast into the form of a graduation project. Each graduation project in the ES program is divided into a preparation phase followed by an execution phase. This report reflects the decisions made during, and results obtained from, the execution phase of the graduation project.

I would like to extend my sincere gratitude to Jeroen Voeten for supervising the project from the side of the TU/e and for providing comprehensive and detailed feedback. Furthermore I cannot thank Eric van Uden enough for supporting me through my initial experiences as a designer at Prodrive and for being my daily supervisor.

Our dream has remained a dream for the past decade, but it is about to come true.

Röbke Geenen

Contents

List of Figures

List of Tables

1 | Introduction

Integrated circuits are created by optically imaging patterns onto a silicon wafer that is covered with a film of light-sensitive material called photoresist. This procedure is repeated dozens of times to prepare a single wafer for further processing which will realize the actual electronic circuits. [1] Optically imaging of the patterns is done by so called photolithography machines, these machines are used to produce nearly all types of integrated circuits.

Figure 1.1: A power cabinet filled with power amplifiers, the subject of this project

Part of the functionality required for the optical imaging process is moving the silicon wafers around inside the machine and physically adjusting properties of the lenses used in the optical path. These physical motions are realized by means of actuators inside the machine. The machines operate at high speed and high precision to provide an optimal yield. At the same time the physical motions involve big masses, requiring large forces. Keeping the physical motions under control therefore requires highly complex control systems and high-force actuators.

Part of the control loop are power amplifiers which take as input the desired behaviour for the actuators as calculated by the control system and translate this input into high current signals to actually drive the actuators. A simplified representation of the control system as used in the photolithography machines is shown in Figure 1.2. Most performance is gained from the feedforward controller which is designed to match the behaviour of the inverse of the plant as closely as possible. The feedback controller is mainly used to provide relatively minor corrections. In the diagram of Figure 1.2 the power amplifiers are situated in the signal path from the controllers to the plant.

Sensing and feedback elements are handled by different parts of the machine and are not part of the power cabinet infrastructure. Multiple power amplifiers together with a power amplifier controller — to allow for external communication — and a power supply — to provide power to all power amplifiers — are assembled into a single power cabinet. An opened power cabinet is shown in Figure 1.1.

Two types of power amplifier are in use, differing in the type of high current signal they can generate. The Power Amplifier Alternating Current (PAAC) type generates alternating current signals used for driving the long stroke actuators. The Power Amplifier Direct Current (PADC) type generates direct current signals used for driving the short stroke actuators. Both types of power amplifier are available in the same form factor.

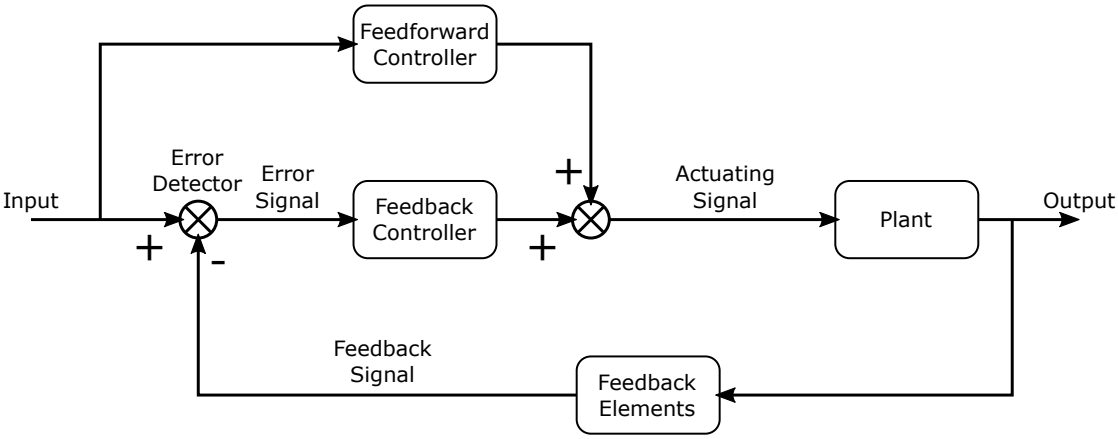

Figure 1.2: Simplified representation of the control system as employed within the photolithography machines

To satisfy the machine performance requirements, the whole machine runs on a heartbeat in the order of several kilohertz. A consequence of this strategy is that the power amplifiers are updated on what to do every couple dozen microseconds. The power amplifiers are told what to do by means of so called set-points that are communicated to them. Each set-point contains information on the high current signal that the power amplifier is requested to generate. For a PADC this information will contain the value of the current in amperes. For a PAAC this information will contain the amplitude of the signal in amperes and the phase in radians. The high current signals generated by the power amplifiers based on the received set-points translate to actual physical behaviour of the actuators in terms of position, speed, torque and power in intricate mechanical ways that are out of scope for this document.

1.1 Report organization

The organizational structure of this thesis report is as follows: first Section 1 introduces the parts of the photolithography process relevant to this thesis and relates this to the context of the power cabinet infrastructure. Next the problem and the requirements necessary for a satisfactory solution are described. In Section 2 the current existing literature related to to the current project is surveyed. Section 3 provides the connection between the existing literature and the requirements. The general approach to the solution of the problem is detailed and high level design choices are explained. The theory behind the inference engine underlying the chosen design approach is handled in Section 4. In Section 5 the detailed design choices leading to the concrete design as implemented are then given and explained. Next the procedures used to verify and qualify the implementation of the design are described in Section 6. In Section 7 the extent to which the resulting design satisfies the requirements is discussed. Finally a conclusion and suggestions for future work wrap up the report in Section 8.

1.2 Problem introduction

Multiple variants of the power amplifiers and power cabinets are produced by Prodrive for the high-tech industry. All produced units undergo extensive testing before delivery, which is per-

formed with a similar infrastructure to the one applied by the customers. This means that multiple power amplifiers can be tested simultaneously. If one of the power amplifiers fails during testing, the only indication about which module failed is the illumination of a red LED on the front panel of the failed unit. No additional diagnostic information is logged and shown to the test operator, complicating the servicing of the failed module. The power amplifiers do however record quite a lot of diagnostic information internally during operation, such as currents, voltages, temperatures, fan status and more.

It would be useful if this information could be recorded by the testing software for logging purposes. Furthermore it would be useful to have diagnostics software that can compute diagnoses of the root causes of any failures. The output of the diagnostics software could then also be recorded by the testing software, which can then immediately report the diagnoses to the user. Logging of the information and diagnoses leads to improved traceability of structural issues, while useful feedback can immediately be provided to ease solving of the failure at hand. The purpose of this project is to design an extension to the existing power cabinet infrastructure with the capability of extracting the existing diagnostic information from the power amplifiers and to perform root cause analysis based on this information.

1.3 Infrastructure overview

An overview of the existing power cabinet infrastructure is shown in Figure 1.3. Solid rectangles represent physical components, while the arrows between them represent communication links. Dashed rectangles represent software or firmware running on the physical components. Highlighted in red are the components that need to be designed in this project or for which the design requires adaptation. The power cabinet infrastructure exists of multiple components which will be detailed below.

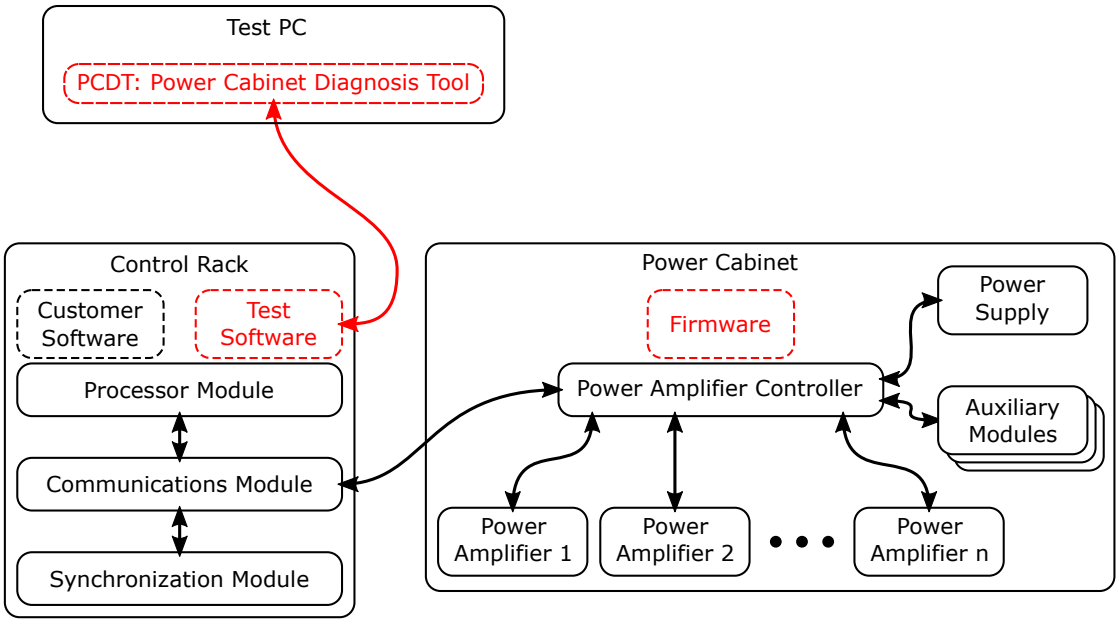

Figure 1.3: Overview of the power cabinet infrastructure

The processor module inside the control rack runs software that determines the set-points to be sent to the power amplifiers. In a production environment the software determines this by communicating with other components of the machine. In a test setup the software communicates with a test PC and the set-points are derived based on the aspects of the power amplifier or power cabinet that need to be tested. Aspects to be tested include but are not limited to the maximum power that an amplifier can deliver, the rise and fall times of the output signal of an amplifier and in the case of a power cabinet the connections between the components.

The communications module connects the power cabinet to the control rack and it does so by translating between the communication interface to the processor module and the communication interface to the power amplifier controller.

The synchronization module receives synchronization pulses and triggers the communications module. This ensures that the whole machine runs on the aforementioned heartbeat. Currently the synchronization module is a separate module, but it is possible that another type of communications module will be employed in the future that does not require a separate module for synchronization.

The power amplifier controller receives instructions from the control rack and distributes those to the power amplifiers. The communication from the communications module is synchronized to the machine heartbeat, but the link to the power amplifier controller can introduce variable latency, leading to unwanted jitter. The power amplifier controller employs a Phase Locked Loop (PLL) to minimize the effects of this jitter.

The power amplifiers receive set-points from the power amplifier controller at regular intervals, based on the machine heartbeat, and actually generate the high current signals that the set-points prescribe. The high current signals are finally fed to actuators in the machine.

The power supply is responsible for taking the main power coming into the power cabinet and converting this to a power source suitable for powering the power amplifiers. The power supply also contains some intelligence and is therefore connected to the power amplifier controller in a similar fashion as the power amplifiers.

A number of additional modules are located in the power cabinet which are not directly involved in the signal path from software to actuators. The power amplifier controller has digital input and output signals which can be used to connect functionality like an emergency shut-off button or other machine and product safety modules.

1.4 Problem statement

The lack of diagnostic information from the power cabinet infrastructure impacts the lead time of finding root causes of defect modules. Improving the availability of diagnostic information and implementing an automated diagnosis system allows failures to be automatically recognized. This means that failures no longer have to be manually recognized, which leads to a decrease in the root cause analysis time of modules at Prodrive.

Furthermore the lack of diagnostic information impacts the lead time of getting a machine back up and running after a failure. An automated diagnosis system can report which module has failed without the need for swapping modules by trial and error. This leads to machines at the customers of the customer to be up and running more quickly, decreasing down time of the machines.

Lastly the lack of diagnostic information means that it is not always clear if a module is defective or not. When this is the case, or when a module is swapped by trial and error but is not actually defective, the module needs to be sent back to Prodrive to be re-tested. Only after re-testing can the module be trusted again. An automated diagnosis system will decrease

the number of modules that need to be re-tested, since non-defective modules can be recognized more accurately in the field. This leads to fewer modules without faults being unnecessarily shipped between Prodrive and the customer.

1.5 Scope and stakeholders

The automated diagnosis system developed in this project encompasses multiple facets in order to provide different features to different users. All users of the solution are stakeholders to the project, since they each have their own use-case requirements. Since their requirements are key to designing a successful solution to the problem, an overview of the stakeholders is presented next. Figure 1.4 shows the relations between the stakeholders and the use-case scenarios of the solution. The stakeholders for PCDT including their time-frame of activity and their responsibilities are listed in Table 1.1.

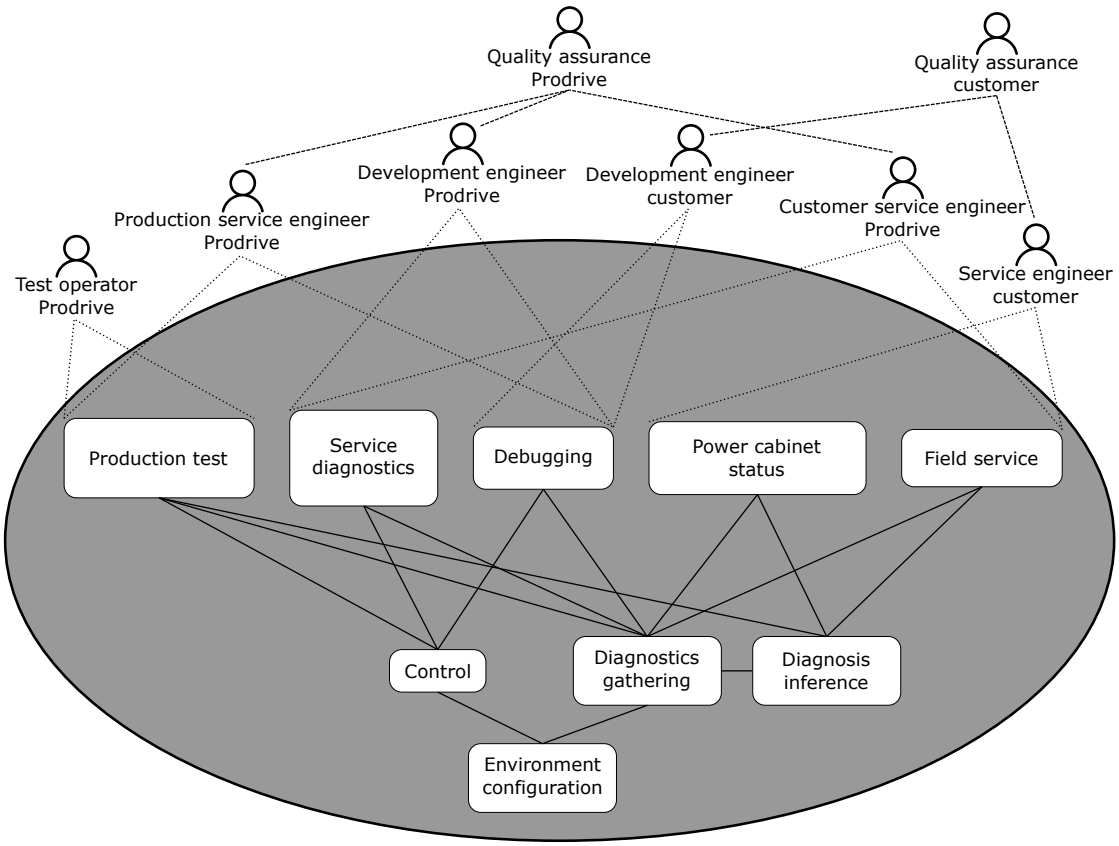

Figure 1.4: Overview of the stakeholders involved with PCDT

PCDT provides support for the following use-case scenarios:

- The **production test** use-case scenario is used by the test framework to show which power amplifier failed and why in the case that a failure is detected by the higher level test. This information is also logged to provide improved traceability.

Stakeholder	Time-frame	Responsibilities
Quality assurance Prodrive, customer	development, support and maintenance	Ensuring structural issues are adequately investigated and solved in order to improve overall quality of delivered goods.
Test operator Prodrive	production	Testing all produced power amplifiers to ensure they are fully functional.
Production service engineer Prodrive	production and repair	Determining why a failed module is non-functional.
Development engineer Prodrive	development, support and maintenance	Ensuring that new products function in the correct way, debugging and verification is necessary for this.
Development engineer customer	development	Integration of new power amplifier types or new power cabinet configurations.
Customer service engineer Prodrive	maintenance	Determining why modules have failed in the field.
Service engineer customer	support and maintenance	Restoring the machine to working order and minimizing downtime for customers.

Table 1.1: Stakeholders for PCDT

- The **service diagnostics** use-case scenario is used by the service engineer to read out diagnostics information from a power amplifier that failed during a production test or in the field. The main goal is to provide useful information to the service engineer that will help determine why the module failed.
- The **debugging** use-case scenario allows monitoring of diagnostics information while predetermined inputs are provided to the power amplifiers. The main purpose is to allow debugging of power amplifiers and power cabinets and performing of system verification during development. A further purpose is to allow service engineers to reproduce and investigate new faults that are not yet encoded in the expert rules.
- The **power cabinet status** use-case scenario allows verification of the condition of a complete power cabinet. If any of the power amplifiers reports a non-operational state it will be flagged for investigation.
- The **field service** use-case scenario is used by the service engineer to determine which power amplifier or amplifiers failed or showed abnormal behaviour while the machine was in operation. This feature will be used to determine which modules to swap in order to get the machine up and running again as fast as possible.

The design step required to provide support for these use-case scenarios results in the following abstract components which are required to be part of PCDT:

- The **control** component is necessary because certain users benefit from the ability to control the power amplifiers manually.
- The **diagnostics gathering** component is responsible for actually collecting the diagnostics information from the power amplifiers, the core of all applications.
- The **diagnosis inference** component is necessary to interpret the raw information from the power amplifiers and to infer diagnoses. The information gathered by the *diagnostics*

gathering component is very low level; it consists only of bits in registers. This type of information is not directly suitable for identification of failed modules or providing solutions to common problems. This component interprets the information providing higher level facts, such as the magnitude of a current, measured in amperes. Furthermore this component combines the facts using expert knowledge thereby inferring new information and ultimately providing concrete diagnoses.

- The **environment configuration** component is required since the *control* and *diagnostics gathering* components are unaware of the actual configuration of the power cabinet and control rack. This component is responsible for configuring any configuration-dependent aspects of the solution.

1.6 Goal and requirements

The goal of this project is to design an automated diagnosis system that is able to automatically detect all known failure modes of the power cabinet infrastructure. Several requirements need to be met in order to achieve this goal and satisfy all use-case scenarios; the automated diagnosis system needs to:

1. be able to automatically infer diagnoses;
2. be usable after production at Prodrive and in the field;
3. be easy to extend and maintain;
4. be based on rules which an expert might use when diagnosing a failure manually, the reason for inclusion of this requirement is explained in Section 3.2;
5. provide an intuitive interface for the domain expert;
6. provide an acceptable level of responsiveness, the inference of a diagnosis should not take too long;
7. be able to be deployed directly after implementation.

The automated diagnosis system as designed in this project is then implemented in a tool named the Power Cabinet Diagnosis Tool (PCDT).

1.7 Real-world use-case examples

Several use-cases have been encountered in the past where PCDT could have caught the failure mode had it existed at the time. A couple examples of this are listed below. The way in which these examples could be converted for use with PCDT is shown in Section 5.4.

- The power for the power amplifiers is delivered through a power supply and a fault in the cabling between the power supply and a power amplifier would lead to a non-operational power amplifier. A corresponding expert deduction for this situation is: ***if*** *the voltage at the output of the power supply is correct* ***and*** *the voltage at the input of power amplifier x is* ***not*** *correct* ***then*** *the problem is in cable y from the output of the power supply to the input of power amplifier x.*

- In order to cool the power amplifiers, two neighbouring power amplifier modules x and y are usually clamped to a cold plate with a single clamp between them. Forgetting to tighten this clamp leads to overheating of the modules. A corresponding expert deduction for this situation is: ***if*** *the temperature of power amplifier x is too high* ***and*** *the temperature of power amplifier y is too high* ***then*** *the clamp between power amplifiers x and y is not tightened properly.*

- A high but not too high temperature inside a power amplifier might lead to a failure in the near future, a solution could be to swap the power amplifier early to avoid long downtime of the machine. These kinds of deductions allow for a predictive element in the diagnosis system. A corresponding expert deduction for this situation is: ***if*** *the temperature of power amplifier x is high* ***and*** *the temperature of power amplifier x is* ***not*** *too high* ***then*** *power amplifier x is about to fail soon.*

- Finally power amplifiers in certain power cabinet configurations are powered via a safety relay which shuts off if the machine is in an unsafe state for operators. Most expert deductions will assume that all power amplifiers should be operational at all times and any non-operational power amplifier has some kind of problem. This assumption is however broken by this safety relay, an additional deduction can help here. For example for power amplifier x which is powered via the safety relay in cabinet configuration y: ***if*** *the configuration of the power cabinet is configuration y* ***and*** *the safety relay is switched off* ***then*** *it is not a problem if power amplifier x is non-operational.* The newly derived fact that states if a power amplifier is expected to be operational or not can then be used by other deductions which would normally assume that all power amplifiers should be operational at all times.

2 | Literature survey

Frameworks and solutions for the automation of the traditionally human task of making diagnoses have extensively been researched. Automated diagnosis systems need to combine facts, such as values of measurements, with knowledge in order to infer new information and ultimately a diagnosis. This is a process that traditionally matches closely to the research area of Artificial Intelligence (AI), therefore most of the research in the literature finds its roots there.

Classically AI has been focussed on systems built by capturing and formalizing human expert knowledge to allow for automated reasoning, the so called expert systems. After the winter in the AI field, most interest has been in systems where machine learning is employed to build knowledge systems automatically.

2.1 Ontological engineering

Gómez-Pérez, Fernández-López and Corcho [2] give an introduction into the expertise of ontological engineering. *Ontological engineering* is defined as the process of development and maintenance of ontologies. They present a collection of definitions on the word *ontology* which all share that an ontology is a formal and explicit conceptualization of a domain or topic area. Usually it represents a conceptualization of the knowledge contained within a knowledge base, where one definition additionally defines that it must be hierarchical.

Ontologies are in essence the formalization of an interface to which knowledge bases need to conform in order to be usable by multiple systems. Ontologies can be a valuable resource when dealing with knowledge based systems where the information contained in their knowledge bases needs to be available in an abstract form. Examples include situations in which multiple systems need to operate on knowledge bases shared among them, or where domain experts from multiple disciplines need to cooperate closely on related knowledge bases.

2.2 System model based diagnosis

Travé-Massuyès [3] provides a survey of research into diagnosis as it is understood in the control and AI fields. Two main theory tracks are identified in these fields, called the Fault Detection and Isolation (FDI) and the Diagnosis (DX) tracks. The alignment and integration possibilities between the two theories is investigated. While both tracks do provide significant diagnostic capabilities, they are also both first principle model based which means that accurate models of the system under diagnosis are required.

Expert systems as described in Section 2.3 can also be seen as a model based system where the expert rules define the model. The model of an expert system is however not based on the system under diagnosis directly, but rather on the knowledge of a human expert. The model defined by the expert rules can therefore be seen as a model of the actual system under diagnosis. The expert rules could also be created by means of encoding the design criteria such as requirements and specifications of the system under diagnosis. The resulting set of expert rules would then more directly form a model of the system under diagnosis without the level of abstraction provided by

the human expert knowledge. Due to the broader spectrum of possibilities that expert systems provide, they will be discussed separately.

2.3 Expert systems

The subset of knowledge based systems built by formalizing human knowledge are known as expert systems. Usually the knowledge they contain is formalized in the form of a set of rules in which case the system is known as a rule based (expert) system. Puppe [4] gives an extensive introduction on the topic of expert systems. The characterization and history of the different types of expert systems are handled. The advantages and disadvantages of expert systems based on several different approaches are dealt with.

First-order predicate logic is introduced as the foundation of most rule based expert systems. Noted deficiencies for first-order predicate logic are the lack of expressive power, adequacy and efficiency, which are partly alleviated by the rule based systems built on top. The major pitfall for rule based expert systems according to Puppe is the "rule spaghetti" which can develop if a large set of expert rules is not structured and modularized well, leading to a lack of clarity in the rules.

Part of the solution to structuring large rule sets might be found in capturing the rules in objects or frames. Object-oriented structures can be employed to guide the structuring of expert rule sets. However, regular logic based inference engines are not able to handle objects directly. Structuring rules in this way therefore does impose additional processing of the rules and facts in the database before use. An extension to the object-oriented approach of structuring rules is to add constraints to the objects. Constraints represent relationships between objects. They can be implemented with tables, more rules or programmatically. While constraints allow for additional expressive power when an object-oriented structuring is employed, they do require the additional solving of a constraint system while processing the objects in the knowledge base.

Probabilistic reasoning and non-monotonic reasoning are introduced as ways to achieve expert systems in the face of uncertainty in the knowledge base. Probabilistic reasoning handles the uncertainty by attaching uncertainty factors to all rules in the knowledge base. The uncertainties may be derived from representative statistics or be estimated by experts. During the inference process the uncertainty factors are propagated. The result of the inference includes a resulting uncertainty factor representing the degree of trust in that particular result. The additional possibilities provided by adding uncertainty factors to the expert rules can seem very attractive to mark less reliable rules. However according to Puppe:

> "The great variety of models for probabilistic reasoning used in expert systems indicates that no model is really satisfactory. The main problem is the uncertain starting position. Even the data collection itself can hardly be made objective in the many fields of application, since it frequently depends essentially on human senses. ... For this reason most expert systems allow no uncertainties in the data collection, but if there are uncertainties based on experience they are represented by qualitatively different values."

Furthermore he notes that much of what is generally represented by probabilistic means can actually be represented explicitly. This means that support for probabilistic reasoning is generally not required, since the same information can almost always be represented without resorting to uncertainty factors. Lastly Puppe also hints at the — at the time quite new — method of Bayesian networks: "A new, promising, well-founded approach to probabilistic reasoning is the method of causal probabilistic nets (Bayesian nets)."

Non-monotonic reasoning on the other hand allows for plausible conclusions to be drawn from incomplete data, where conclusions can later be withdrawn or corrected based on newly gathered facts or other corrected conclusions. This is in contrast to normal reasoning, which is monotonic in the sense that a conclusion is permanent once it is drawn. In a monotonic reasoning scheme conclusions can therefore only be drawn once all underlying facts are completely known. While non-monotonic reasoning follows the real world reasoning of domain experts more closely, the formalisation of non-monotonic reasoning is both a difficult theoretical and practical problem. According to Puppe "... — various calculi have been developed but they are generally semi-decidable and of no use for implementation — ...". This means that for a non-monotonic reasoning driven diagnosis system, checking the syntax of the expert rules will generally be impossible, since a syntax error can lead to non-termination of the program. Even though the diagnosis system can operate flawlessly, given a correct set of expert rules.

Lastly temporal reasoning is introduced as a way to enhance expert rules with temporal information which allows the temporal information on symptoms to be used in drawing conclusions on the diagnosis. Augmenting the expert rules with temporal information is akin to augmenting them with uncertainty factors. However the required adaptations to the inference techniques more closely resemble the techniques required for handling constraints, such as constraint propagation.

Merritt [5] argues that while predicate logic, and in particular Prolog, is quite neglected in current expert system literature, it is actually highly applicable for implementing virtually any real world expert system. Next to the basic principles of creating an expert system in Prolog, the specialized knowledge representation and processing techniques required for industrial software implementations are also covered. Expert systems augmented with confidence factors in the expert rules are also handled extensively [5, Chapter 3]. Furthermore the book is no stranger to concrete examples emphasizing the explained concepts and theory.

Merritt also considers the user interface aspect of an expert system, but glosses over the format in which the knowledge base is stored. The knowledge base is assumed to be stored as native Prolog code. This is however a burden on domain experts if they want to create or update the knowledge base, since the semantic gap between the expertise and Prolog is still too large. Seipel, Nogatz and Abreu [6] introduce two novel methods of implementing a Domain Specific Language (DSL) in Prolog systems to overcome this semantic gap and avoid the need for a knowledge engineer to encode the expertise of the domain expert.

2.4 Bayesian networks

Bayesian networks are cited in the literature as a suitable foundation for diagnostic systems. Such systems based on Bayesian networks can either be derived from domain knowledge or be learned from actual failure occurrences, be they natural or artificial such as by fault injection. Zhang, Wang and Chakrabarty [7] have developed a diagnosis system based on basic Bayesian networks. They have opted for an approach where they learn the Bayesian network based on faults gathered by fault injection on otherwise functional boards. While the solution they have presented is self-learning and allows for automated diagnosis, the faults required for learning do have to be generated before the system can be put to use. Furthermore they indicate that several parameters have still to be optimized such as "... selecting output pins that are the most capable of distinguishing faulty modules, and choosing observation points (registers) to create the least-correlated fault syndromes in the presence of different faults." Both statements indicate domain knowledge about the system under diagnosis is still required to be able to select the best data sources for learning the Bayesian network.

A diagnosis system based on similar principles is described by Butcher and Sheppard [8]. In this paper they discuss the challenges of variations between assets of a fleet in the context of learning Bayesian networks. Variations between individual helicopters in a fleet of helicopters in continuing maintenance are taken as a concrete example in this paper. They note that a single fleet-wide model can provide sub-optimal diagnosis results in terms of accuracy. This less than optimal accuracy is due to the fact that a statistically significant, large, dataset is required to learn the Bayesian network correct relationships between faults and measurements. Correction techniques based on asset-specific conditional probability distributions are employed to alleviate this problem.

In the literature most Bayesian network based systems are designed to be learned instead of based on domain knowledge. Probabilistically augmented expert systems as discussed in Section 2.3 can however be considered to be Bayesian networks built by exploiting domain knowledge.

2.5 Machine learning based approaches

Ye, Zhang, Chakrabarty and Gu [9] comprehensively detail the need for automated diagnosis systems in the current era. The book is concerned with diagnosis, a term they use to refer to root cause isolation of faults just as is required in this project.

Different existing approaches are explained by Ye et al. First of all they mention Bayesian networks [9, Section 1.2.3.1] as discussed in Section 2.4. Without mentioning why Bayesian networks are not considered further, they continue to mention Artificial Neural Networks (ANNs) and Support Vector Machines (SVMs) as potential solutions [9, Section 3.3]. However next to the challenge of gathering the training data "...the training time becomes prohibitively long as the system complexity increases." Finally an approach based on Decision Trees (DTs) is mentioned [9, Chapter 4] which can be machine learned just like ANNs and SVMs but provides faster diagnoses. The difficulties regarding gathering enough training data remains however. This method is based on earlier work from the same authors [10].

Finally Ye et al. summarize the trade off between machine learned and rule based systems quite succinctly [9, Section 7.1]:

> "Knowledge acquisition is the major bottleneck for developing a usable diagnosis engine. In contrast, rule-based and model-based techniques can assist in the initial creation of the knowledge base, but the acquisition of such knowledge requires a technician's expertise and the knowledge may be biased or flawed if the board structure is not fully understood."

Ye et al. continue to present their novel approach combining SVMs and ANNs with machine learning techniques and Weighted Majority Voting (WMV). Their approach is based on machine learning both a SVM and an ANN, the outputs of both systems are then combined by a WMV system. The advantage of an ANN is the quality of its interpretation of the relationship between the measurements and the required diagnosis. The advantage of a SVM is that its solution is globally optimal and unique. This is not the case for ANNs, they can have multiple local maxima. Ye et al. claim that combining both approaches by means of WMV leads to a highly accurate diagnosis. The output of a SVM is unique and the WMV scheme selects a single diagnosis, their complete method therefore always provides a single diagnosis as result.

A fair number of approaches to the design of a diagnostic system are present in the literature. The advantages and disadvantages of the different approaches as well as the decision on an approach for the current project will be discussed in Section 3.

3 | Approach

In this section the design and conceptual approach to the diagnosis system is detailed. Based on the literature survey presented in Section 2, a number of approaches to the design of a diagnostic system have been considered. The conceptual design considerations leading to the final design of the diagnosis system software are explained below.

On the other hand the remaining two challenges in the realization of this project, adapting the firmware for the power amplifier controller and designing a glue layer to allow communication to the control rack software, do not require many design decisions. Their development is mainly guided by the interfaces they need to connect together and by implementation details and therefore they will not be discussed further.

An overview of the proposed diagnosis system is shown in Figure 3.1. On the left a failure is present in the power cabinet infrastructure. PCDT reads out diagnostic information such as voltages, currents and temperatures from the devices in a power cabinet. An important part of PCDT is an inference engine which contains the knowledge to make diagnoses based on the diagnostic information. The inference engine is modularly connected to the rest of PCDT, meaning it can easily be swapped for a different inference engine in the future. On the right a concrete diagnosis is provided by PCDT. Depending on the use-case scenario — *production test*, *power cabinet status* or *field service* — this diagnosis can contain a combination of status information, a concrete root cause diagnosis and low-level diagnostic information.

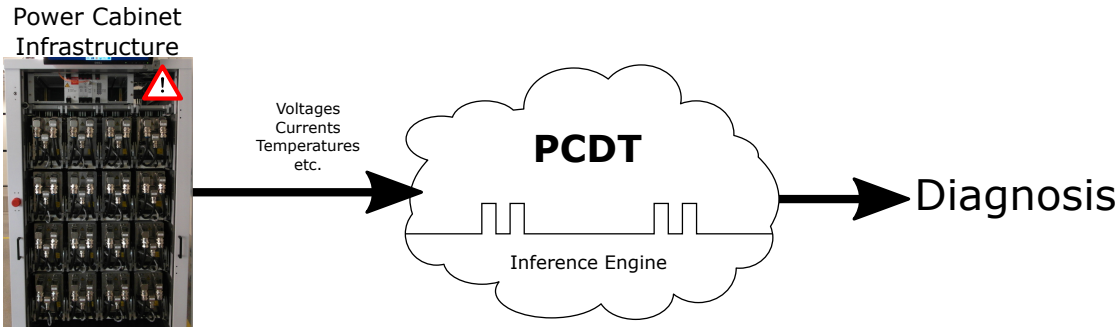

Figure 3.1: Overview of the proposed solution

3.1 Overview of considered approaches

Ontological engineering has briefly been considered as an approach to tackle the problem at hand. Ontological engineering is however a framework for formalizing the concept of knowledge bases. In itself it does not provide a complete solution. Therefore a separate implementation of a diagnosis system would still be necessary.

The conceptualization offered by ontological engineering is mainly of importance when knowledge bases need to be shared, either between systems or between domain experts from different

disciplines. Neither is the case for this project, therefore the conceptualization is not expected to be necessary, only the concrete knowledge base is deemed to be of importance.

A diagnosis system based on a model of the system under diagnosis by means of the type of models used in the FDI and the DX tracks has also been considered. These types of models require capturing all system behaviour at the chosen level of abstraction. Creating and maintaining accurate models for all components of the system under consideration in this project and of the system as a whole will incur significant development time. The burden of maintaining the models is also in direct contrast to Requirement 3. Therefore this approach is not considered suitable for inclusion in the current project. A related approach that might at least lessen the burden of maintenance of accurate models will be discussed for future work in Section 8.1.

Furthermore the method of employing Bayesian networks for diagnosis has also been considered as a solution. However there are some problems with this approach. Machine learned Bayesian networks suffer from the same data acquisition bottleneck as other machine learning based approaches. This will be further expanded upon in Section 3.2. Furthermore machine learned Bayesian networks can provide less accurate diagnosis results as a result of variations between individual assets. Butcher and Sheppard [8] do present correction techniques for this, which can be applicable for a system designed for recurring diagnosis on the same asset, such as the example discussed in their paper. In the case of the power cabinet infrastructure the number of service actions per asset is generally less than half a dozen and often even only once or twice at most. This means that incorporating asset-specific data is not a suitable way to increase diagnostic accuracy.

On the other hand if the Bayesian network is created by means of manually capturing human knowledge, the resulting system is in essence a rule based expert system augmented with probabilistic mechanisms. In this case a rule based expert system is considered a simpler approach, since it can initially be implemented without probabilistic mechanisms. These can then later be added if they are deemed necessary, at which point the system can be regarded as a Bayesian network. This consideration will be described in more detail in Section 3.4.

Finally a knowledge based system is considered as a possible approach. Since a knowledge based system does not appear to share the downsides of the other approaches, it will be the approach of choice.

3.2 Comparison between human knowledge and machine learning based systems

Based on the literature survey presented in Section 2, two major types of approaches exist regarding the design of a knowledge based system. The first type of system that can be designed is an expert system where human knowledge is manually converted into a form suitable for automated diagnosing. The alternative is a system based on machine learning where the diagnoses are learned automatically by the system itself.

The systems based on machine learning have major benefits in that they do not require the manual formalization of human knowledge and that they can learn new diagnoses on the fly with less effort. However machine learning systems ideally would require access to all diagnostic information from the power amplifiers, which is a massive amount of information. Furthermore this information is not logged in the historical data which means that learning of these types of systems can only be performed on new failures. This is in contrast to Requirement 7, since it means the diagnosis system can only be used after a substantial number of new failures have occurred.

Systems based on ANNs or SVMs also need access to all the same information during the diagnosing process itself, which might introduce considerable latency in this process due to the communication links involved. Due to the limited data rate of some communication links between the power amplifiers and the automated diagnosis system these systems run the risk of violating Requirement 6. One can compare this to a human knowledge based system which only needs access to the pieces of information that are manually specified. This argument is less applicable to DT based mechanisms which also only need access to certain pieces of information one at a time during the diagnosing process.

The method presented by Ye et al. [9] is based on the premise that it will be employed during the production phase of products and therefore it makes the assumption that external testing systems are available. On the contrary the envisioned setting for this project aims to also be operational in the phase after product manufacturing without external testing systems. This means that in order to satisfy Requirement 2, only information sources internal to the system under diagnosis are available.

Furthermore the presented approach is suitable for situations in which a large amount of data on failure modes is available, like is the case with high volume consumer products as focussed on in this book. Within Prodrive the focus is more on low volume but high value production which means that the amount of time and money spent per product is generally higher on average, relative to the size of the total investment. While a large amount of data is available from operational machines in the field, this data does not contain information about failures and is therefore not suitable for learning a root cause analysis system. This means that less data is available for machine learning a diagnosis system. This combined with the fact that the power cabinet infrastructure is a highly complex system, this complicates setting up machine learning based approaches.

The learning bottleneck for machine learning based systems can be alleviated somewhat by means of fault injection. However this mechanism requires considerable effort and remains artificial since only faults expected by humans can be injected. Furthermore only root causes considered during the fault injection procedure will be learned. A system learnt by means of fault injection therefore runs the risk of not providing coverage for faults which will actually occur in the field. Note that this same risk is also present in any human knowledge based system.

Like the machine learning based systems, the human knowledge based systems can also be adapted to cope with new failure modes. However while machine learning approaches can do this automatically, the human knowledge based approach will need human intervention to add knowledge about new failure modes. An advantage of the human knowledge based approach is however that the expert rules added by domain experts can also capture the 'why' of a diagnosis. The expert rules can encode information such as: syndrome x combined with syndrome y points to root cause z. While DT based methods also capture the list of syndromes leading to a diagnosis, the human knowledge based systems actually capture the reasoning of the domain expert.

This also means that causalities between a root cause and the symptoms it causes can be captured explicitly. Capturing causalities is inherently impossible in a machine learned system, since causalities cannot be derived from data alone. As Aldrich notes [11], correlation does not imply causation.

Furthermore in a human knowledge based system, expected future failure modes can be encoded without having physical occurrences of that failure mode present. Machine learning systems on the other hand can only be trained on failure modes that have actually occurred, either as a result of fault injection or by natural means.

Considering the advantages and disadvantages of both types of systems it is decided to design a human knowledge based system to fulfil the requirements for this project. It should be noted

however that a deep AI machine learning based approach definitely offers an attractive alternative perspective. It is however considered out of scope for this project since not enough training data is present as of yet. Future research could include investigating how to gather enough training data to pursue such an approach. This would also include determining if the communication constraints due to the limited data rate of some links within the power cabinet infrastructure do not impose unworkable latencies in the diagnosis process.

3.3 Inference engine

As mentioned by Merritt [5], Prolog is suitable as an inference engine for implementing expert systems, including automated diagnosis systems. Furthermore Seipel et al. [6] note that it is possible to define a DSL in Prolog. This allows the expert rules to be written down in an intuitive way, which is necessary in order to satisfy Requirement 5. Therefore it has been decided that Prolog as inference engine in combination with a DSL defined in Prolog will be used to provide an intuitive interface to the domain experts, being service engineers and quality personnel.

While machine learning based approaches are considered out of scope for this project, PCDT will be set up in a modular way. This will ensure that in the future it will be possible to pull out the Prolog inference engine from PCDT and plug a machine learning based inference engine such as an ANN back in its place. A suitable mechanism to allow this future inference engine to be trained has to be devised at that time, and will also be considered out of scope for the current project.

3.4 Structure of the expert rules

As shown by Merritt and Seipel et al. [5, 6] the expert rules in an expert system are often structured as *if-then* rules. The rules can commonly express statements in the form of: ***if*** *some observation* ***then*** *some conclusion* where the conclusion is usually a single deduced fact and the observation is a boolean expression combining several facts. This same form can also be seen in the example deductions presented in Section 1.7. The *if-then* structure is therefore considered to be an appropriate format for defining the expert rules.

It is expected that the in-house expert knowledge can be formalized easily without relying on probabilistic information. Furthermore as Puppe [4] notes, much of what is generally represented by probabilistic means can actually be represented explicitly, without resorting to probabilistic means. Therefore no probabilistic mechanisms are expected to be required in the expert rules. If during implementation, prototyping or verification probabilistic mechanisms turn out to be valuable after all, they can easily be implemented since the Prolog core will be flexible enough to support this. Furthermore since the size of the rule set is expected to remain manageable, no explicit object-oriented or constraint based rule structuring as referenced by Puppe [4] is deemed necessary. If necessary the rule set will instead be structured by dividing the rules into multiple knowledge bases, or files, possibly combined with conditional consultation.

3.5 Conclusion on approach options

Finally we decided to opt for a diagnosis system designed as a human knowledge based system by means of expert rules. The expert system will be based on Prolog as the inference engine with the expert rule syntax defined by a DSL implemented in Prolog. In the end an application will

have been developed that can automatically detect all known failure modes in a power cabinet provided that adequate expert rules have been defined.

Because all known failure modes can be detected automatically, the application will allow service engineers both at Prodrive and at the customer to spend their time investigating new failure modes, which can then be added to the expert rules for future automatic detection. Furthermore the application will help reducing overhead in terms of machine downtime, time spent on failure investigations and the number of modules without faults being unnecessarily shipped between Prodrive and the customer.

4 | Inference engine theory

Lucas and van der Gaag treat the accepted formalisms and methods that underlay expert systems as used in the field. Of special interest for the design of the automated diagnosis system in this project is their discussion on propositional logic and predicate logic [12, Chapter 2]. Both logic formalisms will be handled in the following sections as well as how Prolog, used to implement the inference engine in PCDT, is based on them.

4.1 Propositional logic

Propositional logic can be thought of as the most elementary form of logic, it is therefore sometimes also known as *zeroth-order logic*. Propositional logic is a formalism that allows for the expression of and the reasoning about statements that are either *true* or *false*. These statements are called *propositions* and are either *composite propositions* or *atomic propositions*, *atoms* for short. Atoms are simple propositions that can not be divided into smaller expressions, they are usually represented by uppercase letters such as P or Q. Composite propositions consist of atoms combined with *logical connectives*. They can also be given a name just like atomic propositions. Propositional logic knows five logical connectives:

negation:	$\neg$	(not)
conjunction:	$\wedge$	(and)
disjunction:	$\vee$	(or)
implication:	$\rightarrow$	(if then)
bi-implication:	$\leftrightarrow$	(if and only if)

Negation is written as a unary prefix operator, all other logical connectives are written as binary infix operators. The order of binding is as listed with negation binding strongest and bi-implication having the weakest binding. Additionally parenthesis may be used to override the order of the operators in a composite proposition.

The truth or falseness of a proposition is called its *truth value*. The meaning of the logical connectives in terms of the truth value of two arbitrary propositions F and G is given in Table 4.1.

A mapping from all propositions in a formula to their respective truth values is called an *interpretation* of the formula. If an interpretation of a formula causes the overall formula to have a truth value of *true* it is known as a *model* for the formula. Notice that a formula containing n

F	G	$\neg F$	$F \wedge G$	$F \vee G$	$F \rightarrow G$	$F \leftrightarrow G$
true	*true*	*false*	*true*	*true*	*true*	*true*
true	*false*	*false*	*false*	*true*	*false*	*false*
false	*true*	*true*	*false*	*true*	*true*	*false*
false	*false*	*true*	*false*	*false*	*true*	*true*

Table 4.1: Meanings of the logical connectives

Overall truth value over all interpretations	always *true*	sometimes *true*, sometimes *false*	always *false*
Example	$P \vee \neg P$	$P \vee Q$	$P \wedge \neg P$
Validity	valid	invalid	invalid
Satisfiability	satisfiable	satisfiable	unsatisfiable

Table 4.2: Categorization of the interpretations of a formula

different atoms has 2^n possible interpretations. An arbitrary formula can be categorized based on two criteria regarding its overall truth value over all possible interpretations; *validity* and *satisfiability*. The relation between the overall truth value and the corresponding category is shown in Table 4.2. A valid formula is often called a *tautology* while an unsatisfiable formula is often called a *contradiction*.

Formulas can be transformed into different formulas expressing the same relation between their overall truth value and the truth values of its atoms by means of so called *laws of equivalence*. The notation $F \equiv G$ states that F and G are *logically equivalent* which means that the truth values of F and G are the same under all interpretations.

Given that F and G are arbitrary formulas, propositional logic knows eleven laws of equivalence:

$\neg(\neg F) \equiv F$	(law of double negation)
$F \vee G \equiv G \vee F$	(commutative law, 1/2)
$F \wedge G \equiv G \wedge F$	(commutative law, 2/2)
$(F \vee G) \vee H \equiv F \vee (G \vee H)$	(associative law, 1/2)
$(F \wedge G) \wedge H \equiv F \wedge (G \wedge H)$	(associative law, 2/2)
$F \vee (G \wedge H) \equiv (F \vee G) \wedge (F \vee H)$	(distributive law, 1/2)
$F \wedge (G \vee H) \equiv (F \wedge G) \vee (F \wedge H)$	(distributive law, 2/2)
$F \rightarrow G \equiv \neg F \vee G$	
$F \leftrightarrow G \equiv (F \rightarrow G) \wedge (G \rightarrow F)$	
$\neg(F \vee G) \equiv \neg F \wedge \neg G$	(De Morgan's law, 1/2)
$\neg(F \wedge G) \equiv \neg F \vee \neg G$	(De Morgan's law, 2/2)

Given an arbitrary formula H and a formula I derived from H by only applying these laws of equivalence, it is guaranteed that H and I are logically equivalent. This means that the truth values of H and I are the same for every interpretation of both formulas. That is to say the laws of equivalence are sound. The laws of equivalence are also complete, which means that given a formula J all formulas that are logically equivalent to J can be derived by applying the laws of equivalence. These equivalences prove to be useful when reasoning about propositional logic using resolution as explained in Section 4.1.2.

4.1.1 Conjunctive Normal Form

Reasoning about formulas turns out to be easier if they are of a special form called *Conjunctive Normal Form (CNF)*. In order to understand the form that a formula in CNF takes, it is necessary to introduce some terminology. A *positive literal* is an atom and a *negative literal* is the negation of an atom, collectively known as *literals*. A *clause* is a disjunction of unique literals and a formula in CNF consists of a conjunction of clauses. This means that the overall structure of a formula in CNF is as follows, where A_1 through A_n, B_1 through B_m and C_1 through C_k are literals:

$$(A_1 \vee A_2 \vee \cdots \vee A_n) \wedge (B_1 \vee B_2 \vee \cdots \vee B_m) \wedge \cdots \wedge (C_1 \vee C_2 \vee \cdots \vee C_k)$$

Every propositional formula can be transformed into an equivalent formula in CNF by means of applying a sequence of the laws of equivalence. However for some formulas this causes an exponential explosion in the length of the formula. Other transformations that do not preserve equivalence but do preserve satisfiability exist, such as the Tseitin transformation [13]. These transformations only increase the size of the formula linearly, but they do introduce new literals.

A formula in CNF is often written as a set of clauses, of which all clauses must have a truth value of *true* in order for the complete formula to have a truth value of *true*. Note that this statement exactly corresponds to the meaning of a conjunction of clauses. In this way the previous example can also be written as:

$$\{A_1 \vee A_2 \vee \cdots \vee A_n, B_1 \vee B_2 \vee \cdots \vee B_m, \cdots, C_1 \vee C_2 \vee \cdots \vee C_k\}$$

4.1.2 Resolution

Reasoning about formulas happens by means of *inference rules*. An inference rule is a transformation from one set of formulas to a different set of formulas. Generally it is most useful if an inference rule is *sound* and a set of inference rules in *complete*. Soundness of an inference rule means that each formula derived by means of application of the rule is a logical consequence of the original formula. This is equivalent to stating that truth is preserved by application of a sound inference rule. Completeness of a set of inference rules on the other hand means that all logical consequences of the original formula can be derived by applying a chain of rules from that set. A chain of applications of inference rules is usually called a *derivation* or a *deduction*.

Resolution is a inference rule that is both sound and complete, however the name *resolution* is also often used to mean the chained application of this rule. The resolution rule is only applicable on formulas in CNF. However as all propositional formulas can be converted into CNF this does not detract from its general applicability. Resolution is normally used to prove that a formula G can be derived from a set of known *true* clauses S. Proving that G is a logical consequence of S is equivalent to proving that $S \cup \{\neg G\}$ is unsatisfiable. This means that resolution can be seen as a *proof by contradiction*. In general $\neg G$ is not in CNF it is therefore transformed into $\hat{G}$ in CNF and the union $W = S \cup \hat{G}$ is formed. Proving G then proceeds by means of resolution on W.

Each resolution step starts with the selection of two clauses C_1 and C_2 from W. One literal that is positive in C_1 and negative in C_2, or vice versa, is removed from both C_1 and C_2. The disjunction of the remaining literals from both clauses is added to W as a new clause. The correctness of this assertion can most easily be shown by example. Suppose that $C_1 = A \vee B$ and that $C_2 = \neg A \vee C$. In case A is *true* it must be the case that C is *true*. Conversely, in case A is *false* it must be the case that B is *true*. In any case, regardless of the truth value of A, either B must be *true* or C must be *true*, or both. This simultaneously explains why the complementary literals may be ignored and why the disjunction $B \vee C$ is a logical consequence of the conjunction $C_1 \wedge C_2$.

Suppose now that after ignoring all complementary literals from both C_1 and C_2, no literals remain in either C_1 or C_2. In this case it is said that the *empty clause*, usually depicted as $\square$, has been derived. The derivation of the empty clause is relevant since it proves that W is unsatisfiable, which in turn proves that G is a logical consequence of the given clauses in S, the original goal. This can be verified by considering that a disjunction over an empty set of literals has a truth value of *false*. The set of clauses therefore becomes a conjunction containing a member with truth value *false*. This causes the truth value of the complete formula to be *false* irrespective of interpretation, which is by definition the same as unsatisfiable. Since satisfiability is preserved under resolution the original set of clauses W is also unsatisfiable. The chain of resolution steps required to arrive at a derivation of the empty clause is known as a *refutation*.

Note that in this description of resolution one aspect has been glossed over. Each resolution step selects two clauses from W. In which way these two clauses should be selected has not been discussed yet. However the procedure of selecting the clauses has a profound effect on the number of resolution steps required to arrive at a refutation and therefore on overall resolution speed in an actual implementation. For resolution in propositional logic multiple search strategies exist of which the Davis-Putnam (DP) algorithm [14] is most famous, later extended to the DPLL algorithm [15].

4.2 Predicate logic

The basic constructs making up propositional formulas are atoms, single entities that represent either *true* or *false*. A consequence of this fact is that it is impossible to express general statements about similar scenarios. A more expressive form of logic that can be seen as an extension of propositional logic is *predicate logic*, also sometimes called first-order logic. Predicate logic does allow for the expression of such general statements.

In predicate logic the atoms of propositional logic are extended to form *predicate symbols* or *predicates*, a single predicate symbol is still called an atom. Predicates are usually denoted by an uppercase letter followed by a number of *arguments* separated by commas and enclosed by parenthesis. The number of arguments is called the *arity* of the predicate. Predicates with an arity of 0 drop the parenthesis. The arguments to a predicate are either *variables* or *function symbols*. A variable is usually represented by a lowercase letter from the end of the alphabet, such as x, y or z. A function symbol consists of a lowercase letter from the beginning through roughly halfway into the alphabet followed by a number of arguments separated by commas and enclosed by parenthesis. Function symbols with an arity of 0 drop the parenthesis and are also called *constants*. A *term* is either a constant, a variable or a function symbol. Arguments to predicates and function symbols are always terms. Note that this means that function symbols can contain other function symbols in their arguments. An example of a fairly complex predicate would be $P(f(x, y), c, g(z))$, where P is a predicate, f, c and g are function symbols and x, y and z are variables.

Next to the logical connectives of propositional logic, predicate logic introduces two *quantifiers*. The first is the *existential quantifier* written as $\exists$. Given that x is a variable and F is a formula, the meaning of $\exists xF$ is 'there exists an x such that F holds'. Next there is the *universal quantifier* written as $\forall$. Given that x is a variable and F is a formula, the meaning of $\forall xF$ is 'for all x it is so that F holds'. In both instances all occurrences of x in F are said to be *bound* by the quantifier. Whenever a variable is not bound by a quantifier it is called a *free variable*. The quantifiers bind more strongly than the logical connectives but parenthesis can be used to alter the order of operations.

In much the same vein as atoms are extended to predicates also the notion of an interpretation needs to be extended. Instead of a mapping from atoms to truth values as in propositional logic, an interpretation in predicate logic is a mapping from *ground* predicates to truth values. A ground predicate is a predicate that does not contain variables in its arguments, either directly or as arguments to function symbols. Note that this means that an interpretation consists of a mapping from predicates to truth values for each possible *assignment* of constants or ground function symbols to variables within those predicates.

Lastly additional laws of equivalence governing the quantifiers are introduced in addition to the laws of equivalence carried over from propositional logic:

$$\neg\exists x P(x) \equiv \forall x \neg P(x)$$
$$\neg\forall x P(x) \equiv \exists x \neg P(x)$$
$$\forall x(P(x) \wedge Q(x)) \equiv \forall x P(x) \wedge \forall x Q(x)$$
$$\exists x(P(x) \vee Q(x)) \equiv \exists x P(x) \vee \exists x Q(x)$$
$$\forall x P(x) \equiv \forall y P(y)$$
$$\exists x P(x) \equiv \exists y P(y)$$

While the augmented set of the laws of equivalence is still sound for predicate logic, it is no longer complete as shown by Gödel [16]. This is due to the inherent *undecidability* of predicate logic caused by the fact that it is expressive enough to be self-referential.

4.2.1 Clausal Normal Form

The notion of CNF as introduced for propositional logic is also extended to incorporate all aspects of predicate logic. The corresponding canonical form is called the *Clausal Normal Form*. Note that since its abbreviation is the same as for Conjunctive Normal Form, the term *clausal form* is used here to make the distinction more clear.

The definition of a CNF clause is extended to be a disjunction of predicates of which all occurring variables are bound by a universal quantifier. This means that a clause in clausal form takes the following form, where P_1 through P_m are literals containing the variables x_1 through x_n:

$$\forall x_1 \cdots \forall x_n(P_1 \vee \cdots \vee P_m)$$

Just as CNF is just as expressive as unrestricted propositional logic, the clausal form is just as expressive as unrestricted predicate logic. This means that it is possible to write any predicate logic formula in clausal form [17]. Just as in propositional logic, a formula in predicate logic in clausal form is often written as a set of clauses of which the conjunction over all elements represents the complete formula.

Horn clauses

Note that sorting literals in a clause based on whether they are negated or not leads to a clause of the form:

$$\forall x_1 \cdots \forall x_n(A_1 \vee \cdots \vee A_m \vee \neg B_1 \vee \cdots \vee \neg B_k)$$

This can be rewritten by means of the laws of equivalence into:

$$\forall x_1 \cdots \forall x_n(B_1 \wedge \cdots \wedge B_k \rightarrow A_1 \vee \cdots \vee A_m)$$

In reversed order and with implicit universal quantification of all variables this can be written as:

$$A_1, \cdots, A_m \leftarrow B_1, \cdots, B_k$$

This is the more conventional notation in logic programming. Note that in this notation the commas in $A_1, \cdots, A_m$ represent disjunctions while the commas in $B_1, \cdots, B_k$ represent conjunctions.

A special case occurs when a clause has at most a single positive literal, it is then of the form:

$$A \leftarrow B_1, \cdots, B_k$$

or, where $\perp$ represents *false*:

$$\perp \leftarrow B_1, \cdots, B_k$$

Such a clause is called a *Horn clause*. When a Horn clause contains exactly one positive literal it is called a *definite clause*. Prolog is based around Horn clauses as will be discussed in Section 4.3.

4.2.2 Unification

The procedure to select two clauses for a resolution step is relatively trivial in propositional logic where it depends solely on the complementarity of the literals in question. However in predicate logic resolution needs to take place over predicate symbols which complicates the matter. Take for instance a resolution step over two clauses containing the literals $P(x)$ and $\neg P(a)$, resolution can take place since $P(x)$ and $P(a)$ *unify*. *Unification* is the process of establishing whether two predicate symbols could have the same meaning. Where resolution in propositional logic requires a check on the equality of atoms, resolution in predicate logic requires a check for unification of predicate symbols.

Unification is expressed in terms of finding a *substitution* of variables that makes both symbols under consideration syntactically equal. Unification can be divided into three categories based on the types of the terms involved:

- A variable can be unified with an atom, a function symbol or another variable. However a variable cannot be unified with a function symbol that has the variable as one of its arguments, either directly or as an argument to a nested function symbol. This type of unification will constitute an element of the substitution.

- Two atoms can be unified if they are identical.

- Two function symbols can be unified if their names and arities match and if all corresponding arguments can be unified simultaneously. Note that this type of unification introduces recursive behaviour in the unification process.

A substitution that makes two symbols syntactically equal is called a *unifier*. A unifier that has the lowest number of elements is called a *most general unifier*. Most actual implementations of the unification algorithm are based on finding this most general unifier.

Occurs check

A variable cannot be unified with a function symbol that contains the variable. The check for this condition is called the *occurs check*. Due to the recursive nature of function symbols the occurs check is quite time consuming in actual implementations. Therefore certain implementations disable the occurs check by default and rely on the user to avoid the creation of cyclic terms. Usually a way to perform sound unification with inclusion of the occurs check is provided for use in cases where it is required. This is an important aspect to be aware of since failure to observe the precautions leads to unsound resolution which can lead to incorrect results.

4.2.3 Resolution

Due to the additional requirement of unification during the resolution process, the *control strategy* required for resolution in predicate logic is more involved than the search strategies for resolution in propositional logic. Multiple control strategies for resolution in predicate logic have been developed over time.

The control strategy most of interest since it is used by Prolog is *Selective Linear Definite (SLD) resolution*. SLD resolution is an instance of *input resolution* which itself describes a subset from the set of control strategies known as *linear resolution*. A linear resolution strategy is a control strategy where in each step the result of the previous step is combined with a new clause. Since one side of the input to the current step is always the output of the previous step, this set of strategies does not branch out during a derivation. This leads to a derivation structure that visually appears more linearly, giving it its name. Input resolution further restricts the other side of the input to the current step to come from the original set of clauses, specifically excluding the clauses generated during the resolution process. SLD resolution further augments input resolution with the selection rules required to determine which of the input clauses to select for the next resolution step.

SLD resolution is a very efficient resolution strategy, however it is only complete for Horn clauses and not for the clausal form of predicate logic in general. All resolution strategies applicable to the clausal form of logic in general inherently face the risk of a combinatorial explosion, which is why the dependence on Horn clauses is deemed an acceptable compromise. This also explains why Prolog is built around Horn clauses.

4.3 Prolog

The logic programming language of Prolog is the materialization of an inference engine for predicate logic. While the theory underlying Prolog closely matches the predicate logic as detailed in Section 4.2, some differences in notation and terminology exist. A program in Prolog is written in terms of relations, represented as *facts* and *rules*. Rules are written as reversed implications and take the form of Horn clauses. The reverse implication form of the rules better matches the goal driven inference process performed by SLD resolution. Rules are written with the implication arrow replaced with the character sequence `:-` and each rule is terminated by a period, forming the full form of a rule:

```
Head :- Body.
```

This rule means that `Head` is *true* if `Body` is *true*. A fact is a special case of a rule asserting that `Head` always holds, it takes the form:

```
Head.
```

A *goal* or a *query* is the clause posed by the user to initiate a computation within Prolog, it takes the form:

```
:- Body.
```

Each rule of a Prolog program is referred to as a *clause* and the set of clauses sharing a head with the same name and arity is referred to as a *predicate*. The body of a rule consists of a conjunction of predicate symbols represented as a list of predicate symbols separated by commas. Support is also available for disjunctions represented by semi-colons as an alternative to specifying the clauses separately.

Prolog is built on a single data type named the *term*, a term can either be an *atom*, a number, a variable or a *compound term*. An atom is what is called a constant in Section 4.2 and a compound term is what has been called a function symbol up until now. Prolog provides special support for compound terms representing lists and strings of characters.

A final difference between the notation in Prolog and the one used in Section 4.2 is the letter casing of predicates and variables. While predicate logic literature usually typesets predicates with an uppercase first letter, in Prolog they are written fully lowercased, optionally with underscores. This is due to the fact that anything starting with an uppercase letter in Prolog is interpreted to be a variable. This is again in contrast to the usually lowercase typeset variables found in literature.

4.3.1 Extra-logical features

In addition to the purely logical computation provided by Prolog, certain extra-logical features are also provided to make programming in Prolog easier. The first extra-logical feature of interest is made possible by so called *chronological backtracking*. The SLD resolution strategy searches for solutions in a depth-first manner. This means that it only gives up on a potential solution if it is certain that that solution leads to a failure. Once this happens the algorithm backtracks and tries to prove another potential solution. The order in which solutions are tried is prescribed by the order of the clauses as listed in the program. This fact can be exploited for performance or brevity reasons if only the first or the first few solutions are of interest. Since this makes the order of clauses in a Prolog program significant, Prolog is not a purely declarative language.

The second extra-logical feature of interest is the so called *negation as failure*. While one could assume it to be easily specified that one of the constituents of a clause should not hold, this would actually lead to a second positive literal in the clause. This is a problem since the clause is then no longer a Horn clause and so it has become unfit to be used by SLD resolution. Therefore Prolog does not provide the classical logical negation $\neg Goal$, but provides negation as failure instead, written as `\+ Goal`. Negation as failure works by proving the goal it is applied to. If the goal cannot be proven the negation is assumed to hold. The word *assumed* is used deliberately in the last sentence, since this assumption only holds if the so called *closed-world assumption* holds. The closed-world assumption represents the presumption that a *true* statement is also known to be *true*, meaning it is specified in the program. Another way of phrasing this is; that what is not known to be *true* must be *false*, a lack of knowledge implies falseness. It should be noted however that the application of negation as failure is only sound if the goal of the negation is ground.

4.3.2 Definite Clause Grammars

Parsing of a DSL requires working with its defining grammar extensively. Working with a grammar is made easy by Prolog's extensive support for *Definite Clause Grammars (DCGs)*. DCGs represent the grammar of a language as a set of definite clauses in predicate logic, giving them their name. The set of clauses making up a DCG define the validity of a sentence in that grammar. Recognition and parsing of sentences is therefore reduced to the proving of a statement, which is already the underlying computation mechanism of Prolog.

The syntax of a DCG in Prolog is similar to the syntax of a normal rule except that the character sequence separating the head from the body of the rule is replaced with `-->`. A full DCG therefore looks like this:

```
Head --> Body.
```

In order to understand how DCGs work internally, first the notion of *difference lists* needs to be explained. A difference list is a pair of lists L_1 and L_2 such that their difference represents the list prefix under consideration D. The pair of lists L_1 and L_2 is formed such that D appended with L_2 is equal to L_1. Since Prolog allows the tail of a list to be represented by a variable, difference lists allow the prefix D to be represented without the rest of the list being known. Difference lists are used instead of directly specifying partial lists purely for reasons of efficiency.

DCGs provide a convenient interface for working with lists in Prolog, however they are not much more than syntactic sugar to hide the two difference list arguments normally required for list construction and analysis. DCGs are expanded by the Prolog preprocessor into regular predicates in a fashion similar to macro substitution in other programming languages. This rewriting adds the two difference list arguments required to represent the part of the grammar matched by the DCG rule.

DCGs can be used to validate or parse a sentence according to a grammar, but they can also be used to generate all sentences that are valid under the specified grammar. DCGs are therefore often used to create both parsers and list generators.

Just as arguments can be added to the predicate symbol of regular Prolog predicates, additional arguments can also be added to DCG predicates. These extra arguments allow for the insertion and extraction of additional information during the parsing or generation process. One notable use of these extra arguments is to relate the parsing of a sentence to the Abstract Syntax Tree (AST) it represents.

4.3.3 Use of Prolog as inference engine

It would be possible to write a custom inference engine in an imperative language, for example to lessen the burden of communication between PCDT and the cabinet under diagnosis. However extreme care would need to be taken in order to ensure no infinite loops occur and soundness of the inference process would not be guaranteed automatically. Furthermore implementing a custom inference engine is akin to reinventing the wheel, since the inference process as employed by Prolog matches so closely to the inference process required by PCDT. Additionally Prolog has a strong track record as a logic programming language dating back to 1972 with a rigorous mathematical basis [18], ensuring that the inference engine has a strong foundation.

5 | Design of the expert system

An existing framework providing abstraction of the hardware in a cabinet filled with register based devices is being developed at Prodrive separately from PCDT, called the Cabinet Hardware Abstraction Layer (CabinetHAL). CabinetHAL finds its roots in improving the testability of electronics cabinets, a different type of cabinet. Since the inception of CabinetHAL, support has been added for all components in the power cabinets due to the desire to use its features in new production tests for the power cabinet infrastructure. CabinetHAL is written in the form of a C++ library which can be linked into production tests.

PCDT is integrated into CabinetHAL which provides several advantages. First of all it allows PCDT to make use of the abstraction of device registers and the way to communicate to devices internally present in CabinetHAL. Secondly it allows the integration of PCDT into the new production tests for the power cabinet infrastructure, which already use CabinetHAL, with minimal effort.

The integration of PCDT into CabinetHAL is depicted graphically in Figure 5.1. The CabinetHAL Board Support Package (BSP) is the part of the test software running on the control rack to allow for communication between the test PC and the power cabinet. The hardware and communication component within CabinetHAL provides a view of the devices within a cabinet with the communication links between them abstracted away. This means that the fact that certain devices are connected via other devices is no longer visible. The register file component provides a well defined interface to the registers available within the devices and is used by PCDT to gain access to the required information. The interface provider component of CabinetHAL uses the register file interface to provide the existing functionality of CabinetHAL.

CabinetHAL is used by first creating a model of the cabinet under test in the source code of the production test. This is done by means of an object hierarchy representing the cabinet as one object, containing other objects. Each object represents one device in the cabinet. For the electronics cabinets where CabinetHAL originated, this is required to be recursive, so devices can contain other devices. In order to minimize the amount of effort required in using PCDT, while still having access to all required information from within PCDT, this cabinet model is used as the interface for PCDT as well. This allows using PCDT to be as simple as two lines of code in a production test. One instructing which set of expert rules to use, the other asking PCDT to generate a diagnosis.

A detailed view of the PCDT component as designed in this project is displayed in Figure 5.2. Its individual components are detailed below.

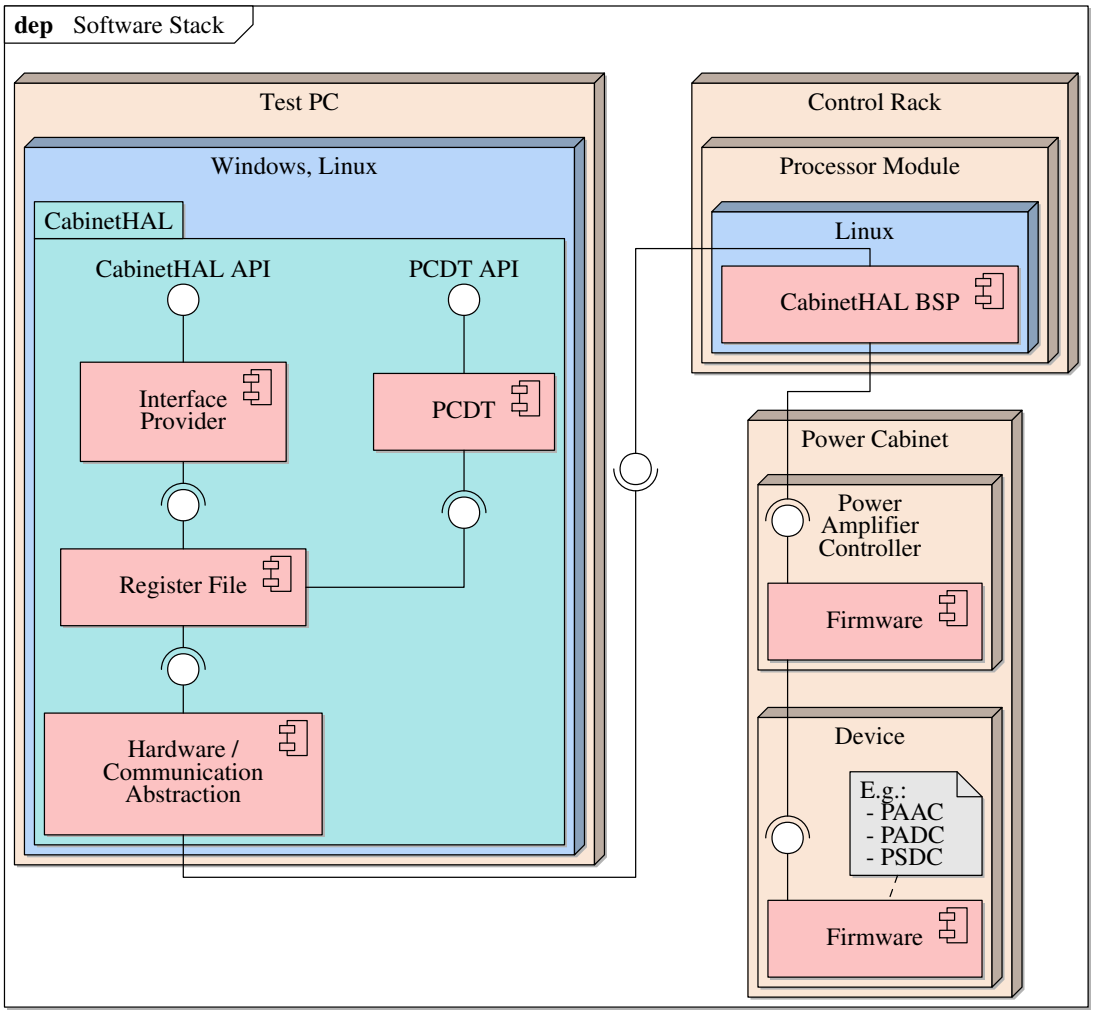

Figure 5.1: Unified Modeling Language (UML) deployment diagram of all major components of the PCDT architecture. Note that this thesis focusses on the PCDT component, the other components are developed separately or already exist.

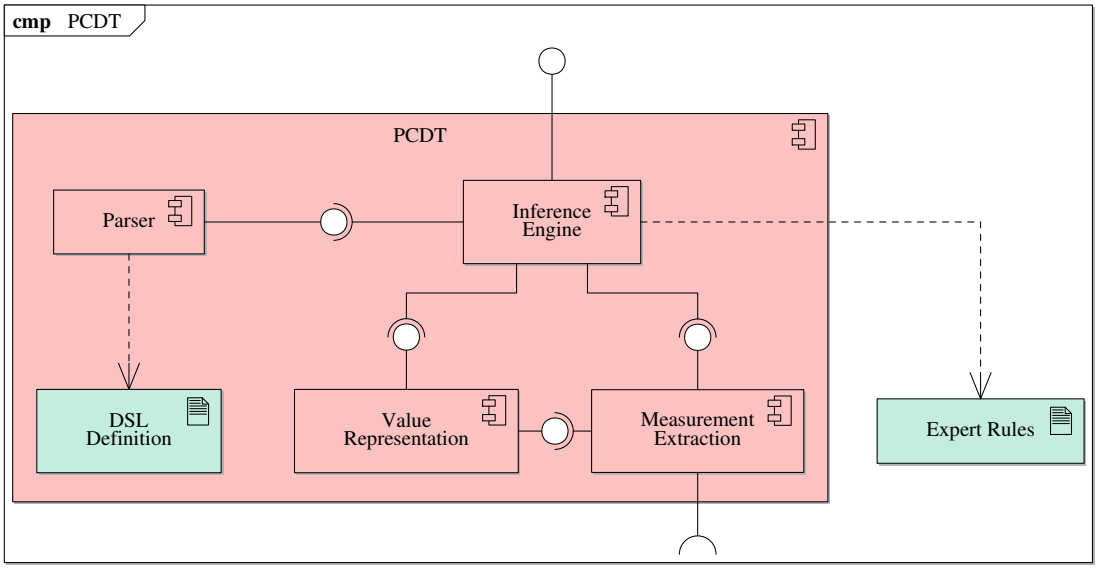

Figure 5.2: Unified Modeling Language (UML) component diagram of the PCDT component

5.1 Inference engine

The inference engine is the component that ties the parser, the value representation and the measurement extraction together to provide the functionality required from PCDT as shown in Figure 5.2. The parser component is described in Section 5.3, the value representation and the measurement extraction are handled in Section 5.5 and Section 5.6 respectively.

The human knowledge required by PCDT is defined by means of expert rules written in a custom DSL, which is described in Section 5.2. Upon loading a knowledge base, the textual expert rules are parsed into an AST, after which they are stored until a request for a diagnosis is made. This allows for loading multiple sets of expert rules, or even individual rules one at a time.

Once one or more sets of expert rules have been loaded and a diagnosis is requested, the stored ASTs are combined into a single larger AST which is then traversed top down. This is implemented by means of a DCG, operating as a list generator. Once a certain diagnosis has been proven, the textual representation of that diagnosis is added to the output list. In certain other places along the AST, such as when the condition of a rule is matched, log entries are additionally added to the output list. At the end of the diagnosis procedure, the generated list will therefore contain a combination of diagnoses and log entries. Both are extracted in order to provide the user with one list containing all applicable diagnoses, and another list containing the log of the diagnosis procedure. This procedure is shown graphically in Figure 5.3. Note that this figure shows the generation of the diagnosis list and the log list separately for clarity, while in the actual design only a single list containing all entries is generated which is filtered at the end of the procedure.

In order to make it as easy as possible on the engineer using PCDT in another piece of software to provide a diagnosis to a user, an additional function is provided to compose the diagnoses, the log entries and some additional metadata into a human readable textual output that can directly be shown to the user.

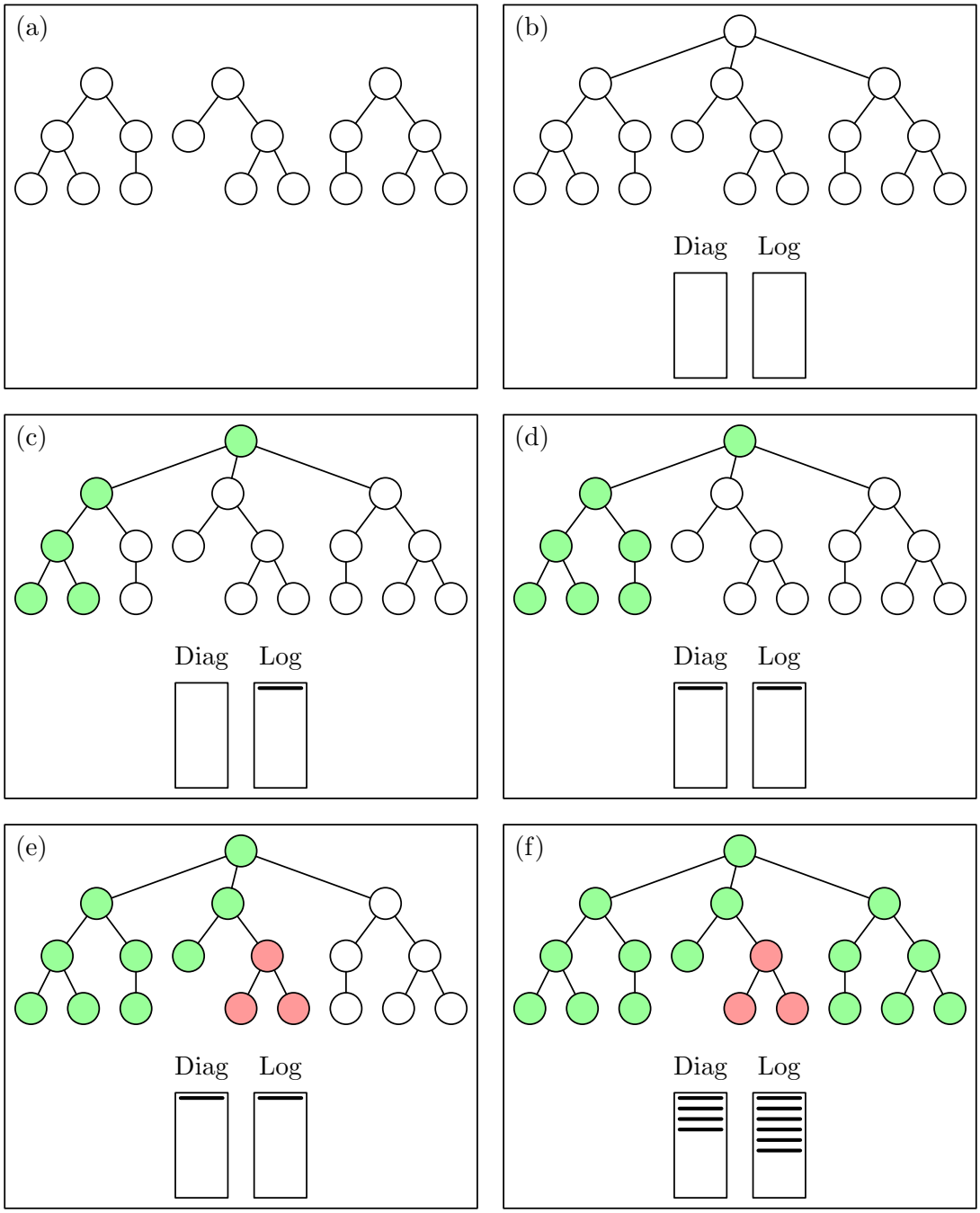

Figure 5.3: Schematic overview of the diagnosis process. (a) displays the state of the inference engine just before starting the diagnosis procedure, the three loaded ASTs are shown separately. (b) shows the combination of the stored ASTs into a single larger AST ready for traversal. When the condition of a rule is matched in (c) a log entry is added to the output list. When the consequence of a matched rule is a concrete diagnosis as in (d), the textual representation is added to the output list. In case the condition of a rule does not match as in (e), the consequence is ignored and the output list remains unchanged. As shown in (f) the procedure continues until the complete AST has been traversed at which point the output list is complete.

5.1.1 Traceability

The list of log entries is provided as additional output next to the list of diagnoses in order to ease finding bugs in the expert rules. The log can for example indicate why a certain diagnosis has been made during the diagnosis procedure, or it can help find why another has not. When PCDT is used within a production test, the textual representation of the diagnosis can also be written to the production test log directly. Doing so will give the service engineer access to the diagnosis as made at the time of the fault occurrence, since the production test log is visible to service engineers. Furthermore the log will provide additional traceability when it is stored as part of the production test log alongside the diagnoses.

5.1.2 Caching measured values

Extracting measurements from devices can be quite costly in terms of latency due to the use of low speed links between some devices and the machine running PCDT. Each measurement is therefore extracted at most once per diagnosis procedure. The first time the measurement is required in the diagnosis procedure, it is extracted from the device and cached within PCDT. The next time the measurement is required, the cached value is used instead.

5.2 Expert rule syntax and semantics

A DSL has been designed in order to bridge the semantic gap between the expertise of the domain expert and PCDT. The expert rule DSL is a declarative language in which the ordering of the rules is not important. The advantage of a declarative language over an imperative language is the decoupling between computation and control flow. In the case of the expert rule language this allows for easy addition of additional rules with little risk for problematic interactions with existing expert rules. Due to the interleaving of computation and control flow, that risk would be much greater with an imperative language. Easing the addition of new information provides for a better experience for the domain expert. The specification of the expert rule syntax is contained within the DSL definition artifact of Figure 5.2. The complete formal definition of the DSL grammar is given in Appendix A, note that this differs from the actual implementation in Prolog as contained within the definition artifact.

Each expert rule is of the form `if condition then consequence.` Where the condition consists of a boolean expression of conjunctions, disjunctions and logical negations. A logical negation, introduced with the `not` prefix operator binds strongest to its argument. Next the conjunction indicated by the `and` keyword binds strongest. Least strongly binding is the disjunction, introduced by the infix operator `or`. The meaning of a rule is intuitively that the consequence is *enacted* whenever the boolean expression of the condition evaluates to *true*.

The variables that are combined into the rule condition by the boolean expression are *findings*. Findings relate measured values from within devices to expected values given by the domain experts. Measurements are referred to as a *named value*, a value which is the result of the measurement identified by a certain name. Named values exist alongside the constant expected values within the expert rules. Depending on the type of the value, different types of comparison are possible. Boolean values can be compared on equality to other boolean values. Values representing physical quantities with like units can be compared on magnitude by the six relational comparison operators; equality (`=`), inequality (`<>`), less than (`<`), greater than (`>`), less than or equal to (`<=`) and greater than or equal to (`>=`). Comparisons on physical quantities can be chained, in which case their conjunction is taken as the result of the comparison. This allows for

easily specifying a range of values. Finally string values can be compared to other string values or matched to a regular expression value.

The expert rules also provide room for arithmetic on values representing physical quantities. The four basic arithmetic infix operators addition (+), subtraction (−), multiplication (*) and division (/) are supported. Multiplication and division have a higher precedence than addition and subtraction and all operators are left-associative, leading to the generally accepted operator order.

A consequence of an expert rule can either be a diagnosis, a deduction or a consideration. A diagnosis is introduced by the keyword `diagnose` followed by a string value containing the textual representation of the diagnosis. Whenever a diagnosis is enacted, this string value is added to the output list as a diagnosis.

A deduction is introduced by the keyword `deduce` followed by an identifier to name the deduction. This identifier can be used in place of a finding in later expert rules. Whenever a deduction is enacted it evaluates to *true* when used in the condition of an expert rule. On the other hand if the deduction is not enacted it will evaluate to *false*. The main difference between a diagnosis and a deduction is that a diagnosis represents a final diagnosis made by PCDT, it corresponds directly to an element in the output. A deduction on the other hand is generally not visible to the user, it is a construct solely for use within the set of expert rules.

Finally a consideration is introduced by the keyword `consider` and is terminated by the keyword `unconsider`. Between these two keywords is the body of a consideration, consisting of a set of expert rules. The expert rules in the body of a consideration will be taken into consideration if and only if the consideration as a whole is enacted. Variable bindings created by the condition of a consideration are also available for use by the rules contained within that consideration. Note that deductions and considerations are similar in that they can affect other rules and their expressiveness is quite similar as well. The main difference between the two is that the use of considerations leads to a hierarchical rule set while the use of deductions leads to a flatter rule set. Another difference is that a deduction must be referenced in the condition of a rule for the rule to be affected by it, on the other hand a rule is affected by a consideration by simply being located within the consideration block.

5.3 Lexical analysis and parsing

Seipel, Nogatz and Abreu [6] have shown two ways to implement a DSL within Prolog. The first way to implement a DSL within Prolog is the internal DSL. In order to implement an internal DSL in Prolog, new operators are defined which make the DSL valid Prolog. This does provide the benefit of ease of implementation and is guaranteed to be quick to parse, since it uses the Prolog built-in parser to do the heavy lifting. However as pointed out by Seipel et al., this approach also has certain drawbacks. The first of which is that certain elements of the syntax are still bound to what Prolog allows inherently. For example identifiers must not start with an uppercase letter, as that would cause them to be interpreted as variables. Furthermore identifiers can not contain periods and rules are required to end in a period, although this last convention is kept in the DSL for PCDT since it matches closely with the end of sentences in natural language. Breaking away from these conventions is not possible my merely defining some new operators.

The second drawback of an internal DSL is that error handling is performed by the Prolog parser, which expects its users to be comfortable writing native Prolog code. Due to the concept of an internal DSL all valid Prolog code is inherently valid DSL code. This means that errors in the expert rules, like starting an identifier name with an uppercase letter, are not signalled as

errors since the resulting code is still valid Prolog code, albeit not code that does what the user expects it to do.

In light of these drawbacks and due to the generally higher flexibility, the PCDT DSL is implemented as an external DSL. An external DSL requires a separate parser from the Prolog built-in parser, this is depicted as the parser component in Figure 5.2. Parsers usually operate on a list of tokens and require a separate tokenization pass, also often referred to as *lexical analysis*, beforehand. The lexical analysis is responsible for translating the raw textual input, considered as a list of characters, into a list of tokens suitable for processing by the parser proper. This same division of labour is employed by the PCDT parser as well, as it makes handling things like comments and consecutive occurrences of whitespace easier.

After the lexical analysis, the flat list of tokens can be parsed into an AST which actually represents the structure of the expert rules as a hierarchy. Both lexical analysis and parsing have been implemented by means of DCGs.

While the inclusion of the complete Prolog implementation of the parser has been omitted from this thesis due to its large size, it can be illuminating to see a small part of it in action. The example shown in Listing 5.1 handles the parsing of a numerical comparison and corresponds to the part of the formal specification of the DSL as listed in Listing 5.2.

```
1  lnode(comparison(numeric, Op, Value1, Value2)) -->
2    (
3      node(literal(number, _), Value1);
4      node(number(_, _), Value1);
5      node(number(_, _, _), Value1);
6      node(named_value(_, _), Value1)
7    ),
8    optional(node(space)),
9    node(operator(Op)),
10   {
11     member(Op, [
12       greater_than,
13       greater_than_or_equal_to,
14       equal_to,
15       less_than_or_equal_to,
16       less_than,
17       unequal_to
18     ])
19   },
20   optional(node(space)),
21   (
22     node(literal(number, _), Value2);
23     node(number(_, _), Value2);
24     node(number(_, _, _), Value2);
25     node(comparison(numeric, _, _, _), Value2);
26     node(named_value(_, _), Value2)
27   ).
```

Listing 5.1: The implementation in Prolog of the part of the parser dealing with numerical comparisons

```
1  comparison_numeric ::= number S? (">" | ">=" | "=" | "<=" | "<" | "<>") S?
2                         (number | comparison_numeric)
```

Listing 5.2: The part of the formal specification of the DSL dealing with numerical comparisons

5.3.1 Error recovery and reporting

An important part of a pleasant experience for the domain expert while writing and trying out new rules is error recovery and reporting. Syntax errors should be caught early and be reported in a clear and concise manner in order to reduce the effort required to solve the error to a minimum. Prolog and by extension DCGs are based on the concept of proving queries given a set of rules and facts. Normally whenever a query is not provable, no result is generated. Parsing is implemented as a query on the given expert rules. Without additional effort this would mean that a syntax error in the expert rules leads to no result, neither an AST, nor an error would be generated. In order to distil the exact part of the expert rules that cause the syntax error, additional clauses are implemented which are applied whenever no other result can be proven. These additional clauses consume and discard the minimal number of characters — in the case of lexical analysis — or tokens — in the case of parsing — needed to cause the remainder to be parsed successfully. In case these additional clauses were needed to establish a successful parse, the characters or tokens they have consumed including their location in the input are reported in an error message to indicate the source of the problem. This error reporting and recovery system relies on the backtracking done by Prolog to try and prove the parse query by means of the error handling clauses in case parsing without error is not possible.

5.4 Expert rule examples

In order to get an impression for the look and feel of the expert rules, the examples from Section 1.7 have been expressed as concrete examples in the designed DSL in Listings 5.3 to 5.6. Do note that the examples below ignore the types of amplifier present in the cabinet, for example in Listing 5.4 the maximum temperature is set equal for all amplifiers, while in a real-world scenario the maximum safe temperature might differ between PAACs and PADCs.

```
1  if
2    353V <= Supply.V_OUT_360V <= 370V and
3    not 350V <= Amplifier.VPWR <= 370V
4  then diagnose
5    "the problem is in the cable from the output of the power supply to the input of the power
         amplifier in slot " Amplifier.APPLICATION_ID.SLOT_NR.
```

Listing 5.3: An expert rule that is capable of diagnosing the situation where a fault in the cabling between the power supply and a power amplifier leads to a non-operational power amplifier

```
1  if
2    Amplifier1.TEMP_ENDSTAGE > 70°C and
3    Amplifier2.TEMP_ENDSTAGE > 70°C and
4    Amplifier2.APPLICATION_ID.SLOT_NR = Amplifier1.APPLICATION_ID.SLOT_NR + 1
5  then diagnose
6    "the clamp between the power amplifiers in slots " Amplifier1.APPLICATION_ID.SLOT_NR " and "
         Amplifier2.APPLICATION_ID.SLOT_NR " is not tightened properly".
```

Listing 5.4: An expert rule providing detection of a loose cold plate clamp

```
1  if
2    60°C < Amplifier.TEMP_ENDSTAGE <= 70°C
3  then diagnose
4    "the power amplifier in slot " Amplifier.APPLICATION_ID.SLOT_NR " is about to fail soon".
```

Listing 5.5: An expert rule that allows for detection of an imminent failure of one of the power amplifiers

```
1  if
2    % The LED_CONFIG_ID indicates the cabinet type
3    Controller.APPLICATION_ID.LED_CONFIG_ID = 218 and
4    % The safety relay is connected to CLI4
5    Controller.MPAC_FATAL_ERROR_HANDLING.FATAL_ERROR_VECTOR.CLI04_ERROR is true and
6    not 5 <= Amplifier.APPLICATION_ID.SLOT_NR <= 8 and
7    Amplifier.APPLICATION_ID.SLOT_NR <> 11
8  then deduce
9    safety_disabled at Amplifier.
```

Listing 5.6: An expert rule deducing the fact that a power amplifier is non-operational due to being disabled by the safety relay instead of due to a device failure, this dedcution can then be used in subsequent rules

A more complete example of a set of expert rules is shown in Listing 5.7 to give a better idea of the expressive power possessed by the expert rule DSL. This example combines the knowledge of the examples in Listing 5.3 and Listing 5.6 and combines it with actually verifying the types of the devices upon which the rules act. Lines 1 through 14 show the deductions that capture the fact that a device is of a certain type. The deductions starting with `is_a_` can be used by subsequent rules to easily restrict on which types of devices they are applicable without duplicating the 12 digit Numeric Code (12NC) — a type of identification number — for each rule. The advantage of this approach is that these deductions can be adapted easily to make all dependent rules apply to a different set of device types. This makes it easy to introduce a new variation of a device, such as a new type of power amplifier controller, since the 12NC only needs to be added or updated in a single location.

Lines 16 through 26 then show the example deduction from Listing 5.6, capturing whether a certain power amplifier is supposed to be operational or not. The example is however augmented to only be applicable on the combination of a power amplifier and a power amplifier controller by means of the deductions `is_a_power_amplifier_controller` and `is_a_power_amplifier` introduced earlier.

Lines 28 through 40 then contain a consideration in which rules can be grouped that are applicable to power amplifiers that are supposed to be operational. In this example only the example from Listing 5.3 is incorporated on lines 32 through 37, but as the comment on line 39 suggests, other rules can be included here as well. Note that the contained rule uses the same binding of the `Amplifier` variable on lines 35 and 37 as established in the condition of the consideration on lines 29 and 30.

```
1  if
2    Device.PPCA_12NC matches `4022\.634\.597.`
3  then deduce
4    is_a_power_amplifier_controller at Device.
5
6  if
7    Device.TXFRAME0.PPCA_12NC matches `4022\.646\.9646.`
8  then deduce
9    is_a_power_amplifier at Device.
10
11 if
12   Device.12NC matches `4022\.646\.4093.`
13 then deduce
14   is_a_power_supply at Device.
15
16 if
17   is_a_power_amplifier_controller at Controller and
18   is_a_power_amplifier at Amplifier and
19   % The LED_CONFIG_ID indicates the cabinet type
20   Controller.APPLICATION_ID.LED_CONFIG_ID = 218 and
21   % The safety relay is connected to CLI4
22   Controller.MPAC_FATAL_ERROR_HANDLING.FATAL_ERROR_VECTOR.CLI04_ERROR is true and
23   not 5 <= Amplifier.APPLICATION_ID.SLOT_NR <= 8 and
24   Amplifier.APPLICATION_ID.SLOT_NR <> 11
25 then deduce
26   safety_disabled at Amplifier.
27
28 if
29   is_a_power_amplifier at Amplifier and
30   not safety_disabled at Amplifier
31 then consider
32   if
33     is_a_power_supply at Supply and
34     353V <= Supply.V_OUT_360V <= 370V and
35     not 350V <= Amplifier.VPWR <= 370V
36   then diagnose
37     "the problem is in the cable from the output of the power supply to the input of the power
           amplifier in slot " Amplifier.APPLICATION_ID.SLOT_NR.
38
39   % Other rules relating to a single amplifier can go here
40 unconsider.
```

Listing 5.7: A more complete example of a set of expert rules, combining some of the previously individually shown concepts

5.5 Representation of measured values

The core concept of PCDT is based on representing measurements and performing comparisons between measurements and constant values. These measurements and constant values are represented by data types for which support is provided by the value representation component of Figure 5.2. Two main data types are identified to be useful in representing measurements; the boolean value and a value representing a physical quantity. The boolean type can represent either the value true or the value false. It is mainly useful to represent yes or no type properties, such as the enabled state of a certain functionality of a device.

5.5.1 Physical quantities

However most expressiveness is provided by the physical quantity value type. This data type represents a measurement in terms of a magnitude and a separate unit of measurement. Together this forms a very powerful representation capable of representing almost all instantaneous scalar physical measurements.

The magnitude of the physical data type is stored as an arbitrary precision rational number. This means that currently measurements with an irrational magnitude or a magnitude with a non-zero imaginary part can not be represented. This is however not an inherent limitation to the physical quantity data type, and support for either or both could be added in the future. The current implementation has been constrained to rational magnitudes in order to limit development time. The magnitude of the physical data type supports arbitrary precision to ensure that the set of representable magnitudes remains closed under addition, subtraction, multiplication and division, with the exception of division by zero. This fact ensures that those operations can be performed safely without risk for loss of information, thereby eliminating any need for checks and preventing subtle bugs. It also means that this data type is capable of representing any range of values, whether it be distances on a nanometre scale or energies in the megajoules, with the same level of precision and without adaptation.

Each rational number is stored as a pair of big integers, one numerator and one denominator. A big integer is an integer with no inherent limit to the magnitude of the value that can be stored. This makes them equivalent to the mathematical meaning of an integer and is in contrast to what most lower level programming languages, such as C++, support natively. Natively C++ supports integers around the size of a machine word, which limits the range of values that can be represented. The use of big integers to store the value of rational numbers is required to provide support for arbitrary precision values.

All four supported mathematical operations on rational numbers can be expressed in terms of a combination of the four basic mathematical operations on the underlying integers. In order to facilitate the four basic mathematical operations on big integers, implementations based on the classical algorithms for multiple-precision arithmetic as described by D.E. Knuth [19, Section 4.3.1] are employed.

The other required part of a physical quantity is the unit of measurement, since a magnitude without a unit as reference does not fully capture the information in the measurement. Lack of support for units of measurement would lead to either the use of implied units or the need to manually specify units for measured values. Both of which increase the risk of mistakes in the expert rules. The International System of Units (SI) is the most prevalent unit system in use, especially in scientific contexts. The units supported by the physical quantity type are therefore based on the SI system [20]. Internally the unit is stored in terms of the exponents of the seven SI base units. Keeping track of the unit in this way allows for easy addition and subtraction of the exponents, which is required for multiplication and division of two units respectively. Multiplication and division of physical quantities allows for arbitrarily complex derived units. Additionally the 22 derived SI units with special names and symbols are supported in the expert rules to provide a shorthand notation. These units are internally converted to the seven SI base units. Currently other shorthand unit notations such as minutes, hours, days, litres or percentages, are not supported. However adding support for other units in the future if so desired is trivial as long as the unit has an exact definition in terms of the seven SI base units.

5.5.2 Strings of characters

Certain values are inherently numerical, but are usually represented in a different format. An example would be a serial number, which is inherently a ten digit decimal integer, but is usually

represented as two groups of two digits followed by two groups of three digits, all separated by dashes. In order to support the formatting of these kinds of values a third data type is supported; the string type. The string data type is also used to allow a textual representation of diagnoses to be encoded in the expert rules.

5.5.3 Regular expressions

A final data type representing a regular expression is supported. Contrary to the other data types, the regular expression type is not used to store measured values, and can only be created by introducing a constant in the expert rules. This data type is useful to allow very expressive comparison operations with regards to strings. For example it allows a range of serial numbers to be matched or it can match all devices produced during a certain period, since the week of manufacture is encoded in the serial number. Support present in the C++ standard library is employed to provide the regular expression functionality.

5.6 Accessing device measurements

In order to improve the logical separation between the core of CabinetHAL and PCDT, the latter is not implemented in the existing classes representing devices. Rather it is implemented as a new set of classes wrapping the originals, shown as the measurement extraction component in Figure 5.2. This allows building and using CabinetHAL without the overhead of PCDT. It however also means that access to the inner working of the original classes, where the device registers containing the values of interest would be accessible, is not naturally present. This problem has been solved by employing the visitor design pattern [21] to recursively scan the cabinet model and create a wrapping object for each device present.

While the wrapping objects do have access to the device registers, the contents are still low level bits in registers. In order to convert the content of the registers into booleans, physical measurements and strings, constant information has to be provided regarding the meaning of the bits. This information is usually incorporated in the documentation of the device under consideration and needs to be converted and be statically included in the PCDT source code. A system could be envisioned in which this information is stored in a dynamic fashion such that it can be changed easily while still being accessible to PCDT. This would however require the implementation of an additional system to interpret the information at runtime. Since it is unlikely that the meaning of the contents of a register change once a device has entered the production phase, the cost of the development of this additional system is deemed too high for the value it provides.

Multiple helper functions are implemented to assign the meaning of the register contents. These helper functions allow specification of the way the registers need to be interpreted, for example if they represent unsigned integers, two's complement integers or integers encoded as binary coded decimals. Furthermore they allow to interpret such integers as physical quantities with an associated scale and unit of measurement, or to interpret the integers as formatted strings such as a serial number. Provided that a value is present in the device registers, exposing it is as simple as finding the right combination of helper functions to describe the contents of the registers appropriately. This also eases the process of adding support for new devices to PCDT. An example of how these wrapper functions can be used is given in Listing 5.8. The lines of code in this example state that the measurement is stored in the `VPWR` register, which is to be interpreted as a 16-bit two's complement integer by means of the `CTwosComplement<16>` and `CInteger` helper functions. Furthermore the meaning of this two's complement integer is

specified to be a scaled voltage. The first two parameters of the `CNumber` helper function specify the numerator and denominator of the scale factor respectively, the remaining parameters specify the unit of measurement to be in volts: $s^{-3} \cdot m^2 \cdot kg^1 \cdot A^{-1} = V$. Finally the string `"VPWR"` specifies the name of the measurement.

```
1  {"VPWR", {
2    CNumber<1ULL << 12, 100, -3, 2, 1, -1, 0, 0, 0>(
3      CTwosComplement<16>(
4        CInteger(rxregf, rxregf.VPWR)))}}
```

Listing 5.8: An example of the wrapping of the measurement of the incoming power supply voltage on the PADC

Another example is given in Listing 5.9, where the lower part of a 12NC — a type of identification number — is stored in the `PPCA_8NC_L` register in binary coded decimal form and the middle part is stored in the `PPCA_8NC_H` register. The high part of the 12NC is constant and not stored on the device, therefore it is given as a constant. The `CCombine<10000>` helper function specifies that each part is four decimal digits wide and should be concatenated to form the full value. Finally the `C12NumericalCode` helper function specifies that the 12NC should be represented as a string value in the canonical form `xxxx-xxx-xxxxx`.

```
1  {"PPCA_12NC", {
2    C12NumericalCode(
3      CCombine<10000>(
4        CBcd(rxregf, rxregf.PPCA_8NC_L),
5        CBcd(rxregf, rxregf.PPCA_8NC_H),
6        CConstantInteger<4022>()))}}
```

Listing 5.9: An example of the wrapping of a 12NC value on the PADC

Measured values within devices are often divided into different sections, grouping related measurements, sometimes also recursively. In order to follow this structure more closely and allow easy correlation between names of measurements in PCDT and the device documentation, the measurements in PCDT are represented as a trie of values. The keys of the trie form the name of the measured value and have the form of a list of strings. Each item in this list represents a level in the trie and usually corresponds to a group of related measurements.

6 | Qualification

Since the implementation of PCDT is integrated into CabinetHAL, the qualification of PCDT also has been incorporated into the qualification of CabinetHAL. Qualification of CabinetHAL is handled by means of Continuous Integration (CI). The CI workflow performs multiple parts of the qualification process automatically after a change has been made in the source code. First of all the source code is built on all platforms supported by CabinetHAL. This ensures no syntactic programming errors have been made. Compilation of the C++ code is performed with the C++ compiler in use for the specific platforms in question. The Prolog code is compiled into a saved state by the Prolog environment built as part of PCDT. This saved state is loaded by PCDT at runtime.

Secondly a static code analysis is run on the code base with the aim of finding as much semantic errors as possible. The static analysis tool is able to detect issues such as memory leaks and non-initialized variables that could lead to the software not behaving as expected. Note that the static analysis is only run on the C++ code as the types of issues it detects are hard to create in Prolog code due to the high level nature of Prolog.

Finally a regression test suite is run on the built source code. While the static analysis is able to detect general C++ programming issues, the regression test suite is designed to test that CabinetHAL behaves according to its specifications. In essence that makes it more specific to CabinetHAL. The regression test suite is divided into two major parts, the first part encompasses offline tests while the second part contains online tests. The offline tests are tests that do not require any cabinet hardware to run. For PCDT the offline tests are composed of static expert rules that do not require the extraction of measurements from devices. A diagnosis is requested based on these static expert rules and it is checked that diagnoses that should be present are present and that diagnoses that should be absent are absent. In this way the functionality of almost the complete expert system and inference engine is tested.

The online tests on the other hand actually verify the interaction between CabinetHAL and the hardware of the cabinet. While the offline tests are run automatically after each change to the source code, the online tests are run on a recurring schedule outside of working hours to allow the cabinet hardware to be available for development during working hours. For PCDT this encompasses expert rules more like those that will actually be used by the domain experts. These rules actually do read out measurements from devices and compare those measurements to expected values. However they do not actually encode for a root cause analysis, instead they are synthetic rules that assume that all devices in the qualification cabinet are fully functional. Also for the online tests a diagnosis is requested and presence of the correct diagnoses is verified. In this way the rules use the fact that the hardware is functional to verify that PCDT works correctly internally. The online tests complete the test coverage of CabinetHAL in general and PCDT in particular. It should be noted that both the offline tests and the online tests are embedded in a test application developed alongside CabinetHAL. This test application is not a real production test, but it uses CabinetHAL in the same way that a production test would.

Verification of the expert rules encoding for actual root cause analyses happens at the time they are written. Once a domain expert has reproduced and diagnosed a new issue, a corresponding expert rule can immediately be written to encode that root cause analysis. After the rule is written, it can also immediately be verified that the rule can identify the reproduced issue.

7 | Discussion

The automated diagnosis system as currently implemented within PCDT is capable of automatically recognizing failures for which expert rule have been written. Since the expert rules have access to all the measurements that a supported device makes internally, all failures that can be detected from within the cabinet hardware and do not render the required measurements inaccessible can be detected. These failure diagnoses can be automatically inferred based solely on the measured values retrieved from the devices under investigation and the provided expert rules. This satisfies Requirement 1 by PCDT being able to automatically infer diagnoses. It also means that PCDT does not need external tools or test equipment, therefore Requirement 2 is satisfied as PCDT is usable after production at Prodrive and in the field. Finally PCDT is based on expert rules as written by a human domain expert, thereby satisfying Requirement 4.

Certain failure modes may cause the measurements required to diagnose the root cause of the failure in one of the devices to become inaccessible. This can happen for example if the failure occurs within the communication link between the machine running PCDT and the failed device, or if the measurement hardware itself is defective. In these cases PCDT is not able to diagnose the root cause of the failure. However, provided that appropriate expert rules have been written, PCDT is still able to reason about the status of connected devices since the expert rules do have access to all other devices in the cabinet. Therefore PCDT is very likely to still be able to identify which device has failed, albeit without detailed root cause analysis within that device. This ability allows PCDT to automatically report which device has failed without the need for swapping devices by trial and error. This is due to PCDT only requiring access to the information absolutely necessary to make a diagnosis, a fact that also ensures that Requirement 6 is met by achieving an acceptable level of responsiveness.

PCDT is currently not yet employed at the customer sites and it will not be in in time for the finalization of this project either. Therefore a reduction in the number of devices without faults that need to be unnecessarily shipped between Prodrive and the customer for re-testing is not yet achieved. Due to PCDTs ability to automatically report which device has failed even in circumstances where no detailed root cause analysis can be established, such a reduction can be established once PCDT is made operational on customer sites.

PCDT is capable of providing a detailed root cause analysis of the failure of a single device or of a complete cabinet. However the actual reduction in the lead time of finding root causes of defect devices needs to be observed over a longer period of time after a substantial set of expert rules have been constructed by the domain experts. Furthermore a reduction in lead time of getting a machine at a customer site back up and running after a failure can only be observed once PCDT is successfully deployed on the customer sites. The expected decrease in down time of the machines can therefore not yet be objectively determined.

Alongside the diagnoses, PCDT also provides extensive logging about which expert rules are matched and from where they were loaded. This added traceability not only allows for easier serviceability of failures for which no expert rules exist yet, it also allows easy verification of new expert rules. Furthermore it establishes an improvement in the availability of diagnostic information in general. The way the actual diagnosis procedure is executed has been set up in a way such that it is possible to add additional log entries during almost all steps of the process. This property could be employed to produce even more extensive logging in the future if that is

determined to be desirable. Care must however be taken to ensure that not too much is logged, since an abundance of information might make interpreting the logging harder rather than easier.

The automated diagnosis system as currently implemented in PCDT is capable of producing multiple diagnoses from a single set of expert rules. This can be useful in instances where multiple failures have occurred at the same time. Multiple diagnoses can however also be unwanted, for example when one failure causes another failure. In such a case the deduction functionality in expert rules can be be used to ensure that the diagnosis for the resulting failure is only valid for cases where it occurs due to different causes. In this way PCDT is flexible enough to either make use of the capability to produce multiple diagnoses or not, based on the requirements of the domain expert. This flexibility in combination with the fact that the expert rules closely match the reasoning a domain expert could use in reality provides for an intuitive interface, as is necessary to fulfil Requirement 5.

7.1 Use of the diagnosis system in the field

In essence it is possible to use PCDT non-intrusively, since it only reads out measurements and does not affect the state of the machine. This means that it would be possible to gather a diagnosis from PCDT while the machine is operational. However the firmware as currently employed on the power amplifier controller in operational machines only allows reading of a very select set of measurements. Loading a different firmware in order to derive a diagnosis using PCDT would make it an intrusive operation which cannot be performed with the machine in an operational state.

A further concern is the impact of the use of PCDT on the available bandwidth of the various communication links within the machine. Due to the inherent deterministic design of the protocols used on top of these links the use of PCDT should not have an impact on the transmission of time critical data over them. However this aspect will need to be qualified to verify that PCDT does not have a negative impact on machine performance before it can be deployed on operational machines.

7.2 Limitations on the representable values

The current implementation of the representation of values within PCDT is very expressive and capable of representing a wide range of measurements. It does pose some limitations however, the most important of which is the inability to represent irrational magnitudes for physical quantities. While an arbitrary precision magnitude can be represented, an exact representation for non-rational real numbers, such as the value of π, is not supported. This means that values that need to represent such a magnitude, an angle in radians for example, currently either have to be approximated or be left unsupported. Approximating these values leads to numerical errors that can be exaggerated by mathematical operations like multiplication and division. Furthermore it causes the exact numerical comparisons, equality and inequality, to generally return a wrong result. This means that the domain expert is burdened with verifying that the order of operations leads to a minimal numerical error and that comparisons to approximated values are made with a margin of error. Similarly, complex magnitudes with an imaginary part are unsupported, although that is considered of less importance.

Certain measurements within devices represent a selection of one state out of a couple of possibilities, usually represented by means of an integer with the associated meaning elaborated in the documentation of the device. Currently these types of measurements will be represented as an integer in the expert rules as well, due to the lack of a way to encode the possible meanings

in the device wrapping code. The separately provided boolean value type can be thought of as a special case of this enumeration-like value type.

Note that neither limitation is inherent to the structure of the underlying implementation of the value representation. Both limitations could be lifted by providing their corresponding implementation within the current framework.

7.3 Maintainability and extensibility

An important part of the successful continued deployment of any system is ensuring that it is maintainable and, where applicable, extensible. Most variability within PCDT will be in the set of supported devices and the accessibility of the measurements within them. Adding support for a new device involves adding support within CabinetHAL if not yet present and wrapping the measurements within the device. Wrapping a new measurement, whether in a new device or in an already supported one, is accomplished by a single line of code, keeping the burden of maintenance low. With the way PCDT has been set up, it has actually become applicable to all register based devices, it is actually in no way specific to the power cabinet infrastructure anymore. This means that Requirement 3 is satisfied more than adequately.

After support for a new measurement has been included in PCDT, the domain expert can start writing expert rules using it. This is immediately the limiting factor in maintenance and extensibility for PCDT. Since it is not a self learning system, the required knowledge does need to be provided by the domain expert. However it does mean that PCDT is immediately deployable, satisfying Requirement 7.

8 | Conclusion

The project presented here has led to a successful implementation of an automated diagnosis system capable of providing additional diagnostic information gathered from the power cabinet infrastructure. The automated diagnosis system is furthermore capable of automatically deducing diagnoses based on that additional information for logging purposes, where it improves the traceability of structural issues. Additionally the automatic root cause analysis provides immediate useful feedback to ease solving of the failure at hand. Lastly the automated diagnosis system has evolved in such a way that it is no longer necessarily specific to the power cabinet infrastructure, to the point that PCDT could be made to support many more complex register based devices with relatively little effort.

8.1 Future work

The implementation of the automated diagnosis system as presented is capable of representing instantaneous scalar measurements of physical quantities. It is however conceivable that the capability of tracking the trends of measurements over time provides a whole new dimension to the expressiveness and diagnostic power of PCDT. A general point of concern in this that deserves special attention is the latency and throughput limitations of the communication links employed between the machine running PCDT and the devices under investigation. Additionally care must be taken to ensure this time as a property functionality does not conflict with the caching of measurements as currently implemented in PCDT. Nevertheless investigating the possibilities regarding time sensitive expert rules is suggested as a promising future research direction.

A second interesting avenue to pursue is the use of augmented reality to provide an immediate visual reference corresponding to the diagnosis presented by PCDT. This would allow the engineer in charge of resolving a failure to visually identify the spatial location of diagnosed root cause by means of a visual overlay, instead of or alongside of having to interpret the textual diagnosis as presented by PCDT currently.

Finally the completeness of the knowledge about existing failure modes and corresponding root causes could possibly be improved by designing a self learning inference engine of some type, such as in the form of an ANN. This new inference engine can then take the place of the current expert rule based inference engine in an attempt to reduce the data acquisition bottleneck. A more complete knowledge base paves the way for more detailed diagnoses that more accurately describe the root-cause of a failure and for better diagnosis coverage so more failure modes can be identified. Possibly this direction of development can be combined with additional research into the possibilities of creating a digital twin for the power cabinet infrastructure, for example to share the core of the digital twin to act as the source of domain knowledge for the inference engine as well.

A digital twin is a replica of a system built by employing measurements from actual physical instances of the system. If a digital twin of the power cabinet infrastructure is created in the future, this opens up possibilities in PCDT. An interfacing layer can be created which adapts the PCDT interface to the digital twin model. This will allow PCDT to use the digital twin as if it was the real system. This can decrease the effort required in finding failure modes that

require new expert rules and it can help during the creation and verification of the expert rules. Additionally it might make it feasible to apply other techniques in the diagnostic system not currently chosen.

Depending on how this future digital twin will be designed, it might even be possible to plug the core of the model, an ANN for example, into PCDT instead of the inference engine as designed in this project. Further training of the digital twin model will then automatically lead to more accurate diagnoses from PCDT with higher coverage.

A | Grammar of the expert rule DSL

The grammar of the expert rule DSL is given in Listing A.1, it is specified using a simple Extended Backus-Naur Form (EBNF) notation. The notation used closely follows the EBNF used by the World Wide Web Consortium (W3C) to specify the grammar of the Extensible Markup Language (XML) [22, Section 6].

Each rule in the grammar defines one symbol, in the form `symbol ::= expression`. Within the expression on the right-hand side of a rule, the following expressions are used to match strings of one or more characters:

`#xN`

: where `N` is a hexadecimal integer, the expression matches the character whose Unicode code point is `N`. The number of leading zeroes in the `#xN` form is insignificant.

`[a-zA-Z]`, `[#xN-#xN]`

: matches any character with a value in the inclusive range or ranges indicated.

`[abc]`, `[#xN#xN#xN]`

: matches any character with a value among the characters enumerated. Enumerations and ranges can be mixed in one set of brackets.

`[^a-z]`, `[^#xN-#xN]`

: matches any character with a value *outside* the range or ranges indicated.

`[^abc]`, `[^#xN#xN#xN]`

: matches any character with a value *not* among the characters given. Enumerations and ranges of forbidden values can be mixed in one set of brackets.

`"string"`

: matches a literal string matching that given inside the double quotes.

`'string'`

: matches a literal string matching that given inside the single quotes.

These symbols may be combined to match more complex patterns as follows, where `A` and `B` represent simple expressions:

`(expression)`

: `expression` is treated as a unit and may be combined as described in this list.

`A?`

: matches `A` or nothing; optional `A`.

`A B`

matches `A` followed by `B`. This operator has higher precedence than alternation; thus `A B | C D` is identical to `(A B) | (C D)`.

`A | B`

matches `A` or `B`.

`A - B`

matches any string that matches `A` but does not match `B`.

`A+`

matches one or more occurrences of `A`. Concatenation has higher precedence than alternation; thus `A+ | B+` is identical to `(A+) | (B+)`.

`A*`

matches zero or more occurrences of `A`. Concatenation has higher precedence than alternation; thus `A* | B*` is identical to `(A*) | (B*)`.

Other notations used in the productions are:

`/* ... */`

is a comment.

```
1  ruleset ::= (S? rule S)* rule?
2  rule ::= "if" S condition S "then" S consequence S? "."
3
4
5  S ::= (whitespace | comment_line | comment_block)+
6  whitespace ::= (#x09 | #x0A | #x0D | #x20)+
7  comment_line ::= "%" [^#x0A#x0D]*
8  comment_block ::= "{%" ([^%] | "%" [^}])* "%}"
9
10
11 condition ::= finding |
12               condition (S ("and" | "or") S condition)? |
13               "not" S condition |
14               "(" S? condition S? ")"
15
16 finding ::= comparison_numeric | comparison_exact |
17             comparison_pattern | deduction_name
18 comparison_numeric ::= number S? (">" | ">=" | "=" | "<=" | "<" | "<>") S?
19                        (number | comparison_numeric)
20 comparison_exact ::= (boolean S ("is" | "is" S "not" | "equals") S boolean) |
21                      (string S ("is" | "is" S "not" | "equals") S string)
22 comparison_pattern ::= string S "matches" S regex
23
24
25 boolean ::= boolean_literal | named_value
26 boolean_literal ::= "true" | "false"
27
28 number ::= (number_literal | named_value)
29                (S? ("+" | "-" | "*" | "/" | S) S? number)? |
30            "(" S? number S? ")"
31 number_literal ::= number_magnitude number_unit? | number_unit
```

```
number_magnitude ::= number_significand number_exponent?
number_significand ::= number_integer number_fraction? |
                       ("–" | "+")? number_fraction
number_exponent ::= ("e" | "E") number_integer
number_integer ::= ("–" | "+")? [0–9]+
number_fraction ::= "." [0–9]+
number_unit ::= number_prefix? number_postfix number_power?
number_prefix ::=
  "Y"  |
  "Z"  |
  "E"  |
  "P"  |
  "T"  |
  "G"  |
  "M"  |
  "k"  |
  "h"  |
  "da" |
  "d"  |
  "c"  |
  "m"  |
  "u"  | #x00B5 | #x03BC |
  "n"  |
  "p"  |
  "f"  |
  "a"  |
  "z"  |
  "y"
number_postfix ::=
  "s"   |
  "m"   |
  "g"   |
  "A"   |
  "K"   |
  "mol" |
  "cd"  |
  "rad" |
  "sr"  |
  "Hz"  |
  "N"   |
  "Pa"  |
  "J"   |
  "W"   |
  "C"   |
  "V"   |
  "F"   |
  "ohm" | "R" | #x03A9 | #x2126 |
  "S"   |
  "Wb"  |
  "T"   |
  "H"   |
  "'C"  | #x00B0 "C" | #x2103 |
  "lm"  |
  "lx"  |
  "Bq"  |
  "Gy"  |
  "Sv"  |
  "kat"
number_power ::= "^" number_integer

string ::= (string_literal | named_value | boolean | number) (S? string)?
string_literal ::= '"' [^"]* '"' | "'" [^']* "'"
```

```
94
95  regex ::= (regex_literal | string) (S? regex)?
96  regex_literal ::= '`' [^`]* '`'
97
98  named_value ::= identifier ("." identifier)+
99
100
101 consequence ::= diagnosis | deduction | consideration
102
103 diagnosis ::= "diagnose" S string
104 deduction ::= "deduce" S deduction_name
105 consideration ::= "consider" S ruleset "unconsider"
106
107 deduction_name ::= identifier (S "at" S identifier)?
108
109
110 identifier ::= [a-zA-Z_][a-zA-Z0-9_]+
```

Listing A.1: The grammar of the expert rule DSL

Bibliography

[1] H. Butler, "Position control in lithographic equipment [applications of control]," *IEEE Control Systems Magazine*, vol. 31, no. 5, pp. 28–47, 2011. 2

[2] A. Gómez-Pérez, M. Fernández-López, and O. Corcho, *Ontological Engineering: With Examples from the Areas of Knowledge Management, E-Commerce and the Semantic Web*. Springer, 01 2004. 10

[3] L. Travé-Massuyès, "Bridging control and artificial intelligence theories for diagnosis: A survey," *Engineering Applications of Artificial Intelligence*, vol. 27, pp. 1 – 16, 2014. 10

[4] F. Puppe, *Systematic Introduction to Expert Systems: Knowledge Representations and Problem Solving Methods*. Berlin, Heidelberg: Springer-Verlag, 1993. 11, 17

[5] D. Merritt, *Building Expert Systems in Prolog*. Berlin, Heidelberg: Springer-Verlag, 1989. 12, 17

[6] D. Seipel, F. Nogatz, and S. Abreu, "Domain-specific languages in prolog for declarative expert knowledge in rules and ontologies," *Computer Languages, Systems & Structures*, vol. 51, pp. 102 – 117, 2018. 12, 17, 33

[7] Z. Zhang, Z. Wang, X. Gu, and K. Chakrabarty, "Board-level fault diagnosis using bayesian inference," in *2010 28th VLSI Test Symposium (VTS)*, pp. 244–249, April 2010. 12

[8] S. G. W. Butcher and J. W. Sheppard, "Distributional smoothing in bayesian fault diagnosis," *IEEE Transactions on Instrumentation and Measurement*, vol. 58, pp. 342–349, Feb 2009. 13, 15

[9] F. Ye, Z. Zhang, K. Chakrabarty, and X. Gu, *Knowledge-Driven Board-Level Functional Fault Diagnosis*. Springer, 01 2017. 13, 16

[10] F. Ye, Z. Zhang, K. Chakrabarty, and X. Gu, "Adaptive board-level functional fault diagnosis using incremental decision trees," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, pp. 323–336, Feb 2016. 13

[11] J. Aldrich, "Correlations genuine and spurious in pearson and yule," *Statistical Science*, vol. 10, no. 4, pp. 364–376, 1995. 16

[12] P. Lucas and L. Van Der Gaag, *Principles of Expert Systems*. International Computer Science Series, Addison-Wesley, 1991. 19

[13] G. S. Tseitin, *On the Complexity of Derivation in Propositional Calculus*, pp. 466–483. Berlin, Heidelberg: Springer Berlin Heidelberg, 1983. 21

[14] M. Davis and H. Putnam, "A computing procedure for quantification theory," *J. ACM*, vol. 7, p. 201–215, July 1960. 22

[15] M. Davis, G. Logemann, and D. Loveland, "A machine program for theorem-proving," *Commun. ACM*, vol. 5, p. 394–397, July 1962. 22

[16] K. Gödel, "Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I," *Monatshefte für Mathematik und Physik*, vol. 38, no. 1, pp. 173–198, 1931. 23

[17] C. Chang and R. Lee, *Symbolic Logic and Mechanical Theorem Proving*. Computer science and applied mathematics, Academic Press, 1973. 23

[18] R. A. Kowalski, "The early years of logic programming," *Commun. ACM*, vol. 31, p. 38–43, Jan. 1988. 27

[19] D. E. Knuth, *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. USA: Addison-Wesley Longman Publishing Co., Inc., 1997. 38

[20] BIPM, *The International System of Units (SI)*. International Bureau of Weights and Measures, 9th ed., 2019. 38

[21] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. USA: Addison-Wesley Longman Publishing Co., Inc., 1995. 39

[22] M. Sperberg-McQueen, T. Bray, F. Yergeau, E. Maler, and J. Paoli, "Extensible markup language (XML) 1.0 (fifth edition)," W3C recommendation, W3C, Nov. 2008. http://www.w3.org/TR/2008/REC-xml-20081126/. 47