Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Measurement and Information Systems

Inductive reasoning supported fault recovery

Scientific Students' Association Report

Author:

Gergő Welker

Advisor:

András Földvári

2022

Contents

Kivonat

A világon nincs olyan ember alkotta rendszer, amely hibátlanul működne, ezért elengedhetetlen a hibadiagnosztika a rendszerleállások okozta kiesések és a karbantartási idők csökkentése érdekében. A rendszerek egyre összetettebbé válásával párhuzamosan a diagnosztikai problémák jelenlegi megoldásai nehezen skálázódnak. Az ipar által jelenleg használt módszereknél jellemzően egy magasabb fokú automatizálási és agilisabb diagnosztikai megközelítésre van szükség.

A dolgozat a klasszikus rendszer szintű diagnosztika eredményeiből kiindulva bemutat egy újszerű, induktív logikai programozás (ILP) alapú, adatvezérelt hibaelhárítást támogató megközelítést és annak egy prototípusán egy alkalmazási példát. Kifejezetten új elem, a szakterületi tudás alapú optimalizációs feladatok támogatása. A dolgozat tárgyalja még a javasolt megközelítés előnyeit, illetve szinergiáját a meglévő megoldásokkal.

A hibaelhárítási problémák induktív logikai programozás alapú támogatásának egyik fő előnye, hogy a kikövetkeztetett, hibajavítást támogató szabályok megmagyarázhatóak, ami a kritikus rendszerek szempontjából elengedhetetlen. Emellett a logika alapú módszerek nem igényelnek nagy mennyiségű adatot a következtetések levonásához, így a szabályrendszer kinyerése gyorsan lezajlik.

A szakterületi tudás és modellek felhasználásával lehetővé válik, hogy az elemzés megoldása egy pontosabb képet adjon a problémáról és kiterjeszthető legyen olyan speciális esetekre is, ahol a klasszikus megoldásoktól eltérően kell kezelni a meghibásodásokat.

Az adatvezérelt hibadiagnosztika alapja az adatgyűjtés és adatfeldolgozás. Az összetett rendszerek hatalmas mennyiségű adatot generálnak, amelyek hatékony feldolgozására van szükség. A kvalitatív modellek átláthatóbb képet adnak a komplex rendszerekről szimbolikus, könnyen érthető módon. Továbbá ezen modellek tetszőlegesen skálázhatók, és támogatják a széleskörben használt modellellenőrzési és diagnosztikai technikákat.

A dolgozat tárgyalja az ILP segítségével történő szakterületi tudás alapú optimalizációs hibaelhárítási megoldásokat. Megvizsgál egy Answer Set Programming alapú ILP keretrendszert, a FastLAS-t, és néhány példán keresztül szemlélteti a bemutatott megközelítést.

Abstract

There are no human-made systems in the world that function in a flawless manner. Thus, fault recovery is essential in reducing system downtime and maintenance time. As systems become increasingly complex, diagnostic problems also become more difficult. Compared to solutions currently used by the industry, today‘s problems typically require a higher degree of automation and a more agile diagnostic approach.

The report presents a novel, Inductive Logic Programming (ILP) based, data-driven fault recovery approach based on the results of classic system-level diagnostics. Additionally, it showcases some application examples with the help of a prototype. An especially new element is the support of domain-specific knowledge-based optimization tasks. The report also discusses the advantages of the proposed approach and its synergy with existing solutions.

One of the main advantages of ILP-supported fault recovery is that the fault recovery action rules can be explained, which is essential for critical systems. In addition, logic-based methods do not require large amounts of data to draw conclusions, so the extraction of the ruleset takes place quickly.

With the help of domain-specific knowledge and models, it becomes possible for the solution of the analysis to give a more accurate picture of the problem and to be extendable to special use cases where failures must be handled differently from classical solutions.

Data-driven fault diagnosis is based on data collection and data processing. Complex systems generate huge amounts of data that require efficient processing. Qualitative models provide an abstract picture of complex systems in a symbolic, easy-to-understand way. Moreover, these scalable models support widely used model verification and diagnostic techniques.

The report discusses ILP-supported, domain-specific knowledge-based optimization solutions for fault recovery. It examines an Answer Set Programming based ILP framework - called FastLAS - and illustrates the presented approach through examples.

Chapter 1

Introduction

Fault recovery inside complex systems is an increasingly important problem. Faults can have a wide range of causes, such as design and implementation errors, human errors or deterioration damages. As a result, faulty systems usually have decreased performance, they do not function as intended, or do not function at all. Recovering from these faults is of utmost importance to maintain the expected level of service. The primary goal of fault recovery is to restore the system to normal operation when the system fails by executing *fault recovery actions*. With the help of proper recovery actions, *system downtime* and *maintenance times* can be reduced with a great extent, and usually, this also results in lowered *maintenance costs*. However, providing reliable fault recovery actions is not a trivial process.

The aim of the report is to present and examine a novel, *Inductive Logic Programming* (ILP) supported, *data-driven*, *domain-specific knowledge-based* approach to the well-researched area of fault recovery. The goal of the approach is to support fault recovery by generalizing, optimizing and validating the recovery actions rules with the help of domain-specific knowledge.

As systems become more and more complex, providing fault recovery actions is also becoming more complicated. Determining appropriate recovery actions for the possible *fault scenarios* while accounting for *error propagation* is an exhaustive engineering process. Additionally, in many cases, there is not a single set of recovery actions that fits every environment in which a system operates. Different external circumstances and constraints can require the alteration of the recovery process to be optimized to different targets, such as *recovery cost* and *recovery time*. To support complex systems, the presented approach incorporates *engineering models* (e.g., information flow model) and *cost metrics* (e.g., recovery time) that can be used to provide solutions for a wide range of fault recovery related optimization and validation problems.

There are many possible approaches to provide fault recovery actions, both manual and automatic. These include — among many others — system modeling and system simulation to identify potential failure points and to provide recovery actions to fix them. Additionally, there are *learning-based approaches* that utilize artificial intelligence and machine learning. All different approaches have different use cases, and can be applied in different scenarios. In many systems, one approach is usually not enough, as it cannot cover the system at the required level, or it would be too complicated to do so. In *complex systems*, the approaches are combined to cover the system in greater detail. Thus, more fitting recovery actions can be provided. While the presented approach is fully functional

on its own, it can also be used in conjunction with other fault recovery approaches to provide even better fault recovery solutions.

1.1 The Presented Approach

The most fundamental part of the approach is that it is based on *logic programming*. The ILP part of the approach provides the ability to learn from examples and to extract *logical rules*. Additionally, ILP can take into consideration a *knowledge-base* containing *background information*, moreover, specific ILP systems also support the optimization of the extracted rules with respect to predefined criteria. This overall combination helps overcome several shortcomings of other fault recovery solutions. The following section describes the goal of each part of the approach:

Knowledge-base - Knowledge-base — or background knowledge — is going to be used to provide additional information about the examined system, such as system architecture and hierarchy. Incorporating this information in the rule extraction process can be very beneficial. Knowing system connections and interactions can help in finding more appropriate recovery actions.

Logic programming - Thanks to logic programming, everything in the workflow is fully explainable, which is a very important concept in fault recovery. *Answer Set Programming* (ASP) — a logic programming language — is going to be used in this report to programmatically represent fault recovery tasks. This means that examples, background knowledge and optimization targets will also have to be translated into code with logic programming.

ILP - ILP intends to provide some of the benefits of machine learning based fault recovery approaches. This mainly includes learning from examples and generalizing to a wider range of fault recovery scenarios with a focus on *explainability*. ILP systems incorporate a knowledge-base into the *rule-extraction process*, additionally, specific ILP systems support the optimization of the task based on *scoring functions*. One such system, which is going to be used in this report, is called *FastLAS* [1].

Scoring functions - Scoring functions are going to be used to optimize fault recovery rules with respect to different criteria, such as recovery cost. They are going to make use of the background knowledge to score recovery actions based on external constraints, additionally, the architecture, components and the hierarchy of the system.

1.2 Workflow Processes

The workflow presented in the report can be broken up into well-separated sub-processes. An overview is depicted in Figure 1.1. Here the separate processes will only be briefly described since in-depth explanations with use cases will be provided in later chapters.

Knowledge-Base Compilation - The first process is called Knowledge-Base Compilation. Its goal is to translate the input system models and external requirements into a form that can be used in the Rule-Extraction process. This includes translating the system model with ASP into a logic-based representation and formulating the external requirements and cost metrics into scoring functions.

Rule-Extraction - The second process is called Rule-Extraction. Its purpose is to extract the fault recovery rules while taking as inputs the examples from the system

and the compiled knowledge base from the first process. In the report, this process is going to be performed with FastLAS.

Evaluation - The final process is the Evaluation. After the rules have been extracted, they have to be examined and checked whether they are correct and provide the intended fault recovery actions. Because the rules are ASP rules, they can be easily interpreted by a system expert, and any required adjustment can be carried out by making changes in the preceding processes.

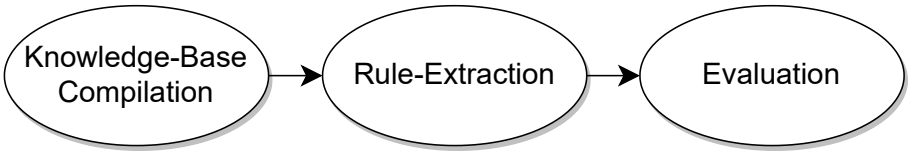

Figure 1.1: The processes of the workflow

1.3 Report Structure

In the following sections, the report details the approach and the workflow in great depth alongside use cases and examples. Firstly, it will take a broader look at fault diagnosis and recovery in Chapter 2, then move on to the presented workflow in Chapter 3. After that, it will demonstrate concepts of system modeling in Chapter 4 that are required to formulate a good rule extraction task, followed by taking a look at the more technical parts of the approach, such as ASP in Chapter 5 and ILP in Chapter 6. After these steps, all the background information will be provided to examine the workflow in action through use cases and examples in Chapter 7.

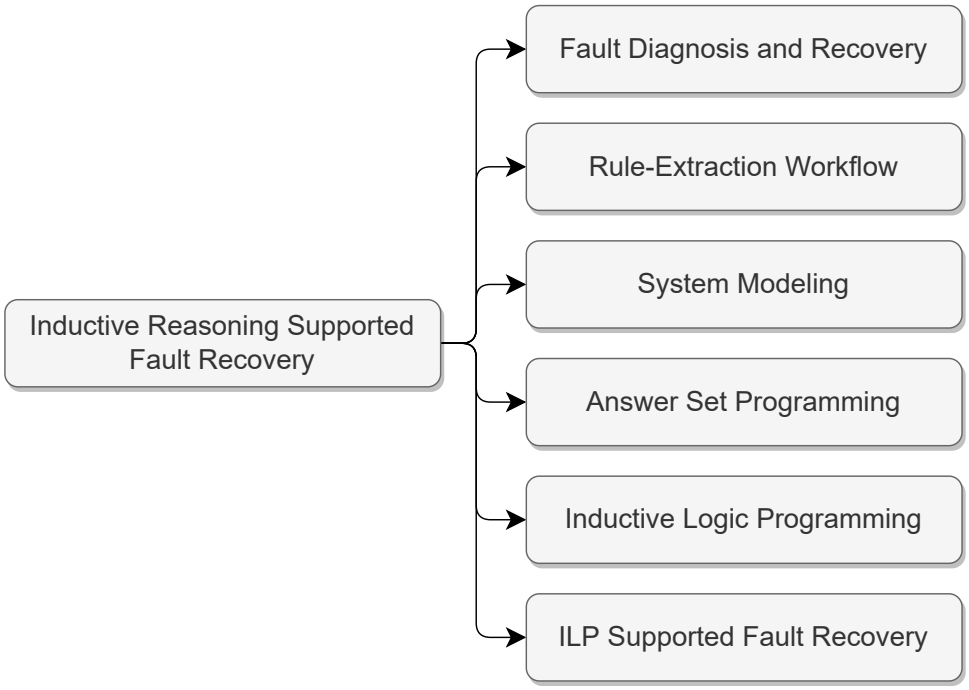

Figure 1.2: Report structure mind map

Chapter 2

Fault Diagnosis and Recovery

This chapter introduces the connection between *fault diagnosis and recovery*. It provides an overview of the possible fault diagnosis approaches, such as analytical model-based and data-driven diagnosis. After that, it details the general concepts of fault recovery.

Figure 2.1 outlines the connection between fault diagnosis and fault recovery. When a fault occurs in a system, the system can no longer properly fulfil its tasks. The consequences of the fault can be visible in countless forms. In the best case, the fault causes only subtle errors in the system operations and does not affect the whole system continuously. On the other end, there can be faults that make the entire system unusable until everything is not repaired. The severity of the faults is dependent on the importance of the components which are affected by the fault. For instance, there can be components which are only used occasionally, and if they become faulty, then they might be restored to normal operation before they have to be used, causing no disruptions to the system operations. In any case, when there is a fault, it must be diagnosed. The task of fault diagnosis is to monitor the system and to find the component causing the faulty behaviour as soon as possible. When the fault has been diagnosed, it is time to begin the recovery process based on the collected data. At the end of a successful fault recovery step, all the faulty system components have to be fixed, and the whole system has to operate as intended.

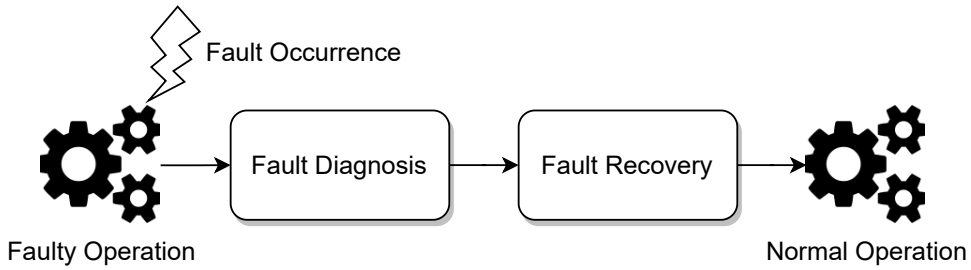

Figure 2.1: Connection of fault diagnosis and fault recovery

2.1 Fault Diagnosis

Fault diagnosis is a closely related field to fault recovery, and in many cases, it is a preceding step that provides the appropriate inputs required by fault recovery. Moreover, many concepts used in fault diagnosis can also be utilized in the fault recovery process. The field of fault diagnosis is highly researched, so it is beneficial to use these well-researched and proven concepts.

Fault diagnosis consists of three main parts, which are the following [2]:

- Fault detection: detection of the occurrence of a fault inside the system, which leads to undesired consequences and system malfunctions
- Fault isolation: localization of a fault
- Fault identification: determining the location, type and severity of a fault

Figure 2.2 depicts the different fault diagnosis methods. The methods can be classified into three groups: *analytical model-based*, *data-driven* and *knowledge-based* methods. Starting with analytical model-based methods, initial research on fault diagnosis dates back to the 1970s [3, 4], where it was used to detect and isolate faulty system components. As systems became more complex and the data produced by these systems became enormous, analytical model-based methods have been replaced with data-driven alternatives that started to make use of data analytics and machine learning. Data-driven solutions [5] proved to be very useful to a wide range of fault diagnosis problems. They are very well suited to detect and locate faults in systems only based on data without requiring an accurate mathematical model of the system. However, the lack of possibility to supply background knowledge to data-driven approaches and their black-box nature led to the popularity of knowledge-based approaches. Knowledge-based fault diagnosis makes it possible to supply known facts about the system and use them together with additional data to deduce new, hidden knowledge about the system with logical reasoning methods.

The knowledge-based fault diagnosis aspect is going to play a prominent role in the report since it is used in the presented knowledge-based fault recovery approach. The methods with green backgrounds are especially important. Those are the methods that will directly appear in the report in later chapters.

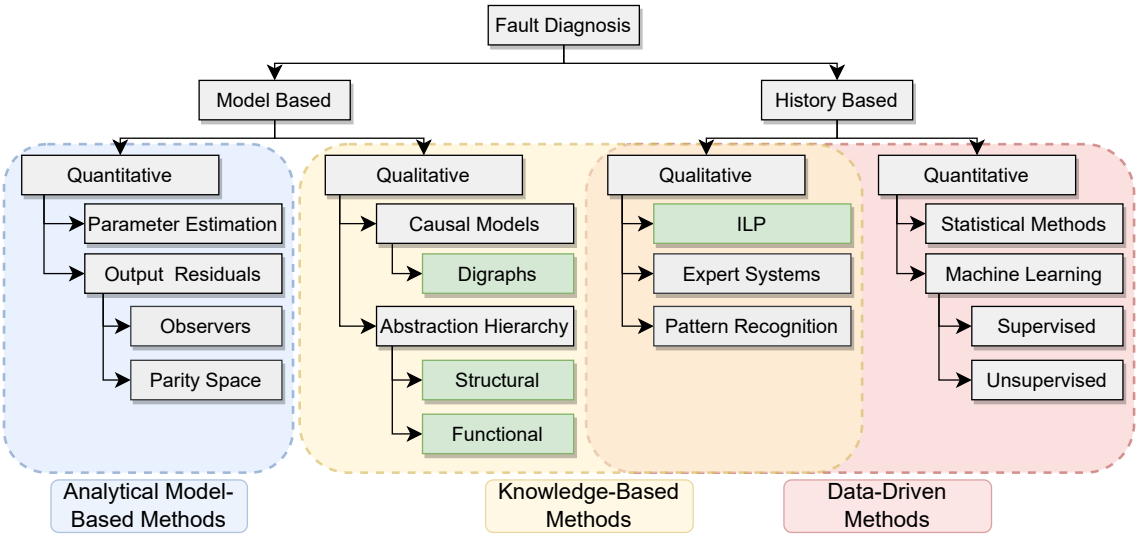

Figure 2.2: Classification of fault diagnosis methods [6]

2.1.1 Analytical Model-Based Fault Diagnosis

The analytical model-based fault diagnosis approach was initially developed to provide fault diagnosis for industrial machines [3]. The goal was to improve the reliability of industrial systems in which components provided overlapping and supplementary functions.

In case of a component failure, the component was not replaced but isolated, and the functions it performed were assigned to other components that could fulfil the required operations.

The foundations of analytical model-based fault diagnosis are mathematical models that accurately describe the processes of the examined system. The models are carefully engineered to estimate the behaviour of the real-world system as closely as possible. The differences between the model's estimated output of the system and the running system's output are used as a basis for fault diagnosis [5]. These differences are usually referred to as *residuals*, and decision rules check threshold limits of these residuals to generate fault symptoms. There are several methods to monitor and process the relationship of the outputs. In most approaches, a nonzero residual indicates the occurrence of a fault inside the system.

Analytical model-based fault diagnosis has many great use cases, and many robust systems use it for fault diagnosis. However, one of its main drawbacks is that it is very complicated to develop precise physical models of large complex systems [5]. Additionally, an extensive monitoring system is required to properly observe the system, which adds another layer of complexity to the process. In cases where analytical approaches cannot be used, other methods have to be utilized, such as data-driven fault diagnosis.

2.1.2 Data-Driven Fault Diagnosis

Data-driven fault diagnosis methods extract information from historical samples of the system. There are numerous approaches, such as clustering [7] and statistical analysis [8]. These methods became very popular due to the following reasons:

- Complex systems generate huge amounts of data, and data is the main building block of these approaches
- They do not require an accurate physical model of the underlying system
- They have the ability to process high-dimensional data interdependencies

Most data-driven methods apply algorithms on the historical samples of the system to discover patterns in the examples. Later these patterns can be used for fault detection and diagnosis by comparing the actual system patterns to the previously discovered patterns. Or in other cases, they can be used to project possible faults that might happen in the future based on telltale fault signs [9].

Data-driven methods include machine learning-based approaches to fault diagnosis. Machine learning became very popular in the field [10] because it can be easily applied to a wide range of fault diagnosis problems. For example, Neural Networks (NNs) [11] can be used as supervised multi-class fault classifiers. The inputs can be attributes from a system, and the neuron outputs can correspond to fault locations or fault types. However, two major drawbacks of machine learning-based approaches — both due to their black box-like operation — are that their inner working is hidden from the outside world, and the models they learn can not be explained.

2.1.3 Knowledge-Based Fault Diagnosis

Knowledge-based fault diagnosis consists of a knowledge base and an inference engine to deduce new insights [4]. They are especially well suited in the following cases:

- For complex systems where accurate mathematical models are not available or they would be too complicated and expensive to be developed.

- Where the fault diagnosis process needs to be explainable because it is not enough to only know the input and output mappings, and the inner workings are important too.
- When there is available background knowledge that can be used during the inference process.

Knowledge-based approaches provide several of the advantages of data-driven solutions since they are based on data too. However, how this data is utilized is different since it is stored in a knowledge base in a logically structured form. This approach very much mimics how intelligent beings utilize information. The known facts are accumulated in a knowledge base which can be used to deduce new facts from unknown data. The new facts can also be added to the knowledge base, providing a way for the process to continuously improve as new data is processed.

The applicability of knowledge-based fault diagnosis depends on the quality and completeness of the knowledge base and the overall knowledge. Therefore, it is crucial to extract and provide information in the background knowledge in a well-structured form. The more background information is available; the more effective the approach can be. Every system-related information can be included in the knowledge base that can be represented in a logical way. This includes the structural, behavioural and functional models of the system.

2.2 Fault Recovery

Fault recovery is just as important as fault diagnosis. Fault diagnosis detects, identifies and locates faults in the system, and this information is usually part of the inputs of the fault recovery process. Whenever a fault occurs, after it has been diagnosed, fault recovery should begin. The main focus of fault recovery is to find and provide the required steps and actions to return to normal operation after a fault has occurred.

In many cases, the fault recovery process can be very complex. Even in a small system, there can be many connections, dependencies and interactions between system components. The errors caused by a fault in one component can propagate to other connected components. The recovery process must consider error propagation in the system to provide actions that target the root component of the errors, where the fault actually happened. In addition, the fault recovery process must be executed rapidly to restore normal operations as soon as possible. From the point of detection, the time it takes to recover is very much dependent on how reliable and accurate the fault recovery process is. If the recovery actions properly target the faulty components and if the actions are well defined, it can shorten the time required to recover.

There are many possible approaches to implement fault recovery. The approaches can mainly be categorized as manual or automatic. A mutual property in both of them is that they require data to be collected about the possible faults and their consequences in the system. This data includes, for instance, system models, logged fault scenarios and system simulation data.

Manual - A manual approach to fault recovery might be completely adequate for smaller systems. It includes a system expert who thoroughly knows and understands the system. Based on the collected data, the expert provides recovery actions for fault scenarios ranging from minor software patches and updates to hardware replacements. This process can be imagined as a manual classification, where a fault or a combination of faults is mapped to a single or multiple recovery actions.

Automatic - Fault recovery in complex systems requires additional steps since the size of the collected data is much larger, and many times it cannot be manually handled. This is where automatic methods can be very beneficial. Automatic methods require the input data to be labeled — which usually comes from the manual step — and based on this data, they can generalize to a broader range of faults in the system. There are many good approaches to automatically extract recovery rules from labeled data, such as Artificial Neural Networks [12].

To implement knowledge-based fault recovery, several previously discussed aspects of knowledge-based fault diagnosis can be utilized. Its features and advantages are very much similar in the two domains. Knowledge-based fault recovery will be detailed alongside the rule-extraction workflow in Chapter 3.

Chapter 3

Rule-Extraction Workflow

This chapter takes a look at *knowledge-based fault recovery* and discusses its parts, aspects and related technologies. After that, the *rule-extraction workflow* will be detailed, which is the main contribution of this report. It encompasses all the required inputs, processes and steps to implement knowledge-based fault recovery.

3.1 Knowledge-Based Fault Recovery

Knowledge-based fault recovery is a *data-driven automatic fault recovery approach*. It aims to provide explainable fault recovery rules based on example data with the help of *reasoning processes*. Figure 3.1 depicts the approach, where the rectangles illustrate data, the rounded rectangle depicts the knowledge-based fault recovery process, and the ellipses depict its components. The two components are the *knowledge base* and an *inference engine*. The knowledge base stores background information about the examined system and provides it to the inference engine in an easily interpretable form. The inference engine deduces new facts from the knowledge base with the help of reasoning methods that resemble the human thinking process. The input data to the process consists of examples and any kind of problem-related information that can be used during the inference process.

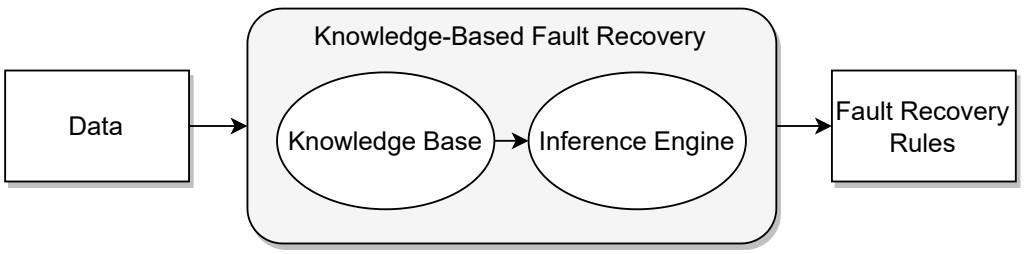

Figure 3.1: Knowledge-based fault recovery approach

The components of knowledge-based fault recovery can utilize a number of methods which are represented in Figure 3.2. These methods can be arbitrarily combined to fit the recovery process to the problem as much as possible. The knowledge base can mainly use the following three knowledge representation mechanisms: *ontologies*, *knowledge graphs* and *logic rules*. How the information is processed from the knowledge base by the reasoning process can be based on *deductive*, *inductive* or *abductive reasoning*.

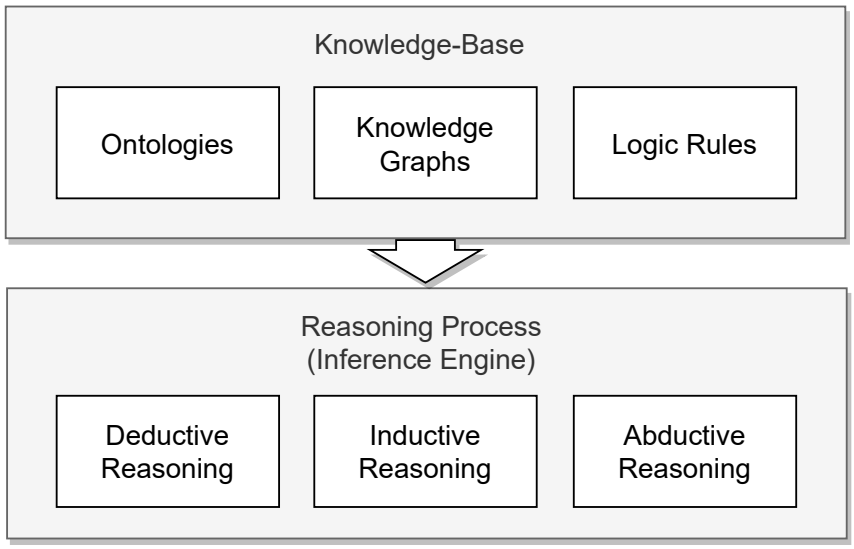

Figure 3.2: Methods of knowledge-based fault recovery (Source: [4] Page: 5)

3.1.1 Knowledge Base Representation

Ontologies describe the domain knowledge with its properties, attributes and relationships by defining classes of objects with nouns. For fault recovery, ontologies can be used to describe fault reasons, fault effects and recommended fault recovery actions.

Knowledge Graphs represent the knowledge in the form of graphs, where the subjects and objects can be mapped to vertices, and the predicates can be regarded as the path between the vertices. The main advantage of this representation is that it can be analyzed with well-researched graph algorithms. Knowledge graphs in fault recovery can be very useful for representing system-related information in the knowledge base, such as connections and dependencies.

Logic Rules encode the knowledge about the system and the faults in an $IF[conditions...]$, $THEN[consequences...]$ form. The representation is very straightforward, and it is a major benefit because it can be easily understood. Fault recovery can make great use of logic rules to encode the examples where the conditions are fault symptoms, and the consequences are recovery actions.

3.1.2 Reasoning Approaches

Deductive Reasoning is the process of reasoning from generalized statements to reach a specific logical conclusion. The result of the process is a valid solution, which means that the conclusion must be true if the premises are true. It is regarded as a top-down approach, starting from general statements to reach specific ones. Deductive reasoning will be used in the report to check the *hypotheses* produced by the inductive process. To illustrate how deductive reasoning works, consider the following example:

- Premise 1: Every star emits photons.
- Premise 2: The Sun is a star.
- Conclusion: The Sun emits photons.

Inductive Reasoning is the process of deriving generalized logical conclusions from specific observations. It starts with a set of observations and tries to find a hypothesis

that explains every observation without contradiction. It is regarded as a bottom-up approach, starting from specific statements to reach general ones. Inductive reasoning will be extensively used in the report, since the whole knowledge-based fault recovery method relies on it. More specifically, it will be used to extract the fault recovery rules from the provided examples with the help of Inductive Logic Programming (ILP) systems. To illustrate how inductive reasoning works, consider the following example:

- Premise: The Sun has risen in the east every morning up until now.
- Conclusion: The Sun will also rise in the east tomorrow.

Abductive Reasoning is concerned with finding the most likely conclusion which can be drawn from an incomplete set of observations. The most likely conclusion is not necessarily the one that is true. Its most important difference compared to deductive and inductive reasoning is that it favors one conclusion above others by demonstrating its likelihood. Abductive reasoning is widely used in the field of fault diagnosis [13]. To illustrate how abductive reasoning works, consider the following example:

- Observation: There is a global disturbance in radio communications.
- Most likely explanation: There has been a solar flare.

Here too, the explanation is not certain and might not be true since there could be other reasons why radio communication is not working, but it is the most likely explanation.

3.1.3 Advantages of Knowledge-Based Fault Recovery

There are several advantages of knowledge-based fault recovery compared to other approaches. One of them is that with the help of the knowledge base, the knowledge about the problems can be continuously accumulated and later on used during the inference process. This provides a great way to fine-tune the process over time to produce more appropriate recovery rules. There are two more major advantages, which can be attributed to how the underlying inference engines usually work. Firstly, the approach does not require huge amounts of data to be able to generalize, so it is suitable for both simple and complex systems. Secondly, since the whole process is based on logic, the inferred recovery rules contain all the logical information about why and when a given rule is applicable. This combination of the advantages makes the approach applicable to a wide range of problems.

3.2 The Workflow

Figure 3.3 illustrates the overview of the workflow. In the figure, squares depict input data to the processes, and processes are depicted with ellipses. The workflow has been designed to be flexible and easily adjustable, making it applicable to a wide range of fault recovery problems. It does not impose any restrictions nor requirements on how a system should be built up or what its architecture should be. This information has to be translated by the developers into a form that is suitable to the workflow's inputs.

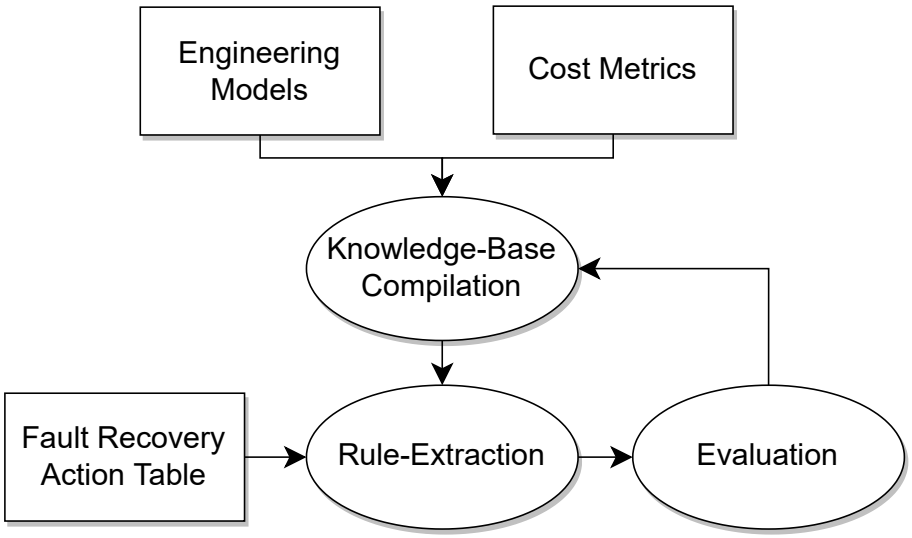

Figure 3.3: Workflow overview

3.2.1 Workflow Inputs

Engineering Models - Engineering models contain technical details about the system in which fault recovery is intended to be introduced. Every architectural data has to be provided to the fault recovery process through this input. This includes, among many others, the functional model, hierarchy, elements, dependencies and interconnections of the system. These engineering models are very important in the rule extraction process since they are used to provide the system's architecture.

Cost Metrics - Cost metrics allow adding external requirements and preferences to the rule extraction process to adjust the produced rules. There are numerous different scenarios for which a fault recovery process could be optimized. The environment in which a system operates, financial constraints, and time-critical requirements are all factors that might require the fault recovery process to be optimized for a specific target cost metric.

Scoring functions provide the basis for including cost metrics in the rule extraction process. The complexity of the scoring functions can range from simple algorithms that optimize the rules with respect to some predefined weights to complex algorithms that take into consideration multiple metrics and try to find the balance between them. Even though FastLAS supports scoring functions, this is not the case in most ILP systems, so the applicability of scoring functions is ILP system dependent.

Fault Recovery Action Table - The fault recovery action table is the most important input to the process. It provides the examples which are used by the underlying ILP system to extract the fault recovery rules. These examples come from the data that is extracted from the system during its operation. The examples have to be labeled — ideally, they should be the output of the manual classification process — and it must be exactly known what combination of symptoms in the system implies which recovery action.

The fault recovery examples can be represented in a tabular format, and from that, they can be mapped to logic rules. In a tabular format, the columns represent the different fault symptoms, a specific column represents the recovery actions suitable to handle the fault, and the rows represent the fault scenarios. In the logic rules,

the conditions are the fault symptoms, and the consequences are the recovery rules. This is the format of the examples that will be used throughout this report in later chapters.

3.2.2 Workflow Processes

Knowledge-Base Compilation - Knowledge-base compilation is the first process. Its inputs are the engineering models and the cost metrics. The goal of this process is to translate its inputs into a form that is easily processable by the ILP system. In the report, Answer Set Programming (ASP) will be used to represent the inputs for FastLAS.

Representing the engineering models in ASP is reasonably straightforward. How one would describe the system with plain words can be easily translated to ASP with the help of ontologies and logic rules. Additionally, knowledge graphs can be used to represent the hierarchy of the system and the connections between the system components. These representation methods provide a general way to describe a wide range of systems.

Cost metrics also have to be represented with ASP. Additionally, to include cost metrics in the FastLAS-based rule extraction process, they have to be defined with scoring functions in a FastLAS-specific way. This required representation of scoring functions will be detailed in later sections (Section 6.3).

Rule-Extraction - Rule-extraction is the second process. Its inputs are the fault recovery action table and the outputs from the knowledge-base compilation step. This is the process where the ILP task is executed to produce the recovery rules. At this point, the examples, the engineering models and the cost metrics all have to be transformed into logical representations with the help of a logic programming language. All the previous steps prepare the different inputs for this step.

In the report, this is the step where the FastLAS rule extraction task is executed. The extracted rules carry the intended characteristics that are defined with the scoring function. This is because the scoring function takes into consideration every provided information to produce a set of fault recovery rules that are optimized to the specified cost metrics.

Evaluation - Evaluation is the last process. When the rules have been extracted, it is time to examine the results and identify any incorrect rules. This process is usually fulfilled by a system expert. If something has to be adjusted, it can be easily carried out by making changes in the knowledge base compilation and rule extraction processes. If everything is correct after the rules have been examined, then the workflow is over, and the rules are ready to be used to provide the fault recovery actions for the system.

3.3 Building Blocks in the Report

In the report, the building blocks of the knowledge-based fault recovery process are Answer Set Programming, inductive reasoning with ILP, and deductive reasoning with clingo [14]. ASP is used to represent everything in the knowledge base in an easy-to-understand way with logic rules and knowledge graphs. ILP provides the ability to extract fault recovery rules from example data with the help of inductive reasoning. There are several ILP systems that could be used in the approach. This report uses the FastLAS ILP system, which has two main advantages over other ILP systems. The first is that it supports

scoring functions that can be used to fine-tune the reasoning process. The second one is that it is very well optimized for large-scale problems, and it can provide optimal solutions to much more complex problems than its counterparts. Clingo will be used in the examples to check the hypothesis — produced by the inductive process — with deductive reasoning.

The following sections will detail the steps and used technologies that are required to implement knowledge-based fault recovery. Firstly, there will be a detailed examination of the modeling approaches (Chapter 4) that provide useful information for the knowledge-based fault recovery process. Then the technologies used in the report will be discussed, which are ASP (Chapter 5), ILP (Chapter 6) and FastLAS (Section 6.3). After these chapters, the use cases and examples (Chapter 7) will give a thorough picture of how the approach and the workflow can be used to solve fault recovery problems.

Chapter 4

System Modeling

System modeling is an essential process related to the presented fault recovery approach as it provides the engineering models for the rule-extraction workflow. It is crucial to have correct and information-rich models about the examined system in order to produce appropriate recovery rules. This chapter details different aspects of system modeling that are important from the knowledge-based fault recovery aspect.

4.1 Architectural Models

Architectural models define the high-level unifying structures, and the behaviour of systems [15]. They provide *system components*, *connections* and *dependencies* with a set of rules, guidelines, and constraints that establish how the system parts work and fit together. They describe a wide-range of the system properties, such as system hierarchy, physical elements, components, relationships and links. Many system modeling methods work with the data provided by architectural models because they contain a comprehensive set of structural and behavioural information about the entire system. Architectural models play a key role in knowledge-based fault recovery at all stages since they provide a major part of the information about the problem that has to be solved.

4.2 Qualitative Modeling

Qualitative modeling [16] is a modeling approach concerned with the understanding and *discrete representation* of the continuous aspects of systems. Since knowledge-based fault recovery is a *discrete method* and requires *symbolic reasoning*, this modeling approach is very important. Three key principles govern qualitative modeling, which are discretization, relevance and ambiguity.

Discretization - Qualitative modeling, as its name implies, looks at continuous attributes and properties in a quantized, discretized way. This turns continuous attributes into entities which can be more easily represented and symbolically reasoned about. Additionally, discretization abstracts away unnecessary components from the continuous domain. As an example, consider a car. To solve different problems related to the car, it is not always required to know the exact velocity of it, only whether it is moving backwards, forwards, or it is standing. In such a case, the continuous property — which is the velocity of the car — can be abstracted away, and the state of the car can be represented with only three words: *backwards*, *standing* and $forwards$.

Discretization and abstraction are both very helpful in eliminating unnecessary system details. They can reduce the computational complexity of the tasks and keep the focus on the parts that truly matter to the specific problem. In knowledge-based fault recovery — especially when the approach is based on ILP systems such as FastLAS, which can not process real numbers — this property of qualitative modeling is exceptionally beneficial.

Relevance - The aspects of how the discretized values are defined are based on the constraints imposed by the problem that has to be solved. Qualitative values are to be relevant for some specific task. To continue the previous example, if the car dynamics are being tested above a certain positive speed limit, then the *backwards* and *standing* states are not relevant. As an other example, the car's velocity might be divided up to discrete intervals, and its dynamic behaviour is determined for these intervals. From a qualitative point of view, the car can be considered to produce the same behaviour inside the intervals, and the exact velocity is not important.

Ambiguity - With high-level abstraction, it is often the case that not enough information is available to accurately determine which of several possible behaviours will occur in a system. Due to this, predictions made by qualitative models can be ambiguous. However, this property of qualitative models can be beneficial and can be used to frame and identify the parts of a problem that are truly important. After the relevant parts have been identified, traditional modeling methods — such as mathematical and numerical models — can be used to clarify the ambiguity for the qualitative models. This reduces the required modeling effort and can help keep the focus on the parts that are important to solve the specific problem.

4.3 Information Flow Model

The information flow model is concerned with how *communication* takes place in a system [17]. Which components communicate, what kind of information is begin transmitted and due to this, what the dependencies are between the components. The information flow model provides a *logical representation* of the system being analyzed. This logical representation can be easily translated for the workflow with logic programming languages, providing valuable information that can be used during the rule extraction process. Knowing the flow of information is especially important to be able to account for *error propagation* inside a system. Errors caused by faults in one component can propagate to other connected components. Without knowing how the information flows inside the system, it would be very hard to consider these dependencies.

The flow of information in most cases can be represented with the help of graphs. When the direction of the communication is also important, directed graphs can be utilized. This is a significant property since it is then possible to analyze the information flow model with graph processing algorithms, which is a well-researched area. As an example, consider the graph depicted in Figure 4.1.

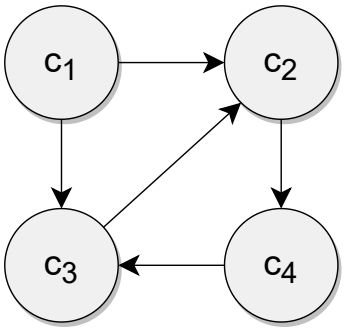

Figure 4.1: Example information flow model depicted as a graph

It is clearly visible how information flows inside the graph. Additionally, in logic programming, the connections between the components are very straightforward to represent. It only has to be stated as a fact that a connection between two components exists. After the connections have been declared, the graph can be processed. A very important notion in information flow modeling — which will also be used in the examples — is the transitive closure.

4.3.1 Transitive Closure

The transitive closure of a directed graph G_D tells for every vertex v_i whether vertex v_j is reachable from it, where $v_i, v_j \in G_D$ and $i \neq j$. Transitive closures can be used in fault recovery to account for error propagation and the effects from components that are not directly connected but through multiple intermediary components. A valuable property of transitive closures is that they have a *matrix representation*, which makes interpretation and processing very convenient. In the matrix representation, the rows and columns are the vertices of the graph. If a cell is marked in the matrix, then it means that there is a connection between the two vertices for which the row and column are marked.

As an example, Figure 4.2 depicts the matrix representation of the information flow model illustrated in 4.1. The columns represent the source components of the connections, and the rows represent the targets. The light grey cells represent when there is no connection between two components, the cells with and X represent when there is. Cells with blue backgrounds represent the directly connected components, and cells with red backgrounds depict the transitive connections.

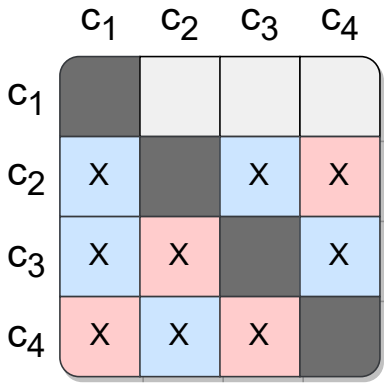

Figure 4.2: Transitive closure of the example information flow model

With this representation, it is very easy to reason about the distant connections in a system. The information flow model will be used in the examples (Section 7.5), and how the transitive closure can be calculated with ASP is discussed in Section 5.4.

Chapter 5

Answer Set Programming

Answer Set Programming (ASP) [18] is going to be used in this report to represent *fault recovery tasks*. The examples, background knowledge and optimization targets will all be translated into code with ASP.

ASP is a *declarative*, *rule-based language* for *knowledge representation* and *reasoning*, developed in the field of logic programming. ASP is used to represent and solve problems with the help of logic programs whose *answer sets* (*stable models*) correspond to solutions. It allows *domain* and *problem-specific knowledge* to be represented in an intuitive, easy-to-understand way. Additionally, thanks to its strong declarative nature, it supports rapid prototyping and the development of software for solving complex search and optimization problems.

5.1 Working with ASP

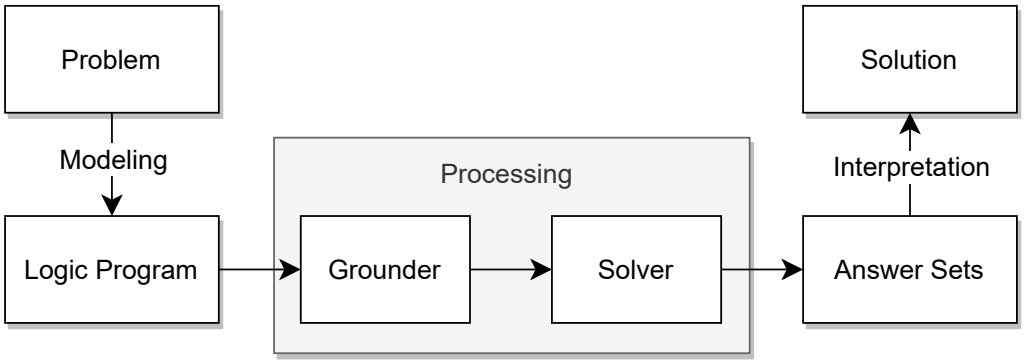

Figure 5.1: Answer Set Programming Process [14]

Working with ASP programs typically involves three stages, which are **modeling**, **processing** and **interpretation** (Figure 5.1).

Modeling is the first stage. Here the problem is translated into a logical representation — conforming to ASP rules — by the programmer. The fundamental approach to writing ASP programs is based on the "generate-and-test" strategy. First, a group of rules are created whose answer sets correspond to candidate solutions. Then, the second group of rules is added, that eliminates candidates which represent invalid solutions.

Processing comes after modeling, and it consists of two sub-parts, namely *grounding* — also called as *variable replacement* — and *propositional solving*.

Grounding is the process of efficiently replacing a predicate program P with a possibly small propositional program whose answer sets are equivalent to P. Given an input program with first-order variables, the grounder computes an equivalent ground (variable-free) program. A naive approach for grounding is to generate all possible propositional programs. However, this is inefficient and often impossible due to computational complexity. Intelligent grounding techniques incorporate a wide range of optimizations such as rewriting, partial evaluation and approaches borrowed from database technology.

Propositional solving is the process of finding answer sets for the propositional programs produced by the grounder. Most solvers use methods developed in the field of satisfiability solving (SAT), such as local search, backtrack search and formula manipulation. Additionally, the answer set search can be further improved by sophisticated search heuristics and techniques like backjumping and clause learning.

Interpretation is the final stage. After processing is finished and the answer sets have been found the output is ready to be examined. Additionally, errors or non-desired solutions can be identified here to make further adjustments to the model provided in the first stage.

5.2 Negation in ASP

ASP employs two kinds of negations, which are Negation as Failure (NAF) and logical negation [19]. NAF is a non-monotonic, logic programming related inference rule, and it is used to derive that a proposition p does not hold (*not* p) from the failure to derive that specific proposition p. Logical negation is the standard negation approach, which takes a proposition p to $\neg p$. There is a subtle but very important difference between the two negations. NAF handles the absence of information about a proposition as a distinct state, while logical negation requires explicit information about the proposition. To illustrate this difference, consider a scenario where a street has to be crossed under the condition that there is no approaching car. With NAF, it is okay to cross if there is no information about the approaching cars, meaning that it cannot be derived that a car is coming. With logical negation, it is okay to cross only if there is explicit information stating that a car is not coming.

The usage of both negations in ASP leads to an additional fundamental ASP concept, which is called the Local Closed-World Assumption (LCWA) [19]. LCWA assumes that a predicate does not hold whenever there is no evidence that it does. This means that everything that is used under the LCWA has to be explicitly stated to be included during the processing. As an example, even numbers which are used by a specific ASP program have to be explicitly stated, or if they are not stated, then the ASP program can not work with them.

5.3 Basics and Building Blocks

The basic building blocks of ASP programs [20] are *atoms*, *literals*, and *rules*. *Atoms* are elementary propositions that may be true or false; *literals* are atoms a and their negations *not* a. In addition to the basic building blocks, there are several language extensions in

ASP, such as *Disjunctive Rules* and *Aggregate Functions*. This section will only discuss the ones that are important for understanding the report.

5.3.1 Facts and Rules

Rules are expressions in the following form:

$$r \leftarrow b_1, ..., b_m, \; not \; c_1, ..., \; not \; c_n. \tag{5.1}$$

where r and all b_i's and c_j's are atoms. Intuitively, rule (5.1) is a justification to "establish" or "derive" that r (the so-called *head*) is true, if all literals to the right of $\leftarrow$ (the so-called *body*) are true in the following sense: a non-negated literal b_i is true if it has a derivation, a negated one, $not \; c_j$, is true if the atom c_j does not have one.

As a concrete example, consider the following rule:

$$engine\_working \leftarrow turned\_on, \; not \, broken. \tag{5.2}$$

It states that, *engine_working*, the rule's *head*, is true if the engine is turned on and there is no reason to think that the engine is broken.

Facts are rules with an empty, or no body at all. They are represented in the following forms:

$$\begin{gathered} turned\_on \leftarrow . \\ turned\_on. \end{gathered} \tag{5.3}$$

Both rules in 5.3 mean the same thing, that *turned_on* is true in all circumstances without any condition.

It is important to point out again that in ASP *not* is not a standard negation operator. Instead, it is meant to stand for a modality "non-derivable". Consider the following example:

$$\begin{gathered} turned\_on. \\ engine\_working \leftarrow turned\_on, \; not \, broken. \end{gathered} \tag{5.4}$$

In example 5.4 for *engine_working* to be derived, *turned_on* should be derived, and it is given as a fact. While *broken* should not be derived, since the program, which describes what is known, has no rule to derive *broken*.

5.3.2 Disjunctive Rules

Disjunctive rules can be used as generators. Consider the following rules with their answer sets below the horizontal line:

$$\begin{gathered} x(1). \; x(2). \\ 0 \; \{y(X) : x(X)\} \; 2. \end{gathered} \tag{5.5}$$

$$\begin{gathered} \{x(1),\; x(2)\} \\ \{x(1),\; x(2),\; y(1)\} \\ \{x(1),\; x(2),\; y(2)\} \\ \{x(1),\; x(2),\; y(1),\; y(2)\} \end{gathered}$$

The numbers on the two sides of the curly brackets indicate the minimum and maximum number of components that can be included in a generated rule. If the number on the minimum or maximum side is omitted they default to zero or the maximum number of combinations respectively.

5.3.3 Integrity Constraints

Integrity constraint are used to eliminate answers sets that do not satisfy the requirements of the body of the constraint.

$$\begin{gathered} x(1).\ x(2). \\ 0\ \{y(X) : x(X)\}\ 2. \\ :-\ not\ y(1). \end{gathered} \tag{5.6}$$

$\{x(1),\ x(2),\ y(1)\}$
$\{x(1),\ x(2),\ y(1),\ y(2)\}$

In example (5.6) there is a generator which produces the same answers sets as in example (5.5). In addition to this there is an integrity constraint which eliminates answers sets that do not include *y(1)*.

5.3.4 Aggregate Functions

[21] Aggregate functions are used on a group of values to produce an output. Such functions in ASP are *#sum*, *#count*, *#min* and *#max*.

$$\begin{gathered} x(1).\ x(2). \\ 0\ \{y(X) : x(X)\}\ 2. \\ :-\ \#count\{C : y(C)\} < 2. \end{gathered} \tag{5.7}$$

$\{x(1),\ x(2),\ y(1),\ y(2)\}$

In example (5.7) the number of atoms in the form of *y(C)* are counted and those answer sets are eliminated which do not contain at least two of them.

5.4 Transitive Closure in ASP

To showcase how ASP works, this section details a small example that calculates the transitive closure (Figure 4.2) of the information flow model depicted in Figure 4.1. The first step to solve this problem with ASP is to define the connections between the components as facts. The facts are depicted in Listing 5.1.

```
% Facts of the example information flow model
connection(c1,c2).
connection(c1,c3).
connection(c2,c4).
connection(c3,c2).
connection(c4,c3).
```

Listing 5.1: Facts of the example information flow model

The second step is to process the connections with the help of two simple rules depicted in Listing 5.2. The first rule adds the $closure(X, Y)$ conclusion to the background knowledge based on the connections. The second rule then adds a $closure(X, Z)$ conclusion if $X \neq Z$ and there exists a connection between X and Y and there is a closure in the background knowledge in the form of $closure(Y, Z)$.

```
% Transitive closure calculation
closure(X,Y) :- connection(X,Y).
closure(X,Z) :- X != Z, connection(X,Y), closure(Y,Z).
```

Listing 5.2: Transitive closure calculation

The two rules are very straightforward and simple. They clearly highlight the advantages of logic programming and ASP. When executed, the result depicted in Figure 4.2 is received.

Chapter 6

Inductive Logic Programming

Inductive Logic Programming (ILP) [22] is an area formed at the intersection of Machine Learning and Logic Programming. ILP programs aim to learn *a set of logical rules* — called a *hypothesis* — based on positive and negative examples while considering a supplied background knowledge. The main characteristics of ILP systems from the machine learning side is their goal to *learn and find patterns* in data, then *generalize and make predictions* about previously unseen datasets. From the logic side, their goal is to find connections in datasets that are *logically deducible* and *explainable*.

6.1 Advantages of ILP Systems

The primary advantage of ILP systems is their ability to give logical explanations about the rules they learn. Today, most artificial intelligence and machine learning solutions function like *black-boxes*. The way they internally work is not transparent. This property of traditional AI/ML algorithms can be a drawback, and in many engineering fields, it is not acceptable to have algorithms that cannot be fully explained. On the other hand, ILP systems inherently possess this ability, as they are based on *formal logic*.

Another positive trait of ILP systems is that they are able to generalize from a smaller number of examples. Thanks to this, the training process can be executed faster, which can be very beneficial in many scenarios, for instance, in time-critical situations. Examples for the training process must be represented with the same formal logic method.

A third very beneficial property of ILP systems is their ability to efficiently encode and utilize background knowledge, which is essential for achieving intelligent behaviour. The background knowledge also has to be represented with the same logic-based formalism so that it is straightforward to incorporate it in the induction task. Traditional AI solutions generally do not allow this in such a smooth manner out-of-the-box.

6.2 ILP Background

In general, an ILP task [22] is a tuple in the form $T = \langle B, S_M, \langle E^+, E^- \rangle \rangle$ where B is a background knowledge, usually in the form of a logic program, S_M is a hypothesis space, which defines the set of rules that are allowed to be in a hypothesis, and E^+ and E^- are positive and negative examples, respectively. A hypothesis — a subset of the hypothesis space — is the inductive solution of the ILP task. The goal of the ILP task is to find a hypothesis H in the hypothesis space S_M that has at least one answer set (when combined with B) that extends every positive example (E^+), and none of the negative examples

(E^-). A hypothesis $H \subseteq S_M$ is an *inductive solution* of T if and only if:

$$1.\ \forall e^+ \in E^+\ \exists A \in AS(B \cup H) \text{ such that } A \text{ extends } e^+$$
$$2.\ \forall e^- \in E^-\ \nexists A \in AS(B \cup H) \text{ such that } A \text{ extends } e^-$$

1. states that for every positive example, there must be a rule in the set of all answer sets of the union of the background knowledge and the hypothesis that covers the example.
2. states the opposite that there must be no rule in the set of all answer sets of the union of the background knowledge and the hypothesis that covers any negative example.

6.3 FastLAS

FastLAS (Fast Learning from Answer Sets) is a system for Inductive Logic Programming [1]. It is based on a restricted version of the general ILP approach presented in Section 6.2, called Observational Predicate Learning (OPL), where the predicates in the examples coincide with the predicates defined by the hypothesis. It has two main advantages compared to other ILP systems, such as its predecessor, ILASP [23]. Firstly, most ILP systems assume that the hypotheses with the lowest number of literals are the best solutions. In contrast, FastLAS supports *scoring functions* that can be used to score *hypotheses* according to predefined criteria. This makes FastLAS applicable to an even wider set of problems and provides great flexibility. Secondly, it is specifically designed to be *scalable* with respect to the hypothesis space — the set of all rules which can appear in the hypothesis. This is achieved by computing a smaller subset of the hypothesis space that is guaranteed to contain at least one optimal solution with respect to a supplied scoring function. This search space can be orders of magnitude smaller than the full hypothesis space.

6.3.1 Hypothesis Space

Many ILP systems make use of mode declarations [24] to specify hypothesis spaces, and FastLAS follows this approach. A *mode bias* is defined as a pair of sets of mode declarations $\langle M_h, M_b \rangle$, where M_h is called the *head*, and M_b is called the *body mode declaration*. A mode declaration is a ground atom whose arguments can be *placeholders*. A placeholder is a term *var(t)* or *const(t)* for some constant term *t*. A placeholder can be replaced by any variable or constant (respectively) of *type* t. An atom a is compatible with a mode declaration m if each of the placeholder variables and placeholder constants in m has been replaced by a variable or a constant of the correct type.

There are two types of mode declarations in FastLAS: *#modeh* for the head declarations and *#modeb* for the body declarations. Mode declarations can be specified with a *recall*, which is an integer that specifies the maximum number of times the mode declaration can be used in a single rule. The maximum number of variables in each rule can also be specified with the predicate *#maxv*.

As an example, consider the hypothesis space in Listing 6.1. There is a single head and three body mode declarations. All three body mode declarations can only be used once in any rule since their recall is set to 1. The maximum number of variables in any rule is set to 1 (line nr. 5), so the body mode declaration, which contains one variable (line nr. 2), is allowed to be in the search space. However, the one with two variables (line nr. 3) is not.

```
1 #modeh(h).
2 #modeb(1,b1(var(type1))).
3 #modeb(1,b2(var(type1),var(type2))).
4 #modeb(1,b3).
5 #maxv(1).
```

Listing 6.1: Example hypothesis space

When processed, Listing 6.1 leads to the search space in Listing 6.2. As expected, every body mode declaration is contained at most once in any rule. Besides this, no rule contains the body mode declaration that has two variables.

```
1 ~  :- b1(V1).
1 ~  :- not b3.
1 ~ h.
1 ~  :- b3.
2 ~  :- b1(V1); not b3.
2 ~ h :- not b3.
2 ~ h :- b1(V1).
2 ~ h :- b3.
2 ~  :- b3; b1(V1).
3 ~ h :- b3; b1(V1).
3 ~ h :- b1(V1); not b3.
```

Listing 6.2: The example hypothesis space's search space

6.3.2 Training Data

In FastLAS, the examples from the training data have to be represented in a FastLAS specific form. The required format is $e = \langle e_{id}, e^{inc}, e^{exc}, e_{ctx} \rangle$, where e is the example, e_{id} is an identifier for the example, e^{inc} is the set of *inclusions*, e^{exc} is the set of *exclusions* and e_{ctx} is an ASP program called as *context*. An inclusion is a head mode declaration for which the example is positive, and an exclusion is a head mode declaration for which the example is negative.

Listing 6.3 depicts a concrete FastLAS example. The *#pos* keyword marks the beginning of the example. The first component in the example is the identifier ($id(1)$), the second component contains the inclusions ($\{turned\_on\}$), the third component contains the exclusions ($\{turned\_off\}$) and the final component is the context, which is a small ASP program.

```
#pos(id(1),{turned_on},{turned_off},{
    value(10).
}).
```

Listing 6.3: FastLAS example format

6.3.3 Scoring Function

One of the main advantages of FastLAS is that it supports the usage of scoring functions so the hypothesis can be adjusted with respect to predefined criteria. This is the feature that allows FastLAS to be used in domain-specific scenarios where the shortest rules are not the most desirable, such as in fault recovery.

A FastLAS scoring function is a function $S : \mathit{Programs} \times T_{OPL} \to \mathbb{R}_{\geq 0}$, where *Programs* is the set of all ASP programs and T_{OPL} is the set of all OPL tasks. A hypothesis $H \in \mathit{Programs}$ is an optimal solution of task $T \in T_{OPL}$ with respect to scoring function S if and only if there is no H' such that $S(H',T) < S(H,T)$.

6.3.4 FastLAS Algorithm

The FastLAS algorithm consists of four main steps: (1) initial construction; (2) generalization; (3) optimization; and (4) solving. The role of the initial construction step is to calculate a subset S_M^1 of the hypothesis space S_M, that is guaranteed to contain at least one solution of the learning task if the task is satisfiable. In most cases, S_M^1 contains very specific rules since each rule is calculated to cover a single example in the set of all examples. The second step, generalization, is responsible for finding rules which are subrules of one or more rules in S_M^1, leading to a larger hypothesis space S_M^2. In the optimization step, each rule is examined in S_M^2, and an optimized version is calculated with respect to the scoring function, which results in hypothesis space S_M^3 that is guaranteed to contain at least one optimal solution. The final step, solving, is responsible for finding a subset of the hypothesis space S_M^3 that covers all examples and which is guaranteed to be an optimal solution of the entire learning task.

6.3.5 FastLAS Example

FastLAS has been used to solve real-world problems in many fields, such as Natural Language Processing, Event Detection and Security. To demonstrate how FastLAS can be used, this section discusses how it was used in the security domain to learn security policies.

The research in Paper [25] was focusing on how Symbolic Learning with the help of FastLAS can be beneficial in the privacy and security domain. Machine learning has become a very valuable tool in the field. However, it is not always applicable. Many security-related problems require the models to be explainable to exactly know what the security and privacy rules that are being learnt contain. The paper discussed a whole system developed for anomaly detection for access requests based on FastLAS and domain-specific scoring functions. The received results were very positive and yielded more accurate results in comparison to traditional AI/ML approaches.

The two main properties of FastLAS that made it applicable in the field were: (1) the learnt security rules were explainable, which is of great importance in the field of security; (2) with the help of scoring functions, the level of generality of the learnt rules could be easily adjusted. Security rules that are too general or too specific are not desirable since they generate a high number of false allows and false denies. The right balance has to be found to generate the least amount of errors.

In the paper, the learning process was divided into three separate sections that covered different aspects of the policy learning task. The exact parts and their respective goals are detailed in the following section:

Input pre-processing - The first part was for input pre-processing. The inputs were network access requests that contained traditional HTTP request data, such as source and destination IP addresses and ports in JSON format. This had to be converted to a flattened representation that complied with the input example format required by FastLAS (6.3.2).

Learning task generation - The second part was for generating the learning task for FastLAS. The data from the network requests had to be translated to head and body mode declarations, and additional background knowledge had to be extracted and properly represented in the hypothesis space for FastLAS. Additionally, the scoring function was defined in this part.

Learning and output processing - The final part consisted of running the generated FastLAS learning task and processing the output. During the processing, the outputs have been converted to proper security rules based on the learned hypothesis so that they could be evaluated on the test data.

The results of the learning process have been compared to the results of three different anomaly detection baselines, which were: (1) One-Class Support Vector Machine, (2) Isolation Forest and (3) Local Outlier Factor. The results from the comparisons showed that FastLAS was the most accurate. Its True Positive Rate (TPR) was the highest and False Positive Rate (FPR) was the lowest among all solutions.

Chapter 7

ILP Supported Fault Recovery

Now that the required background materials have been presented, this chapter showcases the applicability of the approach through different use cases. The use cases will detail the knowledge-based fault recovery process starting from the analysis of the inputs and then move on to the logical representation of the fault recovery action table, system models and scoring functions. In the end, it concludes with the ILP-supported rule extraction process. The different parts of the workflow are going to be incrementally introduced one-by-one. There will be four use cases:

1. The first use case is going to showcase the *generalization* capability of ILP systems and how it can be beneficial to fault recovery. This approach is useful when there is no additional information and model about the examined system.
2. The second use case is going to build on the first use case, and it will introduce the concept of recovery rule *preferences* with the usage of *weights* and *scoring functions*.
3. The third use case is going to combine the first two use cases and extend them with *background knowledge*.
4. The fourth use case is going to showcase the *consistency checking* ability of the approach with the help of the background knowledge.

The inputs required by the workflow — such as fault recovery action tables — are going to be provided in an already processed and abstracted form so that the main focus stays on the parts that truly matter from the workflow's point of view. These parts are fundamental building blocks that can be arbitrarily tweaked and combined to fit the needs of real-world complex systems.

It is important to note that the scoring functions presented in the examples might not be optimal. Their point in the report is to illustrate how they can be used and not to create the most optimal scoring functions.

7.1 Use Case Background

Figure 7.1 represents the system which the examples are going to be based on. There are two system elements, namely units and components. Units are illustrated with rectangles while components with circles. Units are allowed to contain other units and components inside themselves, in contrast to this, components are atomic elements that are not broken down any further and cannot contain additional system elements. Units containing other elements represent hierarchy in the system and directed edges between components illustrate the flow of information and component dependencies.

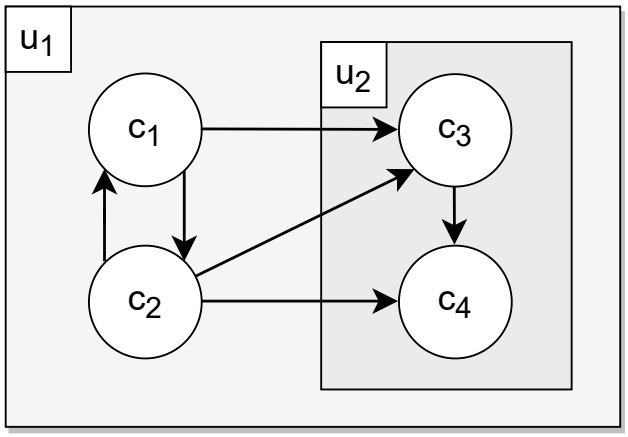

Figure 7.1: Example system architecture

In the use cases the system elements will be abstracted away, in order to keep the focus entirely on the recovery actions. Figure 7.2 represents the recovery actions associated to the elements. The hierarchy and the connections remain the same, the only thing that changes is that system elements are referred to with recovery actions, so whenever a particular combination of symptoms arises that signals the fault of a system element, it can immediately be associated with the recovery action which belongs to the element.

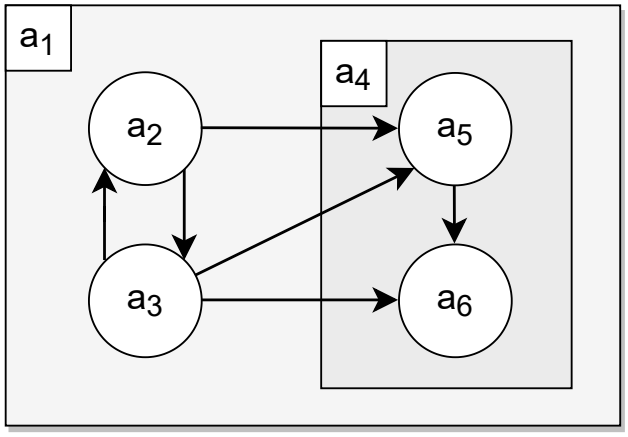

Figure 7.2: Example system recovery actions

Table 7.1 depicts the fault recovery action table (Figure 3.3) representing input symptom combinations and their respective recovery actions for the example system. Data included in the table can be considered, for instance, to be the output of a fault diagnosis step processed by a system expert.

The table contains symptoms and corresponding fault recovery actions based on the combination of the symptoms. There are four symptoms ($s_{1\text{-}4}$), 10 different combinations of the symptoms, and 6 fault recovery actions ($Action_{1\text{-}6}$). In one scenario a symptom can take up only one value out of four possible values. The possible values are: "*True*" (T), "*False*" (F), "*Don't know*" (-) and "*Don't care*" ($\varnothing$). "*True*" (T) and "*False*" (F) indicate that the symptom is definitely present or not present, respectively. These two values are usually the output of system tests that provide an exact answer whether a symptom is present or not. "*Don't know*" (-) is used to indicate that nothing is known about the symptom, but if there was information about it, then maybe there would be a different,

more applicable recovery action for the scenario (Section 5.2). "*Don't care*" ($\varnothing$) is used to indicate that even if there was information about the symptom, the same recovery action would be used (Section 5.2).

Symptoms				**Action**
s_1	**s_2**	**s_3**	**s_4**	
F	-	-	-	1
T	-	F	F	2
T	-	-	-	3
F	T	F	F	3
F	T	-	-	3
-	-	T	-	4
T	-	-	-	5
F	T	-	-	5
-	-	-	T	5
T	-	T	F	6

Table 7.1: The example fault recovery action table

The exact symptoms and recovery actions are not of importance in the examples, the point is only to be able to differentiate between them. In a real-world system, symptoms can be, for example, system malfunctions, hardware faults, erroneous software responses and recovery actions can be hardware replacements and software updates.

7.1.1 Fault Recovery Action Table Representation

Since the input examples are mutually used in every following example, as an initial step they will be represented with the help of logic programming so that FastLAS can use them during the rule extraction/learning process. Symptom combinations appearing in each line of the table (Table 7.1) provide the attributes for the positive examples of the actions appearing at the end of the same line. Actions not appearing at the end of the line will have this symptom combination marked as negative from their point of view.

```
#pos(id(1),{action_1},{action_2,action_3,action_4,action_5,action_6},{
    s_1(f).
    s_2(dont_know).
    s_3(dont_know).
    s_4(dont_know).
}).
```

Listing 7.1: Scenario #1 in ASP

Listing 7.1 shows the first scenario from Table 7.1, represented with ASP for FastLAS. The *#pos* keyword marks the beginning of the example. The first component in the example is the identifier (e_{id}), the second component contains the actions for which this scenario is positive (e^{inc}), in this case it means that if these symptoms appear, then *action_1* is a possible recovery action. The third component contains the actions that are not suitable to recover from this fault (e^{exc}), in this case everything is listed besides *action_1*. The final component is the context of the example (e_{ctx}), which contains the symptoms as they appear in Table 7.1.

All the other lines of the table can be converted to FastLAS processable examples based on this methodology. The process is very straightforward and easily understandable, which

is one of the advantages of the logic-based approach. Now that the fault recovery action table is represented properly, everything is provided to move on to the examples.

7.2 Use Case 1 - Generalization

The generalization property can be very beneficial in the fault recovery domain. The input data provided in Section 7.1 contains scenarios that are too exact. If there is a fault in the system that produces a symptom combination that is not included in the table, then the recovery actions can not determined. In such cases, generalization can help in finding recovery actions.

FastLAS — and ILP systems in general — inherently possesses the ability to generalize from the examples, thanks to the algorithms it is built upon. By default, it looks for a minimal set of rules that cover all positive examples and none of the negatives. The only thing that has to be done to get the generalized version of the table is to define the full learning task with mode declarations, then execute it with FastLAS. The rule heads for which FastLAS will have to learn the bodies are going to be the recovery actions ($Action_{1\text{-}6}$), and the bodies are going to be the symptoms ($s_{1\text{-}4}$). Listing 7.2 shows these head and body mode declarations.

```
% Head mode declarations
#modeh(action_1).
#modeh(action_2).
#modeh(action_3).
#modeh(action_4).
#modeh(action_5).
#modeh(action_6).

% Body mode declarations
#modeb(s_1(const(observation))).
#modeb(s_2(const(observation))).
#modeb(s_3(const(observation))).
#modeb(s_4(const(observation))).

% Constant declarations
#constant(observation, dont_know).
#constant(observation, t).
#constant(observation, f).
```

Listing 7.2: Mode declarations for FastLAS

After running FastLAS, the hypothesis depicted in Table 7.2 is received. The most notable difference is that there are now symptoms that have "*Don't care*" values, and this is the basis for generalization.

Symptoms				Action
s_1	s_2	s_3	s_4	
F	-	ø	ø	1
T	ø	F	ø	2
T	ø	ø	-	3
ø	T	ø	ø	3
ø	ø	T	-	4
ø	ø	ø	T	5
ø	T	ø	-	5
T	ø	ø	-	5
ø	ø	T	F	6

Table 7.2: The generalized version of the recovery actions

To understand how this and the initial table differ, consider the symptom combination in Table 7.3. For this combination the initial fault recovery action table did not provide a recovery action. To check whether the hypothesis provides any recovery actions, it must be evaluated on the example. To do this, the example and the hypothesis together have to be solved with an ASP system, such as clingo [26].

Symptoms			
s_1	s_2	s_3	s_4
T	-	F	T

Table 7.3: Example symptom combination to showcase generalization

Listing 7.3 depicts the clingo task which contains the hypothesis and the uncovered symptom combination. After running clingo, the result consists of two actions that can be used for fault recovery, $Action_2$ and $Action_5$. These recovery actions were not available in the initial fault recovery action table. Here it is important to point out that with the clingo task a deductive reasoning process is being conducted. There is a hypothesis, which contains general observations, since it is the result of the inductive reasoning process. And there is an uncovered, specific example for which the recovery actions have to be found based on the general hypothesis.

```
% Hypothesis
action_1 :- s_1(f), s_2(dont_know).
action_2 :- s_3(f), s_1(t).
action_3 :- s_1(t), s_4(dont_know).
action_3 :- s_2(t).
action_4 :- s_3(t), s_4(dont_know).
action_5 :- s_4(t).
action_5 :- s_2(t), s_4(dont_know).
action_5 :- s_1(t), s_4(dont_know).
action_6 :- s_4(f), s_3(t).

% Uncovered symptom combination
s_1(t). s_2(dont_know). s_3(f). s_4(t).
```

Listing 7.3: Clingo task with the hypothesis and the uncovered symptoms

The example clearly showcases the generalization ability of the approach. Based on the initial fault recovery action table it was able learn a generalized, logically explainable hy-

pothesis that can be used to provide recovery actions even in scenarios when the symptom combinations are uncovered.

7.2.1 Avoiding Over-Generalization

As detailed in the previous example (Section 7.2), by default FastLAS looks for a minimal set of rules that cover all the positive examples and none of the negatives. In our analogy this corresponds to the highest level of generalization. In some use-cases this might be undesirable and has to be avoided, for instance, when the produced rules are too general and a lot of rules would be applicable based on a few symptoms. In the simplest case, the use of integrity constraints (5.3.3) that require the hypothesis to contain specific head and body element combinations is a feasible approach. However, for complex systems where a more flexible approach is required, the favored way to solve this involves the usage of weights and preferences, a concept detailed in the following section.

7.3 Use Case 2 - Weights and Preferences

Weights and preferences provide a great way to tweak the hypothesis to fit arbitrary needs. These preferences usually come from external sources. This example is going to discuss how Cost Metrics (Figure 3.3) can be included in the workflow to provide a way to prefer hypotheses based on different criteria.

Preferences are introduced into the workflow with the help of the knowledge-base and scoring functions. First, weights have to be defined for the different possible fault recovery actions, and they have to be included in the knowledge-base. After this, a scoring function has to be developed that takes these weights into account. Just as in the previous example, all the weights and the scoring function have to be represented with the help of logic programming.

This example is going to build upon the learning task created in the first example (Listing 7.2). To start with, lets define weights for the different actions (Table 7.4). These weights could be defined for a number of purposes, such as the required steps to execute the action, or the cost associated with executing the action. In our example the weights are going to represent the time it takes to execute the action.

Action	**Execution time**
action_1	5
action_3	7
action_2	25
action_4	30
action_5	50
action_6	70

Table 7.4: Actions and their respective weights (Execution time)

Based on the weights, we define a scoring function which prefers more general rules in the hypothesis for actions that have a weight below a predefined threshold value, and produces more specific rules for actions which have weights above the threshold. In the time analogy, this means that if an action has an execution time below the threshold, then it should be generalized, on the other hand, actions with long execution times are going to be preferred in more specific scenarios.

The scoring function can be described with the formula in Equation 7.1, where $weight(A) \geq 0$, $threshold \geq 0$ and $attrs(r_b) > 0$. The left side of the equation, $score_A(r)$, represents the score of action A with respect to rule r. The right side represents the time analogy theory discussed in the previous paragraph, where $ReLu(x)$ is the Rectified Linear Unit, $weight(A)$ is the weight of the action, $threshold$ is the value which can be used to fine tune the generalization and $attrs(r_b)$ is the length of the body of the rule.

$$score_A(r) = \frac{ReLu(weight(A) - threshold)}{attrs(r_b)}$$
$$ReLu(x) = max(0, x) \tag{7.1}$$

Given the scoring function, FastLAS will find the hypothesis which contains rules for which the sum of scores is minimal from all hypotheses. Formally this can be represented with Formula 7.2, where S is the entire search space, H_A are the hypotheses containing the actions with the rules, and H_A^* is the best hypothesis with the minimal sum of scores.

$$H_A^* = \underset{H_A \in S}{argmin} \sum_{r_i \in H_A} score_A(r_i) \tag{7.2}$$

After the weights and the scoring functions have been defined, the next step is to represent them in ASP for FastLAS (Listing 7.4). The first block of code is used to set the generalization parameter, the second block generates the used numbers as facts and defines the *ReLu* function and the final block represents the scoring function. In the example the generalization threshold has been set to 10. This means that $Action_1$ and $Action_3$ are going to have a more general rules since they both have weights below the threshold and $Action_2$ and $Action_{4-6}$ are going to have more specific rules, because they have weights above the threshold.

```
% Generalization threshold
#bias("gen_threshold(10).").

% Number generation
#bias("number(-1000..1000).").
% ReLu function
#bias("relu(X,0) :- number(X), X <= 0.").
#bias("relu(X,X) :- number(X), X > 0.").

% Scoring function
#bias("penalty(Penalty, Action) :- in_head(Action), action_weight(Action, Weight),
       BodyLen = #count{B:in_body(B)}, gen_threshold(Gt),
       relu(Weight - Gt, ReLu), Penalty = ReLu/BodyLen.").
```

Listing 7.4: The scoring function which uses the weights

Executing the previously defined FastLAS task yields the results listed in Table 7.5. The most notable property is that $Action_2$ and $Action_{4-6}$ do not contain *"Don't care"* values, additionally, they have more specific and longer rules. In contrast to this, $Action_1$ and $Action_3$ contain *"Don't care"* values and generally they have shorter rules.

In the time analogy, Table 7.5 means that the actions below the thresholds will be preferred because of their shorter execution times. These rules can be considered to be useful in a time-critical scenario, for instance, if the actions are imagined as element replacements in the system, this could mean that unit 1 and component 3 have shorter replacement times, thus the actions related to them are preferred.

Symptoms				Action
s_1	s_2	s_3	s_4	
F	-	ø	ø	1
ø	T	ø	ø	3
T	ø	ø	-	3
T	-	F	F	2
-	-	T	-	4
-	-	-	T	5
F	T	-	-	5
T	-	-	-	5
T	-	T	F	6

Table 7.5: The weighted version of the recovery actions (The horizontal line depicts the threshold)

7.4 Use Case 3 - Engineering Models

This example is going to showcase the way how Engineering Models (Figure 3.3) can be included in the background knowledge to adjust the rule extraction process. The hierarchy from Figure 7.1 and Figure 7.2 is going to be utilized. The goal is to use the hierarchy of the system elements to set up a recovery action preference for the elements that are on a higher level in the hierarchy and contain more system elements, so in this case recovery actions that belong to units will be preferred. The current goal is similar to the one in the previous example, however, there the weights had to be set manually. Now instead, this information is going to be extracted from the engineering models, thus the process is going to be more scalable.

As the first step, the hierarchy in Figure 7.1 has to be translated to ASP. There are two units, the first unit contains two components and the other unit, while the second unit contains two components. The power of ASP is going to be leveraged to simplify this step by specifying only the elements that are directly contained by the units and leave the calculation of the transitive relations to ASP rules. This representation is depicted in Listing 7.5.

```
% Background knowledge representing system hierarchy
#bias("contains(u1,c1).").
#bias("contains(u1,c2).").
#bias("contains(u1,u2).").
#bias("contains(u2,c3).").
#bias("contains(u2,c4).").

% Background knowledge processing - transitive relations, contained elements
#bias("contains(X,Z) :- contains(X,Y), contains(Y,Z).").
```

Listing 7.5: The hierarchy represented with ASP in the background knowledge

The first block of code describes the system elements that are directly contained. The second block is a single ASP rule which is used to find every other contained element. This part also highlights the advantages of ASP, since a single rule can help us save a lot of time.

Now that the hierarchy is defined, it is time to specify the scoring function that is going to take it into account during the rule extraction process. Equation 7.3 describes the scoring function, where $contains(A_E) \geq 0$, $threshold \geq 0$, $attrs(r_b) > 0$, $score_A(r)$ is the score

of action A with respect to rule r, $ReLu(x)$ is the Rectified Linear Unit, *threshold* is the limit for the number of contained elements above which the rules for the actions will be generalized, $contains(A_E)$ is the count of the contained elements by element E to which action A belongs based on Figure 7.1 and Figure 7.2 and $attrs(r_b)$ is the length of the body of the rule.

$$score_A(r) = \frac{ReLu(threshold - contains(A_E))}{attrs(r_b)} \qquad ReLu(x) = max(0, x) \tag{7.3}$$

Listing 7.6 shows the discussed scoring function represented with ASP for the FastLAS learning task. The first block of code is used to set the generalization parameter, the second block generates the used numbers as facts and defines the *ReLu* function. The last block is the scoring function, the $(ReLu * 10)$ part in the scoring function's numerator is used only because FastLAS does not handle real numbers, and without a scaled numerator the results would be erroneous. The threshold value in the example has been set to 2, in the hierarchy of the system this implies that recovery actions that belong to units will be generalized, since both units in the example contain two or more elements.

```
% Generalization and falloff parameters
#bias("gen_threshold(2).").

% Number generation
#bias("number(-1000..1000).").
% ReLu function
#bias("relu(X,0) :- number(X), X <= 0.").
#bias("relu(X,X) :- number(X), X > 0.").

% Scoring function
#bias("penalty(Penalty, Action) :- in_head(Action), BodyLen = #count{B:in_body(B)},
       ContainCount = #count{Y:contains(Action,Y)}, gen_threshold(Gt),
       relu(Gt-ContainCount, ReLu), Penalty = (ReLu * 10)/BodyLen.").
```

Listing 7.6: The scoring function represented for FastLAS

The hypothesis received after running the FastLAS learning task with the background knowledge and the scoring function can be seen in Table 7.6. As expected, $Action_1$ and $Action_4$ — the recovery actions that belong to units — have more general rules, on the other hand, all the other actions have very specific rules.

Symptoms				**Action**
s_1	**s_2**	**s_3**	**s_4**	
F	-	ø	ø	1
ø	ø	T	-	4
T	-	F	F	2
T	-	-	-	3
F	T	-	-	3
F	T	F	F	3
-	-	-	T	5
F	T	-	-	5
T	-	-	-	5
T	-	T	F	6

Table 7.6: The hypothesis optimized based on the system hierarchy (The horizontal line depicts the threshold)

In Example 2 (7.3) the weights had to be manually assigned to the individual actions, while in this example they were extracted from the engineering models that accurately describe the system. Both approaches have advantages, manual weight assignment can be very useful in cases when there are no models and adequate background knowledge that could be used in the process. However, when there are models about the system, they can be easily represented and used. This approach is much more scalable and can easily adapt to system changes.

7.5 Use Case 4 - Consistency Checking

Modeling complex system accurately is a challenging task, since there are a lot of things that can go wrong and lead to erroneous results. This example illustrates how the input examples and the Engineering Models (Figure 3.3) can be used to check for any inconsistencies during the rule extraction process to avoid incorrect results. In the example, the possible symptoms that can appear in the body of a rule will be restricted, and the rule extraction process will check for any erroneous examples that violate this constraint, and reject them if they do so. If there will be no hypothesis that satisfies all the constraints, then it will mean that there is an inconsistency between the examples and the models. In addition to this, the example will also take into consideration that faults might propagate from one component to the other if they are connected, making use of the connections in Figure 7.1.

The first step is to define the restrictions for the rule bodies, which is going to be based on the initial fault recovery action table (Table 7.1). These constraints will be used to disqualify rules for actions which contain symptoms that do not appear together in a single row with the action in the table (Table 7.1). Listing 7.7 show these body constraints for $Action_3$. All restrictions can be defined in the same way for the other recovery actions too.

```
% Possible bodies for action_3
#bias("possible_body(action_3, s_1(t)).").
#bias("possible_body(action_3, s_1(f)).").
#bias("possible_body(action_3, s_2(dont_know)).").
#bias("possible_body(action_3, s_2(t)).").
#bias("possible_body(action_3, s_3(dont_know)).").
#bias("possible_body(action_3, s_3(f)).").
#bias("possible_body(action_3, s_4(dont_know)).").
#bias("possible_body(action_3, s_4(f)).").
```

Listing 7.7: $Action_3$ possible bodies based on the initial fault recovery action table

The next step is to represent the connections in the system from Figure 7.2 for the FastLAS task. This will be used to account for fault propagation inside the system. Faults that appear in one system element might cause errors in the following, connected elements. The advantages of ASP are going to be utilized again to simplify this step, by only defining the first-order connections between the elements and using ASP rules to calculate the transitive closure (4.3.1) of the connections.

Listing 7.8 depicts the connections and the background knowledge for the FastLAS task. The first block of code describes the first-order connections based on Figure 7.2. The second part is very important, at first it computes the transitive closure of the connections then uses the closure to define the additional possible symptoms for the rules. If there is a transitive connection between two system elements, then it means that there is a flow of information or dependency between the two, and an error in any preceding element can propagate to the elements that it communicates to.

```
% Background-knowledge representing action connections
#bias("connection(action_2, action_3).").
#bias("connection(action_2, action_5).").
#bias("connection(action_3, action_2).").
#bias("connection(action_3, action_5).").
#bias("connection(action_3, action_6).").
#bias("connection(action_5, action_6).").

% Background knowledge processing - closure
#bias("transitive(X,Y) :- connection(X,Y).").
#bias("transitive(X,Z) :- X!=Z, connection(X,Y), transitive(Y,Z).").
% Background-knowledge processing - fault propagation
#bias("possible_body(X,Z) :- transitive(Y,X), possible_body(Y,Z).").
```

Listing 7.8: Connections and background knowledge processing

The last step to introduce consistency checking is to define an integrity constraint that rejects any hypothesis that contains rules for which the head takes up a forbidden body. With the help of the constraint in Listing 7.9, every hypothesis is rejected which contains a head and body combination that is not included in the possible bodies represented in the background knowledge.

```
% Integrity constraint for consistency checking
#bias(":- in_head(A), in_body(B), not possible_body(A,B).").
```

Listing 7.9: Integrity constraint for consistency checking

Now that the integrity constraint has been defined, consistency checking has been introduced to the FastLAS task. To test it, an erroneous example will be added to the task and checked whether there is any hypothesis that can satisfy it. Consider the positive example for $Action_2$ in Listing 7.10. With this example, the output of the rule extraction process will be *UNSATISFIABLE*. This is because the rules for $Action_2$ are not allowed to contain the symptom $s_4(t)$, additionally, there are no connections to $Action_2$ (Figure 7.2) from where this symptom could propagate. In the closure, $Action_2$ is only connected to $Action_3$, but even $Action_3$ is not allowed to contain $s_4(t)$ in its body, so there can not be a hypothesis that satisfies this new positive example. To fix this error, either the new positive example has to be adjusted, or $s_4(t)$ has to be added to the possible bodies of $Action_2$ or $Action_3$.

```
#pos(id(9),{action_2},{action_1,action_3,action_4,action_5,action_6},{
    s_1(f).
    s_2(t).
    s_3(f).
    s_4(t).
}).
```

Listing 7.10: Erroneous example to showcase consistency checking

Consistency checking can be a very beneficial addition to any rule extraction task to mitigate mistakes. It is simple and straightforward to set up and can save a lot of time during the development process. However, it is important to point out that consistency checking does not tell what is wrong, the examples or the background knowledge. It only tells that they are not compliant with each other, and its the task of the developers and the system experts to resolve the contradiction between the two.

Chapter 8

Summary

8.1 Conclusion

The goal of the report was to examine the field of fault recovery and how inductive reasoning can be used to support the fault recovery process. The report thoroughly examined different aspects of the field of fault recovery. Starting from general concepts, it discussed related fields and areas, then moved on Knowledge-Based Fault Recovery. It defined a universal workflow for supporting fault recovery with inductive reasoning. Then it delved into the technologies that can be used to implement Knowledge-Based Fault Recovery, such as Answer Set Programming (ASP) and Inductive Logic Programming (ILP). In the end, it provided examples with the novel FastLAS ILP system to illustrate how the workflow works.

The use cases walked through the different parts of the workflow and showcased how they can be used, what can be included in them, and what their capabilities are. The features presented with the examples are among the most fundamental ones and provide a good starting point for many fault recovery tasks. Additionally, the workflow has been designed to be modular enough to allow the inclusion of further features to extend it.

The examples clearly showed that supporting fault recovery with Inductive Logic Programming systems has many benefits. The main benefits are the explainability of the extracted rules, the ability to generalize from a smaller number of examples and — thanks to FastLAS — the support of domain-specific scoring functions to optimize the hypotheses with respect to target metrics.

8.2 Further Work

There are many aspects in which the research of Knowledge-Based Fault Recovery could continue. One such area is the usage of scoring functions. In the report, simple scoring functions have been used to showcase their functionality. However, there are many sophisticated scoring functions that could contribute far more to the fault recovery process than the ones presented. Another possibility is the examination of the usage of additional system models. Even though the modeling aspects presented here can be used to cover a wide-range of modeling-related problems in Knowledge-Based Fault Recovery, there are numerous other system models (e.g., behavioral models) that can be made use of to fine-tune the process. On top of these two directions, it could also be examined how the workflow and the approach could be modified to support changing systems and requirements to provide a runtime fault recovery approach.

Acknowledgements

I would like to express my gratitude to my supervisor, András Földvári, for his continuous support throughout the last three semesters in which he has been supervising me. Most importantly, during the time which I was writing this report. I learned a lot from him in this period, many important concepts and approaches that I will carry with myself for the years to come.

Bibliography

[1] M. Law, A. Russo, E. Bertino, K. Broda, and J. Lobo, "FastLAS: Scalable Inductive Logic Programming Incorporating Domain-Specific Optimisation Criteria," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, no. 03, pp. 2877–2885, Apr. 2020. [Online]. Available: https://ojs.aaai.org/index.php/AAAI/article/view/5678

[2] S. Ding, *Model-based fault diagnosis techniques: Design schemes, algorithms, and tools*, 01 2008.

[3] R. Beard, "Failure accomodation in linear systems through self reorganization," 1971.

[4] Y. Chi, Y. Dong, Z. J. Wang, F. R. Yu, and V. C. M. Leung, "Knowledge-Based Fault Diagnosis in Industrial Internet of Things: A Survey," *IEEE Internet of Things Journal*, vol. 9, no. 15, pp. 12 886–12 900, 2022.

[5] Y. Wilhelm, P. Reimann, W. Gauchel, and B. Mitschang, "Overview on hybrid approaches to fault detection and diagnosis: Combining data-driven, physics-based and knowledge-based models," *Procedia CIRP*, vol. 99, pp. 278–283, 2021, 14th CIRP Conference on Intelligent Computation in Manufacturing Engineering, 15-17 July 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2212827121003152

[6] R. Arunthavanathan, F. Khan, S. Ahmed, and S. Imtiaz, "An analysis of process fault diagnosis methods from safety perspectives," *Computers Chemical Engineering*, vol. 145, p. 107197, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0098135420312400

[7] J. Hou and B. Xiao, "A Data-Driven Clustering Approach for Fault Diagnosis," *IEEE Access*, vol. 5, pp. 26 512–26 520, 2017.

[8] A. Lundgren and D. Jung, "Data-driven fault diagnosis analysis and open-set classification of time-series data," *Control Engineering Practice*, vol. 121, p. 105006, apr 2022. [Online]. Available: https://doi.org/10.1016%2Fj.conengprac.2021.105006

[9] S. Wang, C. Li, and A. Lim, "A Model for Non-Stationary Time Series and its Applications in Filtering and Anomaly Detection," *IEEE Transactions on Instrumentation and Measurement*, vol. 70, pp. 1–11, 2021. [Online]. Available: https://doi.org/10.1109%2Ftim.2021.3059321

[10] D.-T. Hoang and H.-J. Kang, "A survey on Deep Learning based bearing fault diagnosis," *Neurocomputing*, vol. 335, pp. 327–335, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0925231218312657

[11] S. Heo and J. H. Lee, "Fault detection and classification using artificial neural networks," *IFAC-PapersOnLine*, vol. 51, no. 18, pp. 470–475, 2018, 10th IFAC Symposium on Advanced Control of Chemical Processes ADCHEM 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2405896318320664

[12] D. Rebouças, F. Araújo, and A. Maitelli, "Use of Artificial Neural Networks to Fault Detection and Diagnosis," *ABCM Symposium Series in Mechatronics*, vol. Vol. 5, pp. 665 – 675, 01 2012.

[13] F. Wotawa, "On the use of answer set programming for model-based diagnosis," in *International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems*. Springer, 2020.

[14] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub, "Answer Set Solving in Practice," *Synthesis Lectures on Artificial Intelligence and Machine Learning*, vol. 6, pp. 1–238, 12 2012.

[15] G. Shea, "NASA Systems Engineering Handbook Revision 2," 2017. [Online]. Available: https://www.nasa.gov/connect/ebooks/nasa-systems-engineering-handbook

[16] K. D. Forbus, "Chapter 9 Qualitative Modeling," in *Handbook of Knowledge Representation*, ser. Foundations of Artificial Intelligence, F. van Harmelen, V. Lifschitz, and B. Porter, Eds. Elsevier, 2008, vol. 3, pp. 361–393. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S157465260703009X

[17] W. R. Simpson and J. W. Sheppard, *System Test and Diagnosis*. USA: Kluwer Academic Publishers, 1994.

[18] C. Anger, K. Konczak, T. Linke, and T. Schaub, "A Glimpse of Answer Set Programming," *KI*, vol. 19, pp. 12–, 01 2005.

[19] V. Lifschitz, "What is Answer Set Programming?" in *Proceedings of the 23rd National Conference on Artificial Intelligence - Volume 3*, ser. AAAI'08. AAAI Press, 2008, p. 1594–1597.

[20] G. Brewka, T. Eiter, and M. Truszczyński, "Answer Set Programming at a Glance," *Commun. ACM*, vol. 54, no. 12, p. 92–103, dec 2011. [Online]. Available: https://doi.org/10.1145/2043174.2043195

[21] T. Dell'Armi, W. Faber, G. Ielpa, N. Leone, and G. Pfeifer, "Aggregate Functions in Disjunctive Logic Programming: Semantics, Complexity, and Implementation in DLV," 01 2003, pp. 847–852.

[22] M. Law, "Inductive Learning of Answer Set Programs," pp. 70–73, 2018, https://www.doc.ic.ac.uk/~ml1909/Law-M-2018-PhD-Thesis.pdf.

[23] M. Law, A. Russo, and K. Broda, "The ILASP system for Inductive Learning of Answer Set Programs," 2020. [Online]. Available: https://arxiv.org/abs/2005.00904

[24] S. Muggleton, "Inverse Entailment and Progol," *New Gen. Comput.*, vol. 13, no. 3–4, p. 245–286, dec 1995. [Online]. Available: https://doi.org/10.1007/BF03037227

[25] A. Drozdov, M. Law, J. Lobo, A. Russo, and M. W. Don, "Online Symbolic Learning of Policies for Explainable Security," in *2021 Third IEEE International Conference on Trust, Privacy and Security in Intelligent Systems and Applications (TPS-ISA)*, 2021, pp. 269–278.

[26] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub, "Multi-shot ASP solving with clingo," *Theory and Practice of Logic Programming*, vol. 19, no. 1, p. 27–82, 2019.