Institute of Software Engineering

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Master Thesis

Explainability of Operating Systems

Tobias Huschle

Course of Study:	Informatik
Examiner:	Prof. Dr.-Ing. Steffen Becker
Supervisor:	Dr.-Ing. Kiana Busch

Commenced:	January 15th, 2021
Completed:	July 12th, 2021

Abstract

With the recent rise of machine learning and artificial intelligence, the explainability of software has found its way into the focus of research activities. Black box-like approaches that take critical decisions must be enabled to justify its actions in a comprehensible manner. This thesis takes these considerations and applies them to the area of operating systems and problem analysis thereof. To do so, a user study, conducted among professionals, is presented that shows that simplifying the generation of explanations of the operating system behavior can bring additional value. Furthermore, already available tools will be discussed based on their capabilities with regard to explanation generation. Subsequently, a new approach is proposed that allows to visualize decisions taken by the operating system in a decision graph. These graphs allow to examine how and why a certain value was set by the operating system in a convenient and efficient way. Finally, this approach is evaluated in another user study, which is again conducted among professionals. The final conclusion of this thesis then yields, that an increased focus on explainability capabilities in the context of operating system problem analysis would bring additional value to people working in this area. There is a wide range of other publications that focus on either problem analysis or explainable software, but not on the combination thereof. The proposed approach aims to connect the two areas by providing assistance in deriving explanations and justifications for the internal reasoning processes of operating systems in a convenient way. The potential value is successfully confirmed with an evaluation study conducted among professionals.

Kurzfassung

Mit den jüngsten Fortschritten im Bereich von Maschinellem Lernen und Künstlicher Intelligenz geht ein verstärktes Interesse in der Forschung einher inwiefern Entscheidungen, die von Software getroffen werden, erklärt werden können. Blackbox-artige Ansätze, die kritische Entscheidungen treffen, müssen in die Lage versetzt werden ihre getroffenen Aktionen in verständlicher Art und Weise verteidigen zu können. Diese Abschlussarbeit wendet diese Fragestellung auf den Bereich der Betriebssysteme an. Hierbei wird Fokus auf das Teilgebiet der Problemeanalyse gelegt. Um diese Themengebiete zusammenzubringen, wird zunächst eine Studie unter Experten aus der Industrie präsentiert, um zu zeigen, dass eine Vereinfachung in der Herleitung von Erklärungen für das Betriebssystemverhalten einen zusätzlichen Wert darstellen kann. Zusätzlich werden von den Experten tatsächlich genutzte Tools auf ihre Fähigkeiten bezüglich der Herleitung von Erklärungen untersucht, um zu zeigen, dass eine Erweiterung der Funktionalitäten in Bezug auf Erklärbarkeit möglich ist. Eine mögliche Herangehensweise auf Basis von Entscheidungsgraphen, die es erlaubt schnell und effizient abzulesen warum und wie Entscheidungen im Betriebssystem getroffen werden, wird im Anschluss präsentiert. Dieser Vorschlag wird zu guter Letzt in einer Evaluation verifiziert, die wieder im Rahmen einer Studie mit Experten aus der Industrie durchgeführt wurde. Diese Abschlussarbeit kommt abschließend zum Schluss, dass ein verstärkter Fokus auf einen erklärbaren Entscheidungsprozess im Bereich der Problemanalyse von Betriebssystemen zusätzlichen Wert generieren würde. Es existiert eine Fülle an anderen Publikationen, die sich mit den Themen Problemanalyse und Erklärbarkeit von Software beschäftigen. Die Kombination der beiden Felder wird dabei jedoch vernachlässigt. Der in dieser Arbeit vorgeschlagene Ansatz zielt darauf ab diese beiden Gebiete zusammenzuführen indem eine Möglichkeit aufgezeigt wird Erklärungen für den internen Entscheidungsprozess des Betriebssystem einfacher herzuleiten. Der potentielle Wert dieses Konzepts wird erfolgreich mit einer Studie unter Experten aus der Industrie bestätigt.

Contents

1 Introduction 6

2 Foundation 9
- 2.1 Operating Systems . . . 9
- 2.2 Performance and Problem Analysis . . . 14
- 2.3 Explainable Software . . . 16
- 2.4 User research . . . 18

3 Related Work 20
- 3.1 Problem Analysis Process . . . 20
- 3.2 Explainable Software . . . 21
- 3.3 Software Engineering Techniques . . . 26

4 Study of the Problem Analysis Process in the Industry 27
- 4.1 Study Goals . . . 27
- 4.2 Study Design . . . 28
- 4.3 Study Results . . . 31
- 4.4 Scenarios . . . 35
- 4.5 Summary . . . 37

5 Classification of Existing Tools 38
- 5.1 Classes . . . 38
- 5.2 Classification . . . 40
- 5.3 Summary . . . 46

6 Explainable Software Approaches 48
- 6.1 Decision Graphs . . . 48
- 6.2 Application of Approaches . . . 60

7 Evaluation 63
- 7.1 Study Goals . . . 63
- 7.2 Study Design . . . 63
- 7.3 Study Results . . . 65
- 7.4 Threats to Validity . . . 69
- 7.5 Summary . . . 69

8 Conclusions 70
- 8.1 Summary . . . 70
- 8.2 Outlook . . . 71

9 Appendix 73
9.1 Study of the Problem Analysis Process in the Industry 73
9.2 Tool Classification Scores 86
9.3 Evaluation 89

Bibliography 103

1 Introduction

Explainable software is a field of research that is gaining an increasing amount of traction in recent time. This correlates with the increasing usage of machine learning algorithms. While simple techniques like support vector machines or k-means clustering remain interpretable, more complex approaches lead to models that must be seen as a black box. Especially neural network models do not offer an obvious way to explain why they take certain decisions. To tackle this problem, there is a growing number of research that aims to provide ways to generate explanations for software that leverages complex machine learning algorithms.

The need for such explanations is obvious. Especially software that takes decisions, which have immediate impact on critical business choices or even human lives, has to be fully explainable to ensure the possibility to find the root-cause of a problematic scenario. Such critical software can be found in various areas. One prime example is the automotive sector and its progress towards autonomous driving. Decisions taken in real-life traffic may have serious impacts. In these cases it will always be necessary to be able to analyze the internal reasoning processes of all involved pieces of software.

Another type of software that takes critical decisions in very high frequencies are operating systems. Operating systems are a crucial part of almost all computer systems and are used to bridge the gap between application software and the hardware itself. This implies the handling of time-sharing concepts, virtual memory management or interaction with peripheral devices like hard disks or network cards. As a consequence, operating systems are very complex pieces of software which can easily be interpreted as black-box models due to their large scope and extensive code bases. At the same time, it remains necessary to have the chance to always reconstruct and understand the decisions taken by the operating system. A problem in the internal reasoning process of the operating system can have dire consequences. A workload which is denied access to computational resources for a longer period of time can delay important business processes or cause an outage. As all commonly known and used operating systems are fully deterministic and do not leverage black-box machine learning algorithms like neural networks, root-cause analysis remains primarily a matter of time and available log data.

This does not mean that deriving such explanations is trivial nor feasible in all cases. Certain scenarios might require a deeper analysis of the internal reasoning process which in turn might ask for a deep look into the source code, as not necessarily all decision steps are explicitly described in documentation. For open source operating systems like Linux, an analysis of the source code is always possible, but still requires additional knowledge and skills to be able to understand the underlying concepts. Proprietary operating systems like IBM z/OS do not offer access to their source code, therefore making it necessary to rely on the analysis capabilities of the company offering the product in certain scenarios.

In any case, a wide set of tooling is available to help during the analysis process without requiring to look into the source code. These tools rely on the accessibility of real-time or historic log data in order to provide insights into the internal dynamics of the operating system. Fortunately, such log data is usually available for operating systems or can at least be activated if necessary. While it is possible to generate explanations for a subset of decisions taken by the operating system, this cannot be done in all scenarios.

There are several approaches in the involved areas. As mentioned, deriving explanations is a growing field of research especially in the field of artificial intelligence [SWM17], however without focusing on software that relies on classic implementations like operating systems. Analysis methodologies for operating systems have been subject to research for decades and have been covered by a large set of authors [Jai90] [All08] [Gre13a]. However, they do not focus on the internal reasoning process and particular decisions but on higher abstraction levels. The generation and necessity of explanations based on traditionally implemented software was already discussed decades ago [Swa83]. However, these approaches focused on hard-coded scenarios and did not cover more dynamic cases as the operating system context. The format of explanations is also a well researched topic, as it is necessary to consider which form and level of detail is appropriate [CBF+05]. However, it was not possible to find publications with the focus on explanations for the decision process of operating systems. The adequacy of explanations needs to be evaluated via user studies as their usefulness and comprehensibility is of subjective nature. Strategies and metrics in that area are also widely researched [HBK+17] [HMKL18] [ABC+19]. However, existing publications do not combine the areas user research, explainable software and operating systems. This thesis will therefore address the following research goal:

> **RG**: Evaluate how the explainability of operating system decisions can be increased and how this improves the analysis process.

To do so, a set of research questions will be answered. The following paragraphs will provide more details on the research questions and briefly discuss the approaches that will be used to address them.

The first part of this thesis aims to gather insights into the general environment and proceedings of problem analysis in the operating system context. This will be achieved by conducting a user study among professionals who perform such analysis in their daily job. The study will gather insights into the time effort caused by problem analysis as well as identifying what causes are predominantly identified in conclusion. Furthermore, a collection of approaches and tools used in the process will be conducted. Finally, the study will lead up to query a set of scenarios encountered by the professionals that require the explanation of decisions taken by the operating system. Study design and results will be described in Chapter 4, which will then provide answers to the first research question:

> **Q1**: Which scenarios exist in the area of operating system that require explanation?

The second part of this thesis will address the fact, that there already exists a large set of tools and concepts for problem analysis in the operating system context. Therefore, Chapter 5 will briefly present and discuss the different concepts behind a subset of available tools. The tools will not be selected at random. Instead the previously mentioned user study presented in Chapter 4 will

explicitly ask the participants to name tools that they are using regularly, which will help to keep focus on industry relevant tools. The gathered tools and their capabilities will then be used to answer the second research question:

> **Q2**: Which approaches are used in the industry to explain the operating system decision process today?

The third part of this thesis aims to evaluate possible ways to enhance the explainability of the operating system reasoning process. The proposed approach will make use of the interpretative nature of decision trees and highlight multiple aspects that can be of use to generate explanations. To do so, Chapter 6 will introduce the concept of decision graphs and describe how the log data of the operating system can be extended and leveraged to create these graphs. The presented approaches will provide an answer to the third research question:

> **Q3**: Which approaches could be used to increase the explainability of operating system decisions?

The fourth part of this thesis will then aim to verify whether an increased focus on explanations of the internal reasoning process of the operating system provides actual value. To do so, the approaches described in Chapter 6 will be shown to professionals in a second user study. The participants are asked to provide feedback on these approaches. This feedback will be used to highlight the overall positive response of the interviewed professionals. They agreed that the presented approaches would bring advantages in their daily work while also giving them the capability to derive explanations of the operating system behavior more easily. While the professionals could not imagine to create complete explanations with the described approaches only, they agreed again, that the approaches would pose a useful addition to their existing set of tools. The complete set of gathered results will be shown in Chapter 7 to finally highlight their contributions to the evaluation goal:

> **EG**: Evaluate whether an increased explainability of the internal reasoning process would be a valid enhancement for operating systems.

In order to introduce the necessary foundation for this thesis, Chapter 2 will describe the current state of research in the areas of operating systems, problem analysis, explainable software and user research. Subsequently, Chapter 3 will provide an overview of related work and highlight how this thesis distinguishes itself from those publications. The contributions of this thesis will be shown in Chapter 4 through Chapter 7. Chapter 4 will present a user study that aims to get insights into the problem analysis process of operating systems in the industry. Chapter 5 will classify tools used by professionals during this process by discussing their capabilities including their support to generate explanations for the internal reasoning process of the operating system. Chapter 6 will propose a new approach to visualize the decision process of the operating system. Chapter 7 will then present a second user study which aims to evaluate whether the proposed approach would be of actual value for professionals. Finally, Chapter 8 will summarize all contributions and provide a brief outlook on further research opportunities.

2 Foundation

This chapter will cover the foundations of this thesis. First, an overview of the concepts and internals of *operating systems* will be given in Section 2.1 as well as a discussion on their complexity and the consequential challenge to derive explanations for their internal reasoning process. The subsequent Section 2.2 then highlights methodologies and strategies which originated from decades of research in *performance and error analysis* in the realm of operating systems. Afterwards, the computer science area of *explainable software* will be introduced in Section 2.3. Finally, techniques and activities used in *user research* will be shown together with their advantages and disadvantages in certain scenarios in Section 2.4. The focus will hereby lie on the applications of *user research* in the following parts of this thesis.

2.1 Operating Systems

Any kind of software needs to be run on some form of hardware. Apart from some special cases, a computer system usually has access to a Central Processing Unit (CPU), memory and external devices as shown in Figure 2.1. The CPU serves as the executing component that processes the machine instructions as defined by an application. Memory is fast storage used to retain intermediate results and data that is needed by the application. External devices can be long-term storage like hard drives or network devices that allow to persist data or to communicate with other systems [NL14].

These resources can be shared between multiple applications and services on a single machine. In simple scenarios it can be possible to process all involved tasks in a strictly serial way, i.e. one after another. But in the majority of cases it will be necessary to process all tasks in parallel. Deciding which task gets access to a certain resource at which time is a highly non-trivial task in such a parallelized scenario. Not only is it necessary to ensure a smooth swap between the different programs, it is also necessary to ensure that tasks have fair access to the resources to avoid certain

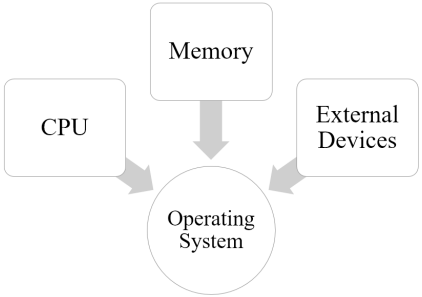

Figure 2.1: Basic environment of an operating system

workload not being process due to a lack of access to CPU for example. The interpretation of the term *fair* in this context does not necessarily mean that all task have the same access to the available resources. Some tasks will have a higher priority than others which is defined either by the person administrating the system or by the system itself [LBS99].

Balancing these requirements to make sure that the whole system runs smoothly and as expected is a complex problems. In order to avoid that people working on the application layer have to consider how their program might use and share the available resources, this task is taken care of by providing an abstraction layer for it. This layer is implemented by *operating system*, which pose the crucial link between applications and the underlying hardware. Operating systems ensure, that multiple programs can execute in parallel on shared resources while also providing interfaces to the hardware itself where necessary by taken constant adjustments and decisions on a very low level of the software stack. This allows applications and services to assume that they can use the available resources exclusively while actually sharing the underlying hardware in a fully transparent way [SPG91].

The complex nature of operating systems combined with their critical task to ensure that multiple applications can smoothly share the available hardware makes them an important backbone in almost all software related scenarios. This in turn means, that operating systems must function at all times. Problems and errors on the operating system level can have dire consequences and can be assumed to be beyond the scope of a standard application developer as the operating system is expected to be standard infrastructure that can be assumed to work in all cases. Fortunately, operating systems have the capability to provide extensive log information that can be used for analysis purposes. This is especially useful in commercial usage where an operating system caused outage might not only frustrate the user, but can also cost a significant amount of money. All these aspects combined make operating systems a great target to evaluate the problem analysis process, i.e. how the available logs can be turned into an explanation for what happened, and discuss how it can be potentially improved.

The following paragraphs will provide more details on the complexity of the operating system by discussing some of its task in order to highlight its choice for this thesis. Two main tasks of the operating system are process scheduling and memory management which will be shown in Section 2.1.2 and Section 2.1.3 respectively. But first, the complex appearance of operating systems will be elaborated in Section 2.1.1.

2.1.1 Pseudo-Self-Adaptive Capabilities of Operating Systems

A core strength of operating systems is their capability to handle a wide range of workload types by managing and distributing the available computational resources [TB15]. To do so, permanent self adjustment is necessary to adapt to changing situations and constellations in the environment. A common approach to tackle this issue are feedback loops [BSG+09]. These lead to an abstract definition in the MAPE-K loop which is shown in Figure 2.2. The individual letters are abbreviations for the different stages of the flow: Monitoring the environment, analyzing the current state and extracting necessary information, planning actions to fulfill predefined goals and executing, all while interacting with a knowledge base that consists of a representation about the system and its environment [KC03].

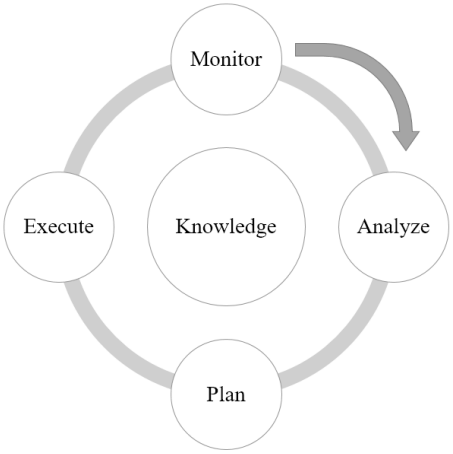

Figure 2.2: MAPE-K loop

One example of such a system is IBM's Workload Manager (WLM), which incorporated the described idea already in the 1990s to aim for better performance. The core idea is to permanently monitor system metrics that provide insight into the needs of each workload class. The administrator can define these classes along with priorities and goals that must be met. WLM then attempts to distribute the available computation resources like CPU in a way to ensure that the specified goals are met [RS10]. This idea also saw coverage in the Linux operating system in more recent research publications by Sironi et al. for example [SBC+12]. The described framework provides an more abstract approach to enhance operating systems in general with self-adaptive capabilities.

Operating systems are also software projects of highly complex nature. Balancing a set of goals while also being able to cope with changes in the environment is a non-trivial task. Specialized design and engineering approaches are a necessity to address these issues. Cheng et al. [CLG+09] and DeLemos et al. [DGM+13] provide an overview over recent approaches in that area. The described ideas especially implicate that the system behavior can change over time, i.e. given the exact same situation twice, the actions chosen by the system might be different. However, those ideas do not take into account how comprehensible the emerging reasoning processes are.

Today, such fully self-adaptive behavior is not implemented in operating systems which are in commercial use (e.g. Linux or IBM z/OS). Due to the vast range of possible scenarios, the reasoning process of these systems is just as hard to grasp as for fully self-adaptive systems though. This leads to the problem that it is hard to explain decisions taken by the system in the aftermath or even predicting them. Comprehending the reasoning process requires a deep understanding of the operating system. This poses especially a challenge if it becomes necessary to analyze the performance of workload on an operating system.

While modern operating systems behave deterministic in principal, slight changes in the environment can lead to different behavior in very similar situations. In these cases, operating systems can be subjectively seen as self-adapting system with non-deterministic behavior. Therefore, this thesis will focus on the operating system level and will discuss the current way to derive explanations and propose an approach that could simplify this process in that area.

2.1.2 Process Scheduling

A major concept within operating system are processes. A process is an abstract representation for any kind of component or application software. Each time a program is started, it is treated as a process. The goal of the operating system is to provide access to the available resources to each process when needed. One particular resource that will be covered in the following paragraphs is the time a process gets assigned on a CPU. This task is called *process scheduling* [JLT85].

There are different type of processes, for example:

- System background processes that provide administrative services.
- System foreground processes that allow the user to interface with the operating system.
- User processes that are started by the user by launching an application for example.

All these kinds of processes share the same set of hardware while executing. This means for example, that all processes take turns on the available CPU. It is obvious that for example a process that interfaces directly with the user must be treated with the highest priority to make sure, that the user does not observe any kind of latency when using the system. Other tasks, like automatic checks for updates, can be delayed in favor of other processes.

Each process can be in one of three states as shown in Figure 2.3. It can be currently executing on the CPU which is the *Running* state. It can be ready for execution and wait to be assigned CPU time which is the *Ready* state. Finally it can be waiting for an external event and is not ready to execute until this event occurred, which is the *Waiting* state. Such external events can be the signal of a completed request sent to an external device. A process can continuously be moved back and forth between the *Ready* and *Running* states. Once it is going into the *Blocked* state it cannot back to *Running* immediately but has to first pass through the *Ready* state [TB15].

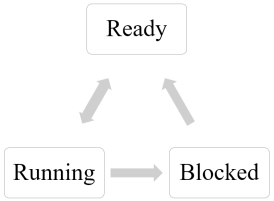

Figure 2.3: Scheduling states

A key decision that is taken by the operating system with a high frequency is to pick the next process which is in the *Ready* that will be granted CPU-time. After a certain amount of time, the currently running process is removed from the CPU and replaced with the selected follow-up process. The selection of the next process varies depending on the operating system in focus. Linux kernels use the *Complete Fair Scheduler (CFS)*[1] by default. The goal of this particular algorithm is to treat all processes in a fair way. This is achieved by granting the next available CPU time slot to the process which has received the least amount of execution time so far. In order to determine the next process, a red-black-tree is used. All processes are sorted into this tree in order of their execution time. The

[1] https://www.kernel.org/doc/html/latest/scheduler/sched-design-CFS.html

left-most node contains the process with the least amount of CPU time and can therefore be selected in constant time. After a process is finished with its time slice, it can be sorted back into the tree in $O(\log n)$ [CLRS09]. Different priorities of processes are accounted for by applying a decay factor on low-priority processes that influences the length of their time slices on the CPU. Finally it shall be noted, that currently running processes can propose another process to be run next on a CPU. In this case, the operating system needs to decide whether the proposed process can be ran without violating the fairness paradigm of the scheduler.

Nevertheless, this task remains non-trivial and can influence the behavior of the system in a critical way. Analysis of the scheduling behavior can be necessary to fully understand why a system is not performing as expected.

2.1.3 Memory Management

Another critical task performed by the operating system is memory management. As described in the previous paragraphs, multiple processes run within an operating system at once and share the available hardware resources. While the CPU can be shared by all active process by taking turns one after another, this is not feasible for memory. A single process can easily reach memory consumption of several gigabytes. Swapping the memory for each switch on the CPU initiated by the scheduler would cause an immense overhead and delay the execution times massively. In order to avoid such swaps, it is necessary to organize memory access in a clever way. The physical memory needs to be sliced in a way that all running processes can make use of it while maintaining a fully transparent view onto it. To do so modern operating system assign *address spaces* to all processes. These address spaces provide each process with its own virtual version of the available memory. This means especially that each process can make use of all possible addresses without having to deal with the potential use by other processes. This abstraction is a main task of the operating system [BC05].

By making use of address spaces, each process gets its own virtual copy of memory. This means that all processes combined work with an amount of memory that exceeds the physical resources available by huge margins. In order to keep the system running, the operating system needs to take decisions on which parts of the virtual memory of each process must be available in physical memory and which parts can be swapped out to slower persistent storage. To ensure good performance, a process that is about to be scheduled next, needs to have the memory it is using to be available on the physical layer. Otherwise the loading time from the persistent storage would significantly slow down the execution speed [TB15].

Therefore, each address spaces is organized in *pages*. Each page maps to a certain piece of virtual memory for a certain process. It is the decision of the operating system to select the pages to be held in physical memory and how to organize them. As shown in Figure 2.4 for two address spaces blue and green, the pages of an address space are not necessarily grouped together in real memory. Instead, it is well possible, that the operating system decides to split the pages or leave certain areas of the physical or real memory unallocated. For the address space itself though, the pages are accessible as if they would be one continuous piece of memory. If a process requests to access a page that is not available at the moment, a *page fault* occurs and the page needs to be costly loaded. To avoid such page faults, the operating system needs to decide which pages can be safely swapped out [CO72]. There are many strategies that can be used to achieve this goal. One example is the

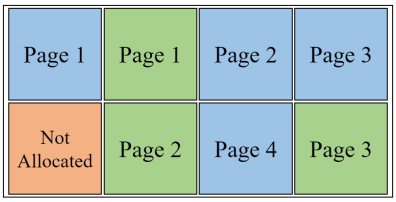

Figure 2.4: Pages in real memory

Least Recently Used (LRU) algorithm which always selects the page which has not been used the longest among all currently available pages. A variation of this algorithm is also used in the Linux kernel[2].

In general, this decision is non-trivial. Even if a page has not been used for a long time, there is no guarantee that it will not be used in the near future as well. In the case of an unexpected amount of page faults, the performance of a system can degrade severely. In this scenarios it is necessary to perform an analysis of the paging behavior to fully understand why the system is not performing as expected.

2.2 Performance and Problem Analysis

As described in the previous section, operating systems perform critical tasks that are of a non-trivial nature. In the case that a component of an operating system fails or does not perform as expected, dire consequences can arise. Performance degradation or outages caused by such failures require analysis that can only be done with a specific set of skills. Users, as well as IT professionals without the necessary background, do not have a deep knowledge of the concepts and technical details that can be found within the kernel of a modern operating system. The following section will therefore provide a short overview on performance and problem analysis with focus on involved roles, metrics and methodology.

Performance and problem analysis can be performed by multiple job roles in the IT area. System administrators for example need to identify problems and performance issues on the systems they are maintaining. It is their job to ensure that the system runs smoothly and can be adapted to changing workloads. Operating system developers have to be able to debug and analyze the behavior of the system they are working on. For them, it is particularly valuable to understand potential bugs and architectural problems and their influence on performance. Support staff works on the front-line with the customer. They have to find ways to explain to the customer why the system does not perform as expected. All these roles cover problem analysis on a part time basis while also handling user requests or developing new operating system features. Additionally, there are job roles, performance engineers for example, that solemnly focus on problem analysis [Gre13a]. Their common ground is the need to be able to understand the internals of the operating system. Some cases might even require to gather explanations for the internal reasoning of the operating system. Validating this point and proposing a new approach to tackle this issue will be covered by the following chapters of this thesis.

[2] https://www.kernel.org/doc/gorman/html/understand/understand013.html

For the purpose of performance and problem analysis it is possible to evaluate a wide range of metrics. The most common ones are the following [Jai90]:

- **Response Time:** Maximum or average duration to process a request.
- **Throughput:** Measurement on how much workload can be processed in a certain time-frame, e.g. number of transactions.
- **Resource Utilization:** Capacity of a resource used by the system. A fully utilized resource can be an indicator that the system is overloaded with work.
- **Reliability:** Measurement on how long the system is up and running without any errors of a certain severity.
- **Availability:** Measurement on how long the system can be ran without having to stop it for maintenance or due to problems.

It shall be noted that some operating systems explicitly work towards optimizing their performance in relation to the mentioned metrics. IBM z/OS for example claims to focus on being especially reliable and available, i.e. running without errors and downtime[3]. At the same time, the sub component WLM allows to explicitly define response time and throughput[4] goals for each type of workload which also maps to the metrics mentioned above [RS10].

Performance and problem analysis methodologies exist in a variety of flavors and aim to provide structured proceedings to approach problematic scenarios. One example is the USE method which proposes to focus on three possible indicators of potentially critical situations:

- **Utilization** of the available resources. Over- or under-consumption of certain resources can provide important clues towards the cause of the problem.
- **Saturation** of wait queues of a system. Queues that are continuously filled with a large amount of requests for example, can provide indication of an unbalanced system.
- **Errors** that are reported by the system, e.g. repeating network package loss.

The analysis needs to be preceded by the development of a problem statement. This includes a detailed description of the problem, an assertion of recent changes made to the system and the collection of relevant debug and log data [Gre13b].

In contrast to problem analysis, performance analysis can be extended with a proactive component in addition to post-mortem analysis scenarios. This process is often referred to as *capacity planning*. The basic idea of this approach is to generate a forecast on future workloads and system utilization in order to address upcoming shortages before they actually occur along with potentially severe consequences. To generate such predictions Allspaw identified the three following steps. First, a definition and selection of appropriate metrics need to be done that can be used to generate a model of the system and its workload. Second, the constraints of the resources of the system need to be applied. Finally, the combination of metrics and constraints can be fed into an analytic model that allows prediction on potential resource shortages given its input [All08].

[3] https://www.ibm.com/docs/en/zos-basic-skills?topic=it-mainframe-strengths-reliability-availability-serviceability
[4] velocity in WLM terminology

The distinctive process step of problem analysis in comparison to performance analysis lies in the fact, that it is often necessary to evaluate the source code of the operating system. This becomes inevitable in order to understand the internal reasoning process and identify the area that causes the problem. Zeller describes a generalized seven step approach that represents an average problem fix. The first steps involve the collection of information in the form of problem descriptions and log data. Second, it is necessary to reproduce the error to verify and better understand the issue. Subsequently, test cases should be created that make sure the problem is easily to reproduce in the future while also allowing to let a test tool catch the problem if it would occur again. The following step focuses on code analysis and aims to thereby identify the input sources that contribute to the cause of the problem. After these sources are identified, those which are most likely to cause the problem need to be prioritized and further analyzed which yields the penultimate step to find the root cause. The seventh and final step is the actual fix of the problem within the source code [Zel09].

In general, all the described scenarios and approaches require a detailed understanding of the internals of the operating system in order to generate explanations that justify its behavior.

2.3 Explainable Software

Explainable software is a area of research in the IT field that strives to make decision processes of software more transparent by proposing ways to generate explanations for the behavior of software. This transparency has seen a rise in demand with the growing awareness for data privacy. People who are affected by decisions taken within a business process or a state authority for example must be able to obtain justifications for these decisions. This is necessary to ensure that these decisions are not influenced by unfair or discriminatory bias. To enforce this transparency, laws have been passed or are in the making. A prime example is the General Data Protection Regulation (GDPR) [18] that was implemented in the European Union in 2018. Especially paragraph 71 focuses on ”the right to obtain an explanation of the decision” where the decision ”is based solely on automated processing and produces legal effects concerning him or her or similarly significantly affects him or her” [18].

The idea of infusing such abilities, the explainability of internal reasoning processes, into computer driven systems is discussed in a wide range of areas, all under the cover of the term *Explainable Software*. Internet of Things devices, like smart thermostats or washing machines, are great yet simple examples why this is important. A thermostat that adjust the temperature in a non-predictable manner [YN13], will unlikely see huge sales numbers. Smart washing machines that can adapt to the schedule of the user can be helpful or confusing depending on their behavior [BVK+14]. These simple use case highlight how important explainability can be on a very simple level already.

In more complex areas, explainability transforms from just being a selling point to becoming a hard requirement. In critical environments the need for decision justification can even go beyond the data privacy as covered by GDPR. Systems that provide suggestions for healthcare treatments [HBPK17] [WRK+17] or contribute to the driving of an autonomous car [CRP18] [Coc18] are good examples for such areas. The decisions taken in these scenarios can have dire consequences which can lead to casualties in the worst case. Therefore, the involved reasoning processes need to be explainable to allow root cause analysis and be able to prevent such failures in the future.

A crucial distinction that needs to be considered when generating explanations for a system is the entity that receives them. The audience for these explanations can reach from end-users who usually cannot and do not want to know about the internal reasoning process of a system. An explanation that features too many details or is highly complicated might frustrate or confuse an end-user. Developers at the same time might be interested in precisely such explanations in order to be enabled to properly debug a scenario. This comes with a great challenge during the design of an explainable piece of software as it becomes necessary to consider the level of detail, terminology and amount of information for different scenarios [MHS17].

Furthermore, the nature of the underlying system needs to be taken into account. Systems that are working on a static decision tree can describe their decision process very conveniently while also being comprehensible. More complex approaches that might for example include machine learning algorithms are usually less straightforward. A well researched tool for clustering task are support vector machines (SVMs). By creating a hyperplane that separates binary classes, in the simplest case, it becomes possible to categorize entries of a data set. The generated decision boundary can often not be described in an intuitive way which is why research is done to tackle this issue. One approach is to create clusters on each side of the hyperplane which are then used to derive rules based on common properties. By evaluating these rules it becomes possible to reason why an element is located onto a certain side of the hyperplane [NAC02].

Another machine learning approach are Bayesian Networks. These offer a way to describe dependencies and causalities within in a model. Inputs (findings or evidence) are connected to output probabilities for a certain even via a network of unobserved variables and the involved joint probabilities. There are multiple aspects that need explanations in these networks. For example, it is necessary to justify how the observed evidence generates a certain probability of an event, e.g. why does a set of symptoms lead to a diagnosis of a certain disease. A set of approaches to explain Bayesian Networks have been described by Lacave and Diez [LD02].

As for Bayesian Networks and Support Vector Machines, there are more machine learning approaches that are research for their capabilities to provide insight into their reasoning process. One example that proves to be highly non-trivial are deep learning approaches. These algorithm provide models that are usually of black-box nature. Generating explanations for such model continues to be a topic of research [SWM17].

The challenge to derive explanations is nevertheless not limited to machine learning concepts. Even deterministic software can pose challenges in the process to justify its behavior. Large and complex pieces of software can take decisions in a multitude of scenarios. The involved reasoning processes can be the results of decades of work by generations of developers. Especially software on lower levels of the stack are often complete black-boxes for the application developer and the end-user. If a problematic behavior has its root cause within such a low level black-box, an extensive analysis will be necessary. To do so, a time-consuming analysis of source code and log files might be necessary in order to be able to provide an explanation to the behavior of the system. As shown earlier, operating systems fall into this category perfectly. They are complex and large pieces of software whose internals are not clear to all its users. This thesis will therefore aim to bring these two areas, operating systems and explainable software, together in order to evaluate whether this would provide advantages during error and performance analysis.

2.4 User research

A project that is supposed to be rolled out to a wider audience or has impact on business critical processes needs to be planned carefully. Before the actual development, it is necessary to determine the circumstances under which the software will be used. This process is called *software engineering* and is a core research area of computer science [LFW15].

A key element of this process is the definition and identification of the target user group which is intended to use the program. This process is called *user research*, again a broad area within software engineering [CB05]. There are a multitude of approaches to choose from:

Personas — Define one or multiple fictional person(s) that represent(s) the common groups of users that will make use of the software to be written. These personas attempt to give an insight into the daily work and needs of the people affected while also allowing a certain degree of generalization. Following design decisions are verified against these virtual users to ensure that all requirements are met as intended [Coo+04].

Surveys — To capture the ideas, needs and opinions of a wide range of potential users, it is of great aid to evaluate the design progress by executing surveys among potential users. This is especially useful if the audience to be surveyed is co-located and personal interviews would pose a significant cost factor [Fow13].

Prototyping — Design flaws must be identified as early in the engineering process of a software project as possible. An issue which is found at a late stage comes with complex and expensive refactoring tasks that can endanger the timeline of the project. To avoid such scenarios, it is helpful to continuously work on prototypes that mimic the behavior of the project at early stages without having all functionality fully developed. Where necessary, mock-ups can be added for demonstration purposes [KLSZ92].

The list above is of course only a short excerpt of all available strategies. Unfortunately, there is no strategy that provides a one-fits-all solution. Each project requires careful planning on which approach, or a combination of multiple, is suited for the given scenario.

This thesis will rely on the usage of interviews in order to provide qualitative insights. Interviews are a commonly used way to handle more complex topics as it allows for a more detailed explanation of the topic than surveys for example do while also allowing to extract a deeper understanding of the interviewees comments through a conversation. These conversations can be held in multiple ways. They can be unstructured, where the actual interaction between interviewer and interviewees resolves mostly in a guided conversation. A semi-structured interview consists of a pre-defined set of potentially open-ended questions. Finally, a structured interview consists of very narrowly designed questions and tend to be very similar to the approach of surveys in their aim for quantitative data [DC06]. This thesis will use semi-structured interviews as this strategy relieves the interviewer partially from steering the conversation, instead, it is possible to follow the pre-defined set of questions.

The flow of an interview is a widely discussed topic. One possible way to conduct an interview is to split it into five phases. First an introduction is done to explain the overall topic and the goal of the interview. Then a set of easy questions is posed in order to introduce the interviewees into the

topic and encourage their involvement. Subsequently, the detailed questions are asked where the key results of the interview are supposed to be collected. The fourth phase represents a summary of the gathered answers and the chance to ask for final thoughts of the interviewee. The final phase is a wrap up of the interview [CB05]. This thesis aims to follow this structure for the conducted interviews.

Conducting an interview is a non-trivial task. In order to gather high quality results an experienced interviewer is necessary. There are a multitude of potential pitfalls that can come up during an interview. One of the most common ones is the introduction of bias. Bias can manifest in multiple ways. Leading questions can steer the conversation into a direction that the interviewer wants to get to, but does actually deviate from the actual perspective of the interviewee. Another example of bias is the obvious or even subtle reaction of the interviewer towards the statements of the interviewee. Signs of agreement, like nodding or vocal approval for example, can motivate the interviewees to talk in a way that resonates with the interviewer instead of talking about their actual thoughts [CB05].

2.4.1 Evaluation Metrics

Measuring characteristics or capabilities of a system is a non-trivial goal. On the one hand, it is necessary to consider which metrics can be used to measure the system [BF08]. On the other hand, metrics in the sense of numeric values might not be sufficient, especially if a social component is involved [BR08]. This holds true for measuring the explainability of a system for example because the perception of explanations is very subjective. Herbst et al. proposed metrics to assess the performance itself of systems that are constantly adapting to the current situation, while hinting that it would be necessary to also investigate the evaluation of explainability [HBK+17]. Another approach aims at the explanation of the underlying models of Artificial Intelligence (AI) systems by identifying the features that have the most influence on a particular decision and by that, yielding a different level of explanation [ABC+19]. This leads to the conclusion that evaluating the explainability of a complex piece of software, for example operating systems, is not necessarily straightforward and requires a thought out study to capture results that allow to draw appropriate conclusions.

3 Related Work

The following chapter presents publications that are related to this thesis and highlights how this thesis differentiates itself from those. As this thesis combines multiple ideas, it is necessary to note, that these influences come from multiple areas. Certain aspects have been adopted while others where tweaked and adjusted to meet the requirements to contribute to the research goal.

3.1 Problem Analysis Process

The core assumption of this thesis is the non-triviality of problem and performance analysis. There are multiple publications that show the amount of influences that must be taken into account. At the same time there are different perspectives that can be taken during analysis. The following paragraphs will cover publications within this context.

Zeller [Zel09] provides an extensive insight into the problem analysis process with focus on identifying and fixing bugs. The core principle is the TRAFFIC model, shown in Figure 3.1, which describes the basic steps necessary to properly solve a bug. The focus of TRAFFIC relies on the analysis of source code and the execution of test cases and scenarios. Detailed description on the mentioned steps are provided and can be used as a guide line for the classic bug fixing and analysis process. This thesis however approaches the analysis process from another perspective. In contrast to Zeller, it is assumed that it is not necessarily possible to recreate the problematic scenario and apply tools like debuggers to step through the code in order to understand and identify the root causes. Instead, this thesis aims to propose a way to derive this understanding solely with the log output of a system. Additionally, this thesis puts focus on operating systems while Zeller offers a more generic approach for arbitrary software. This distinction results from the fact, that most applications do not provide log files with enough detail, if any, that would be sufficient to follow the internal reasoning process of the software.

Figure 3.1: TRAFFIC model

Gregg [Gre13a] takes another perspective which is mostly driven from the scope of performance analysis. The analyzing process does not focus on the alignment of actions and source code, but shifts to identifying and interpreting patterns or anomalies in the log data of operating systems for example. The target type of software does hereby map with this thesis by specifically addressing operating systems. In another publication, Gregg [Gre13b] defines the USE method which states that each resource needs to be checked for utilization, saturation and errors. This approach provides

guidance for the analysis process when system metrics are available through log files or sampling tools. In general, the remarks made by Gregg form a guideline on how to approach operating system performance problems. The described approaches are focused on the analysis of system performance metrics and aim to find problems on a higher level. This thesis however aims to provide insights into the internal reasoning process of the operating system by evaluating entries in the log files. Hereby the scope is on a slightly lower level as the approaches by Gregg. This thesis in fact, aims to propose a new approach and strategy that lies in between the source code level debugging and the more high level performance analysis.

Another important aspect of problem analysis is the time required to provide an explanation for the problem and ideally a fix. Boehm et al. [BAC00] discuss several classes of models that can be used to estimate software development cost. The focus in these approaches lies on the ex-ante estimation of a project. This thesis however concentrates on the maintenance time frame after the successful shipment of the software.

Weiss et al. [WPZZ07] describe a way to predict this time effort by applying the k-nearest neighbor algorithm to an existing error database. The derived model can than be used to estimate the time complexity of problems that have not been fixed yet. The model bases its prediction on the error description and title and performs reasonably well. This thesis however aims to gain an empirical insight into the experiences of professionals with regard to the time consumption of problem analysis. Hereby, the focus lies on the ratio between analysis and other workload and the time frame consumed by such an analysis. This takes into account, that the actual analysis is not the only time effort that requires attention, but also other delays like waiting for customer or expert feedback.

Zhang et al. [ZKZH12] discuss potential influences on the overall analysis and fix time cost. However, they focus on aspects that revolve around the bug description itself, e.g. its severity or its description length. While they take into account the effect of the operating system on which the bug was reported, they do not cover delays caused by other external reasons as described previously. In general, publications in the area of bug fix time estimation or evaluation do not focus on the analysis of problems in the internal reasoning process of operating systems. This thesis however, does exactly that and aims to shed light on the perspective on analysis in the field of operating systems.

Overall, there is a large set of research that focuses on the problem analysis process of software and operating systems in general. However, the capabilities to generate explanations for the operating system behavior and simplifications thereof are not covered by the publications discussed above. This thesis aims to bridge this gap by providing a connection between the two fields operating systems and explainable software.

3.2 Explainable Software

This thesis aims to shed light on the problem and performance analysis from the perspective of Explainable Software. The following paragraphs will present publications that are related to the approach proposed by this thesis.

Adadi and Berrada [AB18] provide an extensive literature research on explainable artificial intelligence (XAI). This broad overview highlights two aspects. One, explainability is necessary in a multitude of areas like transportation, healthcare, legal, finance and military. This need is shown by

listing a set of studies and other publications from the respective areas. Second, the motivation for explainability is presented by highlighting four scenarios where explanations are particularly relevant and useful. Being able to **justify** decisions is necessary to have auditable software and comply with legislation like GDPR. Explainable software also allows to increase the **control** over software by allowing better insights into potential flaws and problems. Furthermore, good explanations allow to continuously **improve** the software by being able to better understand its current behavior. Finally, explanations allow to potentially **discover** new approaches in the reasoning process while also being able to comprehend why a particular unusual action was chosen. One popular example is the success of AlphaGo Zero [SSS+17] which is able to beat the best Go players in the world while also continuously surprising with unexpected moves which turn out to be crucial for its success. In summary, Adadi and Berrada offer a great overview over the existing approaches in the area of XAI while also confirming the intention of this thesis, which is to show that increased explainability brings additional value to the operating system problem and performance analysis process. However, Adadi and Berrada focus explicitly on XAI, while this thesis has a particularly different scope in the area of operating systems. Furthermore, the approach presented in this thesis differs from the XAI approaches, which is not surprising, as artificial intelligence model have drastically different concepts in comparison to classic software like operating systems.

Samek et al. [SWM17] stress the necessity of explainability for black-box like software which can often be found in approaches out of the field of artificial intelligence. This is fully aligned with the concept of XAI mentioned in the previous paragraph where the necessity to be able to justify decisions made by software is a major requirement. Samek et al. show that there are approaches that allow the visualization of explanations for decisions taken by artificial intelligence model and highlight the advantages of such capabilities, e.g. detecting problems or new insights. The concept of using visualizations to improve explainability is also picked up by this thesis. However, it is applied on a different scope, namely problem and performance analysis, and in a different area, namely operating systems.

The described necessity for explainability can be found in multiple areas. Cysneiros et al. [CRP18] describe explainability as a fundamental requirement for the field of autonomous driving. While the majority of people is willing to let artificial intelligence support them in scheduling appointments or maintaining functions of their homes, being driven around by a machine might still cause discomfort. It will be required to build software that can explain the decisions they take while navigating through traffic to build the necessary trust to raise the acceptance of autonomous driving. This requirement is also present in the area of operating system. Anyone, from end-user to the application programmer or system administrator must not be bothered with worrying about the internals of the operating system. To achieve this, it is necessary to be able to explain and justify the operating system behavior at all time. This thesis incorporates the same premise in that it is necessary to be able to build and maintain trust in the system. However, it focuses on a different area, namely operating systems.

Holzinger et al. [HBPK17] highlight that the medical domain increasingly uses artificial intelligence approaches to assist medical professionals in the diagnosis process. The usual black-box nature of such algorithms is not acceptable in this area. A decision must be verifiable to avoid wrong treatment suggestions. From a legal perspective it is a hard requirement to be able to analyze the complete diagnostic process to allow the discovery of the root cause in the case of an error, which might have had fatal consequences in the worst case. Additionally, privacy concerns arise during the selection process of necessary data. Patients have the right to know why certain information about them are being collected and which role they play in the reasoning process of the used software.

This thesis incorporates this requirement, as it is compulsory to be able to explain decisions taken by the system to a customer/patient and which input values influence the decision process. However, it focuses on a different area, namely operating systems.

The approach proposed by this thesis in Chapter 6 is based on the assumption that decision trees and graphs are a visualization that is considered to be particularly comprehensible and easy to interpret. A simplistic decision tree is shown in Figure 3.2 which immediately reveals that state A can transition into state B if variable a is greater or equal than 100. Otherwise, it will transition into state C.

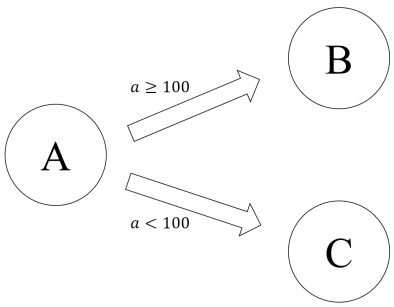

Figure 3.2: A sample decision tree

Cheverest et al. [CBF+05] for example present a user feedback survey that confirms this assumption by analyzing a smart office environment. By learning behaviors of the workforce, it is possible to take certain actions pro-actively (e.g. opening windows, activating fans) in an automated manner. These actions do not have to be taken by the workers themselves any more, which gives them more time for their actual tasks. The reasoning process is kept transparent and comprehensible by either visualizing the underlying decision trees, or using them for an easy generation of textual justification. Such trees can generate a textual representation of its decision process like ”The fan was started because the humidity was high”. The comprehensibility of the reasoning process is highly appreciated by the users as it gives them the confidence to build trust into the system. This shows that decision trees are received particularly well even by those who do not necessarily have an IT background. Therefore, this thesis presents an approach that also relies on this advantage of decision trees. However, the transfer to the problem and performance analysis process of operating systems done in this thesis shifts the scope significantly.

Another example for the advantages of decision trees in their capability to visualize reasoning processes is shown by Wan et al. [WDH+20]. The idea of neural-backed decision trees connects the two paradigms of black-box like but highly accurate machine learning models that are produced by neural network algorithms and the interpretability of decision trees. By replacing the final layer of a neural network with a decision tree it is possible to gather insights into the overall reasoning process of the neural network. This decision tree works with probabilities instead of binary branches which is not surprising due to the probabilistic nature of most machine learning algorithms. Nevertheless, the ability to gain insights into the decision process was very well received in three studies. This shows again the interpretability of decision trees. This thesis however does not aim to make neural networks more explainable but focuses on operating systems.

Similar to the approach proposed in this thesis, Swartout [Swa83] presents the XPLAIN framework that aims to connect the source code of a program in the medical diagnosis context with explanations and justifications for its behavior. The framework aims to capture the underlying reasoning process of the programmers and experts involved in the development of a software which has the purpose to provide diagnosis in the medical sector. By allowing the definition of relationships and hierarchies between pieces of knowledge a so-called *automatic programmer* is enabled to generate the necessary code which yields the desired software, including the explainability capabilities. Swartout also highlights, that hard-coded explanations in natural language, or canned explanations, are error prone and potentially incomplete. This comes from the fact that it is almost impossible to predict all possible questions towards the program, especially in more complex environments. At the same time, further enhancements might require adaptions on multiple locations, which can be easily missed. Another way shown to derive explanations is a source code analysis approach, which allows to infer the flow and behavior of the program. Unfortunately this poses strict requirements on the source code and its formatting in order to be able to draw insights from it in an automated way. The approach proposed by this thesis is obviously similar to the one presented by Swartout. The XPLAIN framework requires a drastically different development process in comparison to those in place in classical operating system development. This thesis therefore focuses on a way to gather explanations from the log files of the operating system in combination with slight adjustments to the source code. These adjustments obviously intertwine with the comments of Swartout with regard to canned explanations and explanation generation by source code analysis. The problem of hard-coded explanations is avoided, although not explicitly ruled out, by giving the analyst the chance to connect a chain or combination of decisions and thereby creating a better overview over the larger context. At the same time, the problem of relying on a certain source code formatting is relieved by proposing a set of code annotations that are used to generate the necessary data points in the operating system log files. In summary, the conceptual idea behing the XPLAIN framework contributes to the approach presented in this thesis. However, there are clear points of distinction as shown above.

Core et al. [CLV+06] aim for a similar goal as the XPLAIN framework described above. In distinction to XPLAIN, they focus on a way to provide a generic approach to build explainable artificial intelligence systems. Nevertheless, the underlying architecture contains similarities like a database that contains pieces of knowledge that are brought together and put in relation to one another. The approach to work with stored knowledge and generate explanations based upon it is the main differentiation point in comparison to this thesis. As shown, there is research being done in that particular area. This thesis however focuses on a way to make the generation of explanations easier while not requiring a complete re-design of the operating system.

Horvitz et al. [HBH88] bridge the gap between decision theory for expert systems and AI. They show that it is in general possible to represent decision processes of software in a comprehensible way. Again, the visualization method of choice are decision trees and graphs. This thesis picks up on the fact that the operating system reasoning can be transformed into an interpretable representation. This option to do so is shown in general. However this thesis applies it explictily to operating systems while also proposing a way on how to derive the desired decision graphs from log files. It shall be mentioned, that Horvitz et al. also mention the difficulties in generating alternate reasoning sequences based on an observed chain of decisions. This thesis also proposes a way to evaluate alternate decisions. However, it first focuses on more local alternatives in Section 6.1.7 to avoid the

implications of finding correct sequences. The more elaborate approaches proposed in Section 6.1.8 and Section 6.1.9 have to cope with this problem, whereas a potential technical implementation remains out of scope for this thesis.

Miller et al. [MHS17] raise the concern that the developers of explainable software might not be the most suitable people to design the explanation interaction with the user. This concern comes from the fact, that developers usually create solutions that fit their own needs when it comes to understanding critical parts of the system. The so created level of detail might not be appropriate for the end users of the system and might potentially even confuse them. They reason that it is necessary to involve people who are actually working with the software to be explained in the interaction design process. This thesis picks up on that idea by gathering the environment in which professionals are doing problem and performance analysis in the operating system context, as shown in Chapter 4. The same goes for the evaluation of the newly proposed approach, again working with professionals, as shown in Chapter 7.

Herbst et al [HBK+17] propose a set of metrics to evaluate the quality of the reasoning process of a piece of software that is able to take decisions. Obviously the ratio of correct decisions is a critical indicator on how well such a system performs. Also the amount of possible outcomes taken into considerations can provide good insights as it gives an intuition on the complexity of scenarios the system is able to handle. At the same time, it needs to be evaluated how valid these alternative considerations are. If there is only one reasonable decision among the alternatives, while all others are not at all relevant, the reasoning process might waste time in evaluating too many irrelevant options. Finally, Herbst et al. propose either the availability or even the quality of explanations for the reasoning process as a key metric to assess the quality thereof. However, they do not provide ways on how to derive these explanations but just call for their necessity. This thesis focuses on this particular aspect. Namely, how the generation of explanations in a complex reasoning process like the one within operating systems can be improved.

Hoffman et al. [HMKL18] discuss techniques on how to evaluate whether a system, in this case an explainable artificial intelligence system, provides useful explanations for its reasoning process. At first, it needs to be established what kind of explanation needs to be generated. As Hoffman et al. show, it is possible to take multiple perspectives and different levels of abstraction that require explanation. Furthermore they present a checklist that covers certain properties of explainability. Examples for these properties are the completeness of the derived explanations or whether they can increase the trust in the system. This thesis makes use of the aspects discussed by Hoffman et al. to derive an evaluation study that verifies the approach proposed in Chapter 6. However, it was necessary to take certain adaptations and specializations to create a mapping to the area of operating systems and ensuring that the evaluation fits into the time frame of this thesis.

Overall, there is a large set of research that focuses on the explainability of software in general. However, the capabilities to generate explanations for the operating system behavior and simplifications and evaluations thereof are not covered by the publications discussed above. In combination with the related work presented in Section 3.1, it becomes obvious, that the combination between the two fields operating systems and explainable software has not been covered so far. This thesis aims to bridge this particular gap.

3.3 Software Engineering Techniques

Irwin et al. [IKL+97] describe an approach, Aspect-Oriented Programming (AOP), to write software that differs from the classic programming paradigm. Instead of splitting programs into modules of executable code, the split is proposed to be along the aspects of the code. This means that certain functionality is supposed to be mapped to a set of aspects, which are essentially capabilities of an object. This allows for an alternative description of objects. A common example for AOP is AspectJ[1] which aims to enhance Java with AOP capabilities. This thesis proposes an approach that makes use of AOP-like concept. The proposed way to generate code that fills log files with certain information uses code annotations which can be mapped to AOP principles. This thesis only provides a very rudimentary proposal for said annotations which is why future work can be done to potentially explore opportunities for a more coherent syntax.

Two of the proposed approaches, described in Section 6.1.8 and Section 6.1.9 are related to the concept of program slicing as discussed by Weiser for example [Wei84]. Program slicing aims to reduce a program to the path that needs to be analyzed, i.e. the path that produces an observed error. This allows the analyzing person to solely focus on a single part of the control and data flow. The resulting *slices* allow an efficient way to follow and understand the source code in a debugging scenario for example. Korel and Laski [KL90] introduce the term of dynamic slicing where the created slices remain executable while allowing to modify certain variable contents. These slices are so-called *backward slices* as they represent the behavior observed so far. Binkley et at. [BDG+04] also discuss the concept of *forward slices* which represent the behavior to be expected and predicted for the near future. Impact analysis, as proposed by this thesis in Section 6.1.8 is comparable to the idea of backward slicing as it aims to find a minimal path within the reasoning process of the operating system. This path originates at a certain point in time and is searched in a backwards direction. Backtracking, as proposed by this thesis in Section 6.1.9 also aims to find a backwards path to a certain decision within the reasoning process. This decision is chosen based on the fact whether it is possible to generate a forward path which reaches a desired decision outcome, which is similar to forward slicing. However, the approach from this thesis cannot be fully matched to the concepts of program slicing, as it is not immediately connected to the specific source code instructions but works on a slightly higher abstraction layer.

[1] https://www.eclipse.org/aspectj/doc/released/progguide/index.html

4 Study of the Problem Analysis Process in the Industry

The first contribution of this thesis addresses Research Question Q1:

> **Q1**: Which scenarios exist in the area of operating system that require explanation?

To do so, a user study is conducted that aims to collect insights into the problem analysis process as executed by professionals. First the goals of this study will be refined in Section 4.1. Subsequently the design of the study will be presented and explained in Section 4.2. Section 4.3 then highlights the results of the study. The crucial outcome of this study with regard to answering Research Question Q1 is the collection of problem analysis scenarios that require the explanation of the internal reasoning process of the operating process. Those will be discussed in Section 4.4.

4.1 Study Goals

In order to investigate if operating systems could benefit from enhanced explainability capabilities it is necessary to determine which scenarios require the derivation of explanations. In order to get to these scenarios it needs to be verified how the overall analysis or debug process affects the interviewed professionals, i.e. how much time they spent on analysis, which approaches they use and how their general proceedings look like. By gathering information on the complexity and time-impact of problem analysis and explanations it becomes possible to draw conclusions on the overall usefulness of improved explainability of the internal reasoning process of the operating system. Furthermore, the evaluation of the proceedings of the interviewees allows to verify their capabilities to describe relevant scenarios targeted by Research Question Q1. Therefore, the conducted study aims to answer the following sub-questions:

SQ1 How much time is spent on problem analysis (in total and per problem)?

SQ2 Which approaches are used in the analysis process?

SQ3 How does the analysis process look like? What are common steps?

SQ4 Which scenarios are potentially common across multiple groups on an abstract level?

Question SQ1 is supposed to evaluate whether it is worth to optimize the analysis process. It would be a valid assumption that there is no value in additional explainability if professionals only spend a fraction of their time with analysis or if each problem can be solved quickly.

In addition to providing input to the answer of Research Question Q1, this study will also gather a baseline for the classification of existing tools in Chapter 5 which will be used to answer Research Question Q2. Question SQ2 aims to collect potential candidates to be explored.

In order to identify potential toeholds for explainability to provide additional value, Question SQ3 aims to find obstacles in the current analysis processes as conducted by the participating professionals.

To finally answer Research Question Q1, Question SQ4 collects scenarios that can be used as concrete examples when discussing the helpfulness of the presented tools and concepts.

4.2 Study Design

This section will explain the overall study design. First the format of the study will be stated alongside an explanation why it was chosen. Afterwards, a description will be given on how the participants were chosen. A brief discussion on the purpose of the posed questions will conclude this part.

4.2.1 Format

As described in Section 2.4, there is a wide range of different forms that can be used to gather information about the users of a program or a system.

The overall goal of the conducted study is to pin-point potential issues in the problem analysis process in the field of operating systems. To do so, the opinions and perspectives of professionals need to be collected. With the focus set on qualitative results over quantitative ones, a user study method that only requires a smaller sample size is sufficient, which disfavors a *survey* as it requires a larger number of participants and lacks the required level of detail.

Diary studies or *field studies* would provide very detailed insight into the daily work of the participants. Unfortunately, it is not easy to find professionals who are willing to go through the massive overhead that is created on their end by taking part in such a study. Therefore, these options have been ruled out as well.

Another approach that could have lead to valuable insights are *focus groups*. These come with a large overhead in preparation and are particularly hard to evaluate. The major drawback lies in the need for a well trained and experienced moderator to keep these meetings efficient and productive which caused this method to be excluded from the available options as well.

The method finally chosen is the *interview*. It allows to collect qualitative insights in a time efficient way which makes it easier to find professionals willing to participate. The interviews were scheduled for 30 minutes each and held online[1]. The question were on display throughout the conversation and all answers were written down on screen to make sure that the answers were captured correctly by the interviewer. The chance to get into a dialog while keeping the discussion structured was deemed to be the best compromise for the given purpose. An experienced interviewer would be beneficial here, but is not strictly necessary, which is in favor of the author of this thesis.

[1] due to the COVID-19 pandemic

4.2.2 Participants

All participants of the study are professionals in their field with multiple years of experience. They were chosen from a range of different job roles:

- Operating System developer: Involved in the code maintenance and enhancement of the system with new feature.
- Performance Analyst: Analyzes bottlenecks or unexpected behavior with regard to the performance of the system.
- System Administrator: Maintains, manages and configures computation centers and the involved operating systems.
- Support Professional: Analyzes unexpected behavior of the operating system on the customer side and provides guidance

The common feature of all involved job roles lies in their need to have a detailed understanding of the internals of the operating system to do their work. They all could potentially benefit from extended explainability of operating systems which is why those roles were selected. The specific set of job roles was gathered by the availability of the professionals during the time frame of the execution phase of the study.

By taking into account a broader range of professions, the study also aims to identify whether there are common issues across the mentioned roles that could be addressed with Explainable Software. Interviewing only a specific group of professionals within a single job role bears the risk to miss special requirements and needs of other job roles, which is another reason why a more diverse set of interviewees was chosen.

4.2.3 Questions

The set of questions is composed of multiple parts, which will be described in the following. The full set of questions and the introduction for the participants can be found in the appendix in Section 9.1.

The first block (Questions 1.1 to 1.4 in Table 4.1) is supposed to verify that the interviewees have relevant knowledge in the topic and to make sure that they are working with operating system related topics and problems on a regular basis. These questions will be used to retain only study results from those interviewees who have actual experience in the area. A set of operating systems is offered to choose from to potentially adapt the further flow of the interview. This block of questions is also intended to provide an easy start into the conversation.

1.1	What is your current primary job role?
1.2	How frequently do you work with the following operating systems in your current job role?
1.3	How much time do you spend working with the following operating systems in your current job role?
1.4	What level of operating system knowledge does it take to do your current job properly?

Table 4.1: Questionnaire questions 1.1 through 1.4

The second block (Questions 2.1 to 2.7 in Table 4.2) aims to provide insight into the analysis process itself with Questions 2.1 through 2.6 aiming to answer sub-question SQ1 and Question 2.7 answering sub-question SQ2. Questions 2.1 to 2.3 target to find out whether problem analysis is a major part of the daily work of the interviewees while also querying how much time is spent per problem. A distinction between average and complex problems is done to take into account that there might be a significant deviation between very simple issues and ones that require a large amount of research or debugging. To account for potential differences in weekly working hours of the interviewees, question 2.1 queries percentages in order to identify how much problem analysis the interviewee is doing on average. This especially addresses potential differences between full-time and part-time employees. Question 2.2 and 2.3 gather a less concrete metric by just distinguishing between magnitudes of time (hours to months) as their purpose is to gather an insight into the distinction between average and complex problems. Questions 2.4 and 2.5 are supposed to provide insights into the possible causes of the analyzed problems. The offered error types cover the different roles that are involved in the process (user error → end-user, setup error → admin, bug → developer, and finally unexpected reasons which are out of the control of all involved parties). Question 2.6 then aims to determine if there are additional obstacles that impede the analysis process. These might not necessarily be circumvented by Explainable Software approaches but should be mentioned anyways. Question 2.7 finally will provide a list of industry relevant tools that will be discussed in Chapter 5.

2.1	How much of your work time do you spend on average analyzing problems per week?
2.2	How much time do you usually spend on an average problem?
2.3	How much time do you usually spend on a complex problem?
2.4	How common are the following reasons to be the cause of the problems you have to analyze?
2.5	How much time does a problem analysis take on average given the error cause?
2.6	How much time do the following factors consume during your average problem analysis?
2.7	Which strategies and tools do you usually use to analyze problems?

Table 4.2: Questionnaire questions 2.1 through 2.7

The final block (Questions 3.1 to 3.4) is supposed to provide actual examples of the analysis process, identify potential issues and collect sample scenarios that can used in the classification Chapter 5 as well as in the evaluation Chapter 7. These scenarios will be used to apply the discussed approaches of Explainable Software on an abstract level and evaluate them with the feedback of professionals in an additional user study.

Questions 3.1 through 3.3 also aim to provide answers to sub-question SQ3 while Question 3.4 aims to answer sub-question SQ4 and finally Research Question Q1.

To gather these scenarios, the interviewees are guided through a fictional error scenario in which they have to go through a deep analysis in an incremental way and describe what they would do. Starting with the error report of a user, they have to identify what the user did and if the user may have done an error on his/her side in Question 3.1. Subsequently in Question 3.2, it is assumed that the user did everything correctly and it becomes necessary to analyze the environment setup and configuration for potential problem. In the final step in Question 3.3, the interviewees are supposed to consider that the system is setup correctly and the only remaining option is a flaw in the reasoning process of the operating system itself. These incremental, yet deep, analysis journeys are assumed to be those which are among the hardest to evaluate as they finally require to understand the current

decision process of the operating system. Therefore, Question 3.4 asks the interviewees to describe scenarios like this to get insights into actual problems that can be used to discuss whether increased explainability on the operating system level can be of value.

The chosen incremental approach also aims to assist the study participant in understanding what kind of scenarios are supposed to be gathered by Question 3.4. By explicitly narrowing down the scope of the analysis scenario the participants are guided to ignore too trivial scenarios while also focusing on such scenarios which can actually be traced back to the internal reasoning process of the operating system.

4.3 Study Results

The following section will provide and discuss the outcome of the previously described study. Focus will hereby be set on the following subareas. Section 4.3.1 covers the time spent by the professionals when analyzing problems while differentiating between complexity of the problem and the type of its cause. Section 4.3.2 lists and briefly introduces the tools used by the study participants. A subset of these tools will be shown in further detail in Chapter 5. Section 4.3.3 focuses on the analysis process as described by the interviewees and will highlight potential aspects where Explainable Software can provide additional value. Finally, Section 4.4 provides error scenarios that the professionals encountered in their job. Transcripts of the answers provided by the interviewees can be found in the appendix in Section 9.1.

All participants of the study provided answers to Questions 1.1 through 1.4 that confirm the assumption that their level of qualification allows them to provide profound input to the study.

4.3.1 Time Effort

Apart from one operating system developer, all interviewed professionals stated, that analyzing problems consumes at least half of their working time (Question 2.1) as shown in Figure 4.1. It is noteworthy, that the operating system developer stated the lowest quotas across the interviewed professionals. Performance analysts and especially support personnel are using a higher percentage of their working time for problem analysis. The answers have been aggregated across all operating systems.

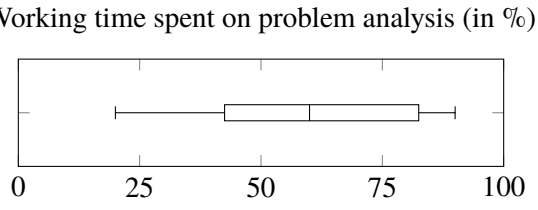

Figure 4.1: Visualization of answers to Question 2.1

While average problems only require hours to days to analyze (Question 2.2), complex problem can potentially take weeks to be sorted out (Question 2.3). A visualization of the given answers can be found in Figure 4.2. Again, the answers have been aggregated across all operating systems.

This indicates clearly that the analysis process is a major part of the daily work for people working close with operating systems. Identifying a way to save time in this process would therefore be beneficial. An investigation whether Explainable Software is able to provide such a saving is therefore concluded to be worthwhile.

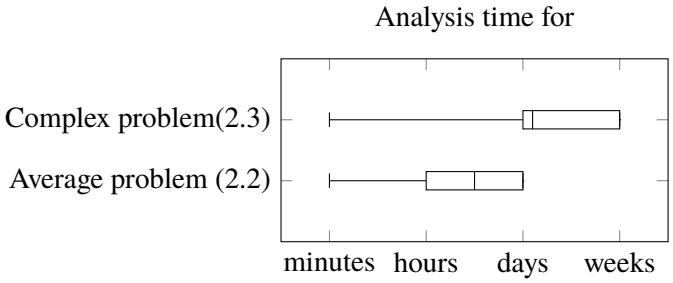

Figure 4.2: Visualizations of answers to Questions 2.2 and 2.3

Questions 2.4 and 2.5 provide insight into how often certain error causes are determined in the end and how much time the respective analysis takes. The answers to these questions are visualized in Figure 4.3 and will be discussed in the following passage.

One available option are external reasons, i.e. an unexpected rise in workload or a power outage. The interviewees stated that these issues only occur rarely, but vary largely in time effort during analysis. The cause is either immediately evident or is particularly difficult to be analyzed. Therefore and due to the uniqueness of these situations it is non-trivial to identify and explain these issue programmatically. Because they are also not a major pain point, as they do not occur frequently, these issues will either not profit from Explainable Software approaches or take an immense amount of additional research. Hence they will be neglected in this thesis.

Another option are user errors, e.g. end-users of the operating system report an error that they caused by not following the documentation or entering wrong data. The occurrence of this error cause varies drastically. But at the same time, all interviewees agree that the analysis of these problems is completed within a short amount of time. The wrongdoing of the end-user can be identified quickly due to the deep knowledge of the operating system that the interviewees have. Value through improved explainability can be generated if these explanations can be presented to the end-users themselves such that they don't have to contact their support teams. The professionals would not necessarily profit from generated explanations as the analysis process itself is already not very time consuming. Therefore, this error type will also be neglected.

Setup (or configuration) issues are an additional option. These errors cover systems which have been setup with wrong or inconsistent parameters or configuration files. With less variance in the occurrence frequency, these problems occur more often on average than user errors. Hence, these situations are not a rarity. At the same time, the required time to identify a bad configuration as the root cause is significantly higher. The reason behind this is an ambiguity in the correlation between configuration parameters and the internal reasoning process of the operating system, i.e. what influence did the parameter have on a certain decision. Therefore, the professional has to either know how the configuration contributes to the overall operating system behavior or needs to perform a deep dive analysis within the code if the documentation is not sufficient. This poses the additional problem, that is hard to predict the potential ramifications of a configuration change,

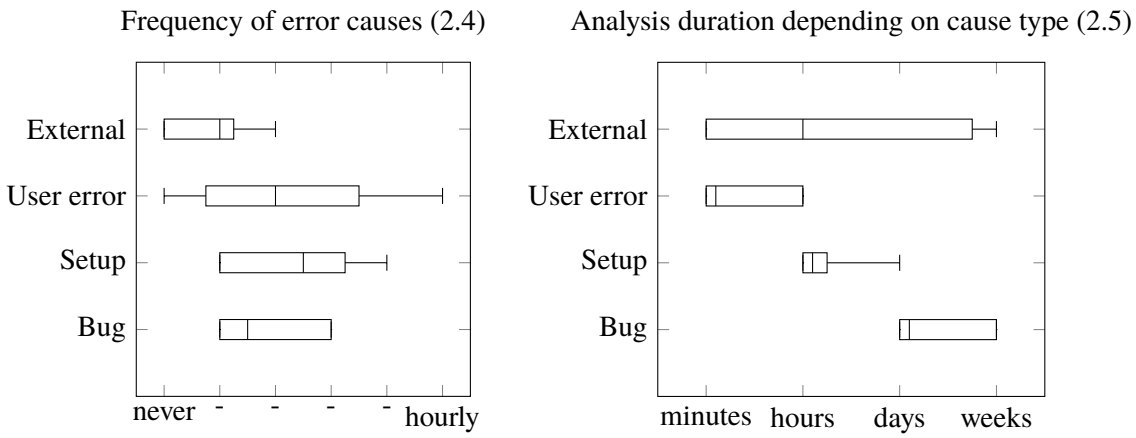

Figure 4.3: Visualizations of answers to Questions 2.4 and 2.5

which makes it hard to judge whether such a change will fix the problem while also not causing additional problems through side effects. Increased explainability can be of huge value in this case by allowing a better intuition when judging the correlation between the action of the operating system and the configuration.

The final available option are operating system bugs, i.e. cases where the behavior of the operating system differs from the design of the development teams. Potential reasons are programming errors or unexpected code paths that were not accounted for during the design process. While not occurring particularly frequent, analyzing these type of issues require the most time of all encountered options. During analysis it is necessary to identify that the operating system does in fact not behave according to its design or documentation. This requires a deep understanding of the internal reasoning process in order to distinguish between a bug and another type or error cause. Explanations on the decision process have potential value as well in this scenario as they would make it easier to understand how an erroneous decision was made. This means for example to be able to easily see which information was used ot get to the decision and which premises were given. Improved Explainability can again be of value here by shortening the analysis time by giving faster insights and a straightforward guidance to the location were the decision was made.

Additional mentions are hardware and network errors, both occurring occasionally and taking hours to days to analyze. These are particularly difficult due to two reasons. One being that the interfaces over which the operating system is communicating with the peripheral hardware are not clearly documented in several cases. As a consequence there are also no coherent logs available. Instead it is necessary to manually match the operating system logs to the hardware logs. This aspect needs closer interaction between the involved parties which can later be a potential point to also evaluate the value of Explainable Software in this area. The second reason why analysis of hardware and network errors are time consuming lies in the fact, that operating systems have the capability to adapt to changes automatically. These adaptions can occur without any notification but may interfere with the overall performance of the system. Identifying an occurrence like this requires to analyze a large amount of log data and the experience to know that this can even happen. Improved explainability can help here to report and justify adaptations due to hardware and network changes.

Figure 4.4 visualizes the answers for Question 2.6 which aims to ensure that the major time efforts spent on analysis are posed by issues that can be addressed by improved explainability. The interviewees stated that the actual analysis is not the only time consuming part but still represents the largest part in their analysis process.

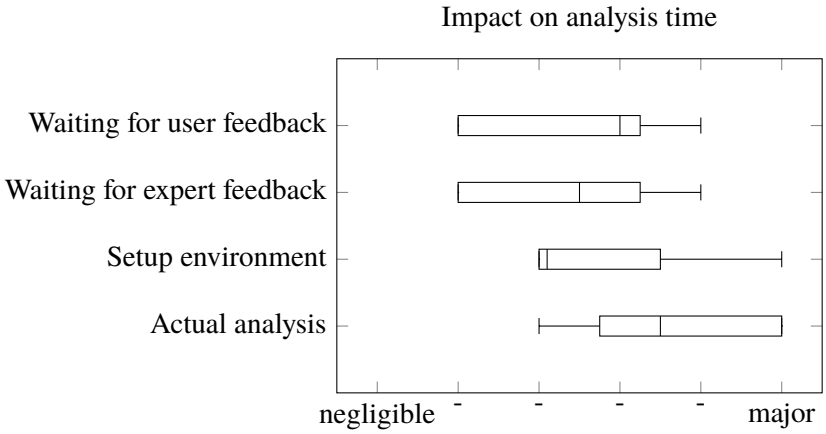

Figure 4.4: Visualization of answers to Question 2.6

Setting up environments is a first step in order to recreate the scenario and contributes as well to the analysis process. This is not necessary for all cases though and could be potentially avoided. The latter can be the case, if the sole purpose of recreating is to be able to play around with the system in order to find out what it does. Creating the environment is a time consuming factor and could be potentially restricted when the operating could justify and explain its actions.

One interviewee stated that time consumption when waiting for experts or users is high in his/her case due to time differences. Another interviewee stated time difference as an issue as well, but with less impact. These general communication overheads can potentially be reduced by improved explainability. If the operating system can answer the questions that would be sent to an expert for example, this communication path could be removed.

An additional mention was the problem to figure out where to start the analysis within an operating system core dump. This issue was categorized as a major pain point by the interviewee. Improved explainability has the chance to shorten this process by pointing the analyzing person straight to the relevant information.

4.3.2 Tools

The purpose of Question 2.7 is to collect a list of tools actually used by the professionals during their analysis process. A list of options was given in order to give the interviewees an idea which kind of tools were queried. This list is not complete and has potentially introduced bias. The interviewees ignored the list for the most part and named additional tools that they are using. A subset of the named tools will be discussed in Chapter 5.

One outcome worth mentioning is that all interviewees stated that they are not getting all analysis capabilities from the tools available to them. This is not necessarily a flaw of the existing tools, but due to the very specialized environments the interviewees are working in. In order to get such very specific insights they use self-written tools that are either not available to the public or end up as a contribution to open-source software. The latter holds true especially for Linux environments.

4.3.3 Analysis Process

Questions 3.1 through 3.3 aim to collect insights into the analysis process of the interviewees. The following paragraphs will discuss the steps described by the participants of the study. Notably, the overall structure of the analysis process is very similar across the different job roles of the interviewed professionals. Hence making it negligible to attempt a distinction between different personas in the problem analysis context.

An initial step that multiple interviewees agreed upon was the gathering of debug data. This data can be a full system core dump or other component specific data. The received files often contain a big chunk of data, wherein it is hard to find the correct spot which contains the information needed. It was mentioned that there a potentially multiple spots. This could be addressed by an improved explainability that provides pointers into the debug data which lead to the decision and the input parameters of the internal reasoning process.

Another major step that emerges from the study is the step to distinguish the expected behavior from the observed one. This includes multiple sub-steps. First, the verification of the error description, i.e. to check whether the problem is reproducible and viable. Subsequently, the documentation ought to be studied if the expected behavior cannot be derived from experience. Furthermore, it might be necessary to compare the gathered debug data to the source code in order to verify whether the system behaved as expected or not. Explainable Software could be of use here by providing quick insights into the reasoning process, not only which data was used but also how it was used. This would be beneficial as it would support the understanding and training process as well.

Once the error is deemed viable, the necessary data that led to the decision is identified, and the divergence from the intended behavior is verified, the final step is the correction of the problem. Finding and implementing a solution is non-trivial because it is not easy to predict what impacts a change to a configuration or the source code might have. While fixing the reported issue, it might create several new ones. Overall, these steps match those described in the literature, for example by Zeller [Zel09]. An improved explainability can be of value here by making the operating system more predictable and comprehensible, which in turn would potentially increase the trust into the system.

4.4 Scenarios

Question 3.4 finally aims to collect actual problem analysis scenario that will be used in Chapter 6 in order to give concrete examples how the process can be improved by having an increased level of explainability for the internal reasoning process of the operating system. The following paragraphs

will briefly describe three of the collected scenarios and highlight their underlying abstract problem. Other mentioned scenarios are very similar in nature which shows that the different problem causes affect all job roles found among the interviewees.

Scenario 1 - Unexpected Internal Adjustments

The reported problem was an unexpected, erroneous behavior of a peripheral device. The root cause was eventually determined as a wrongly set bit within a data structure which was under control of the operating system and used to access the device. While it was possible to identify the mentioned bit fairly quickly as the cause of the problem, it was a very time-consuming effort to determine which component was responsible for this change. There were no traces or debug data available that allowed conclusions on when and why the bit was flipped, hence the analysis process finally developed into a intensive study of the source code.

Scenario 2 - Unexpected Hardware Interactions

Additionally to Scenario 1, the interaction between the operating system and peripheral hardware can create another set of issues. Changes on the hardware site can lead to automatic adaptions by the operating system. These are not necessarily reported properly while still having impact on the overall performance of the system. This unexpected change of behavior is hard to explain as there might be no trail to follow. Another aspect in this area are the non-transparent interfaces, driver and configuration options used for the communication between operating system and the external hardware. These issues demand a time consuming analysis process as also stated in one answer to Question 2.5.

Scenario 3 - Unexpected Use of Resources

The final scenario that will be used in the further section of this thesis has its origins in the performance and workload analysis. Two interviewees described situations where available resource were not used as expected. In one case a single unit in a set of identical processors was performing significantly worse than its peers. The identified cause were on the one hand side network issues but on the other hand also the unexpected scheduling of infrastructure workload to the mentioned unit due to a defaulting mechanism. In another case given by an interviewee, a user deviated from the configuration recommendations provided by the development team. This caused unexpected behavior in the processor assignments during scheduling as well. The combination of settings chosen by the customer lead to an unexpected code path. In both case an extensive analysis of debug data was necessary to identify that the underlying problems were missing or unexpected configuration choices.

4.4.1 Threats to Validity

Due to the limited time frame of this thesis, certain cutbacks had to be made which have to be taken into account when reading the study results.

There are only 6 interviewed professionals who have taken part in the study. In order to get a broader overview, more interviews will have to be conducted. Furthermore, the interviewees are all working for one company, which potentially gives a one-sided overview. The company is working with a wide range of different operating systems. Therefore this drawback is not as critical as a certain variety of different operating systems can be covered.

The design of the study followed the assumption that it might be possible to derive different personas and behavioral approaches based on the profession. From the small set of interviewees, these assumption could not be proven, hence, the evaluation in Chapter 7 will not focus on that aspect.

Apart form the interviewees, there are also flaws on the interviewer side. The interviews were conducted by the author of this thesis himself. Unfortunately, the experience in the art of interviewing has to be seen at an entry level. While attempting to avoid introducing unintended bias or posing questions in a suggestive way, there is no guarantee that this has been achieved at all times.

4.5 Summary

The overall goal of the conducted study was to answer Research Question Q1:

> **Q1**: Which scenarios exist in the area of operating system that require explanation?

To do so, it was first confirmed that the analysis of operating system issues is a time consuming task (SQ1). Subsequently it was established that the major effort that has to be invested is the manual analysis of large amounts of debug data, partially alongside a source code analysis. By collecting insights into the actual analysis process of the interviewed professional, the time consuming nature of the process was highlighted again, while also providing concrete pain-points and problematic aspects. Adding capabilities to the operating system, that justify and explain the internal reasoning process, can create potential value that allows for an easier and more convenient analysis process. To evaluate how already existing tools cover this area of providing insights and explanations in the context of operating systems, a set of tools was collected, used by the professionals, answering sub-question SQ2. Additionally insights into the general analysis process were collect, tackling sub-question SQ3. Also, a set of scenarios was collected (SQ4) and then converted into abstract problem statements that will be used in Section 6.2 to demonstrate how approaches of the area of Explainable Software can potentially provide additional value to the operating system analysis process. The described scenarios highlight that there are circumstances where it is necessary to have deep knowledge of the internal reasoning process of the operating system. This knowledge can be either gathered by years of experience in the field or for example by adding explainability on the decision level to the analysis and debug capabilities of the operating system. Possible approaches will be shown in Chapter 6.

In conclusion, this chapter provided answers to Research Question Q1 by providing a set of scenarios that highlight the necessity and complexity to derive explanations for the behavior of the operating system.

5 Classification of Existing Tools

The following section will introduce a classification of operating system analysis approaches. First, Section 5.1 will introduce the classes that will be used in the classification process. Subsequently, Section 5.2 will give an overview over a subset of tools named by the professionals in Section 4.3.2. The abstract concepts of these tools will then be elaborated and classified. A soft classification will be used as most tools cannot be put strictly into one class due to their continuously growing set of features. This means that a tool can be part of multiple classes. To allow further distinctions a score will be provided on how well the tool fits into the given class.

The tools chosen for discussion are taken from the answers of Question 2.7 of the questionnaire in the user study shown in Section 4.3.2. The subset from this tool is chosen based on the availability of documentation and number of mentions by the interviewees. This means especially that special purpose tools that are not available to the public will not be discussed.

The chosen tools and their classification will then be used to provide answers to Research Question Q2:

> **Q2**: Which approaches are used in the industry to explain the operating system decision process today?

5.1 Classes

A total of three tiers of classes will be presented in the following paragraphs which will then be used in Section 5.2. Each tier contains three classes. The three tiers *use cases*, *time scope* and *presentation* are chosen to allow a multi-dimensional view on the capabilities of the discussed tools. The following paragraphs will provide details on the classes themselves.

Tier 1 - Use Cases

The first tier to distinguish the presented tools is based on their use cases.

Performance and resource usage analysis is a core field of interest when it comes to the efficiency of large computer systems. In order to identify performance bottlenecks or over-provisioning, it is necessary to correlate a wide range of different metrics. Tools that aim to aid in this area therefore need to have capabilities that allow a broad overview either on the whole system or a larger subsystem. At the same time they need to provide capabilities to access and process detailed information on the system performance.

Behavior analysis and debugging is necessary if errors need to be identified where the internal reasoning process of the operating system needs to be understood. In order to follow the reasoning process, a very specific set of debug data needs to be made available and presented in comprehensible fashion.

Event and message logging provides a broad overview on the overall state of the system. This use case does not require very detailed insight into debug data, it is merely intended to identify significant problems quickly.

Tier 2 - Time Scope

The second tier will evaluate the time scope of the analysis that can be performed with the given tools.

Real-time analysis and monitoring is focusing on the current state of the system. The goal here is to ensure that the system performs as expected and is not facing any error scenarios. Data of interest here are for example the current CPU consumption or the number of active tasks or jobs.

Long-term analysis and post-mortems are performed after an unexpected scenario was encountered. The purpose of this type of analysis is to determine the root-causes of erroneous behavior or to identify and evaluate anomalies. Relevant data here can be a snapshot of the system in its error state or data along a certain timeline if a potential build-up to the problem has to be taken into account.

Prediction and simulation tools are used for example to estimate the behavior of the operating system given a set of resources or expecting a certain amount of workload. This is particularly useful when it comes to solve previously identified issues, as it might be non-trivial to assess how a change in the system configuration or an adjustment in the available resources might impact the performance in the future.

Tier 3 - Presentation

The third tier will differentiate the presented tools by their presentation style.

Aggregation of data allows to condense information into characteristic values, e.g. a mean value of a set of values. The goal here can for example be to gain an immediate overview on the general state of the system or to gain insights into meta information which requires the analysis of multiple sub-components of the operating system. It requires the person who analyses the data to precisely know which combinations of values are to be expected.

Visualization is a more natural way for a human to look at data. Graphical representation can make it easier to compare data and analyze time-dependent trends. It also makes it easier for humans to recognize patterns in a large set of data. Visualization can be performed on the raw debug data or on previously aggregated data. Hence, there is a potential dependency to the previous category.

Explanation of the behavior of the operating system is often synonymous to te interpretation of log and debug data by an analyst. Tools can support this process by providing pre-formulated explanations of data and combinations thereof in natural language. Another way to assist the analyst might be the collection of data that contributed to a certain decision and highlight how impactful each value was.

5.2 Classification

The following paragraphs will iterate through the list of official tools used for problem analysis which were mentioned by the interviewed professionals in Section 4.3.2. A brief introduction will be followed by the classification itself. A value between 0 and 3 will be assigned per category for every tool, which will indicate how much focus is put on each category by the tool. This soft classification is necessary as most tools cannot be assigned to a specific class due to their large amount of supported features and functionalities. Brief explanations on these decisions will be given where needed.

The amount of values is chosen to allow a distinction between the discussed tools while avoiding a too granular resolution where it becomes impossible to properly reason why a tool is associated with a certain value.

The values have the following meaning:

0 → The tool does not fit into the category. This means there is either no built-in support or the category does not coincide with a valid use-case for the tool.

1 → The tool can be used in the given category, but it is not its main purpose. Additional knowledge or customization is necessary to leverage the tool. There is also no dedicated interface to other tools.

2 → The tool offers rudimentary support for the category. It delivers basic insights and/or the output can and is intended to be used by other tools.

3 → The tool offers full support for the category. Very detailed insights are possible.

A summary of the value assignments can be found in the appendix in Section 9.2.

5.2.1 top

The first tool to be discussed is *top*[1] which is integrated into all major Linux distributions by default. Based on a command line interface, it allows to monitor the resource usage and other properties of all currently active processes. Resources are CPU and memory consumption, which is rudimentary information. The displayed properties are the current priorities of the processes, their niceness and processor allocation details. The available fields can vary depending on the Linux distribution and the underlying hardware architecture.

The main focus of *top* lies on real-time performance evaluation. Displayed values are updated continuously. A very limited set of load average values for intervals of 1, 5 and 15 minutes allows a rough estimate whether the workload on the system is increasing or decreasing. Visualization is negligible as the only available graphs are very simple block diagrams for resource usage. Behavior analysis is beyond the scope of *top*. The same goes for log analysis capabilities and simulation aspects. It does also not provide explanations of the reasoning process of the operating system.

[1] https://man7.org/linux/man-pages/man1/top.1.html

Overall, *top* allows to monitor the performance of a system in real-time by providing an aggregation of the main performance metrics. Giving a brief overview over the state of the system is the core aspect. A detailed analysis is usually not possible. The assignment of *top* into the classes can be found in Table 9.1.

5.2.2 SDSF

The System Display and Search Facility (SDSF) for IBM z/OS[2] offers a much more detailed overview and level of insight into the operating system. In addition to a z/OS console interface available through the Interactive System Productivity Facility (ISPF), SDSF allows programmatic interactions via Java or REXX.

Similar to *top*, SDSF allows to monitor the resource usage and properties of currently active processes on the system. While *top* only aims to provide a broad overview over core system performance metrics, SDSF offers very detailed insights. Aside from a more granular representation of memory usage, data on the paging behavior for example, SDSF offers detailed insights into the I/O and network activities of the system. The displayed values include information like average response times and utilization percentages for devices and network connections.

Again in a similar manner as *top*, SDSF displays priority information and properties of active processes. Supplemental data is available on the overall system state, i.e. version of core components and data structures and parameters of the last boot process. In addition to the display of real-time performance data, SDSF provides overviews on the output of other tools like Health Checker, which will be discussed in a later paragraph. Some historic information can be gathered by accessing the core system logs which persist all messages issued by the system.

Visualization are not part of the concept of SDSF. Neither are simulation capabilities and explicit explanations of the reasoning process of the operating system.

The main focus of SDSF lies on real-time performance evaluation. By aggregating a wide range of system data, it allows to generate a detailed overview of the current state of the system. The additional access to log streams provides a way to analyze some high-level long-term trends and changes. The assignment of SDSF into the classes can be found in Table 9.2.

5.2.3 sysstat

A more detailed view of performance metrics on Linux can be provided by *sysstat* (system statistics)[3], which is a collection of tools that can be additionally installed onto most common Linux distributions. Included in the collection are for example the following tools:

iostat: Input/output (I/O) statistics representing the read/write performance for local or remote storage devices

mpstat: CPU statistics that provide insights into how much time was spent per processor in a certain state, e.g. idle, iowait, etc.

[2] https://www.ibm.com/docs/en/zos/2.4.0?topic=security-sdsf
[3] https://github.com/sysstat/sysstat

pidstat: Statistics for active processes on the system and their resource usage as well as information on paging frequency

These tools allow a very detailed analysis of the current state of the system. Additionally, the included tool *sar* allows to schedule the collection of performance data over a longer period of time with cron or systemd is also included in the *sysstat* collection. This allows to gather long-term performance and activity data.

Using the also included *sadf* it is possible to export the data generated by *sar* into machine readable formats, e.g. CSV, XML, etc. This output can be used to feed other tools like Microsoft Excel or OpenOffice which can then generate graphs and charts. In fact, there is a wide range of available visualization tools that aid in the analysis process by visualizing the available *sysstat* data in useful ways. Similarly it is possible to use the collected data to generate predictions on the future workload trends by further evaluating said data. A prediction or simulation of the reasoning process is not supported. Neither is dedicated access to message and event logs.

Similarly to SDSF and *top*, *sysstat* allows real-time performance analysis. Additionally, it provides support to collect long-term data, which can be used to visualize and analyze the performance of a system on a broader scale. The detail level of the collected allows to infer certain operating system behavior as these values are part of a subset of decisions in the internal reasoning process. This requires deep knowledge of the Linux kernel, as there are no explanations how the available data is used apart from the source code. The assignment of *sysstat* into the classes can be found in Table 9.3.

5.2.4 RMF

The Resource Measurement Facility (RMF) for IBM z/OS[4] works of a very similar concept as *sysstat*. A set of three so-called *Monitors* offers data for a variety of performance reports.

Monitor I continuously collects data on a user-defined frequency. The gathered information consists for example of CPU usage, paging behavior or I/O consumption and performance. These data points allow the deduction of long-term performance analysis and can be fed to the built-in post-processor component which allows to generate visualizations like graphs or charts. Monitor II allows to create reports of the running system based on comparisons to previously taken report snapshots. This includes possibilities to evaluate performance metrics for user-defined time intervals. Monitor III provides a real-time view of the running system and can be used for short-term interactive performance analysis. It can display the current system status and offers short interval data like average response times per workload type as well as summary statistics across all workloads.

RMF works with System Measurement Facility (SMF) data to store and display the performance metrics. These data records contain very detailed information on the overall system performance. Additionally it is possible to derive detailed insights into the behavior of the system. This is possibly, because the SMF data records used by RMF contain information on metrics that are used by other components to take decisions. This does still require very specific knowledge of the operating system, as there are no explanations how said data is explicitly used in the reasoning process. Event and message logs are not explicitly accessible from within of RMF.

[4] https://www.ibm.com/docs/en/zos/2.4.0?topic=security-rmf

The level of detail provided by RMF goes beyond the scope of *sysstat*. A Linux tool that closes this gap at least partially is *perf*. It allows to analyze hardware events like cache misses, branch instructions or software events like context switches, page faults, etc. This tool has not been mentioned by the professionals and shall therefore not be covered in further detail.

Overall, the scope of RMF is to enable real-time and long-term performance analysis. To a certain degree it even allows to analyze decisions taken by the operating system due to the very detailed information stored in the used SMF records. A built-in support for visualizations brings the collected data into a more humanly comprehensible format. Additionally, these reports can be exported in machine-readable format to allow further custom analysis by the user. The assignment of RMF into the classes can be found in Table 9.4.

5.2.5 IPCS

In contrast to the previously discussed tools, the Interactive Problem Control System (IPCS) for IBM z/OS[5] does not collect data, but offers an interface to analyze system memory dumps and traces.

Memory dumps on IBM z/OS can either be created by a process terminating in an error state, an invalid state of the operating system or user defined conditions and commands. The resulting data structures are large chunks of binary data which contain a complete or a pre-defined subset of the memory that the system was working with. This data is not human-readable, which is where IPCS comes into play. It provides predefined shortcuts that allow the user to navigate through relevant parts of the memory dump and format areas that are regularly needed during debugging into a humanly comprehensible form. These overlays allow for quicker insights into the state of the system when the snapshot was taken. Within the available data it is possible to backtrack certain decisions of the operating system as all data that was available to the system is part of the memory dump. Unfortunately, this requires deep knowledge of the operating system itself and can partly only be done by the developers of the component under investigation. This behavior is very similar to the GNU Project Debugger (gdb)[6] which is part of most Linux distributions.

Another capability of IPCS is the analysis of traces that are either written by the different operating system components or by the Generalized Tracing Facility (GTF)[7]. The latter is a service aid very similar to the Linux tool *dtrace*[8], which essentially allows to trace certain events during execution of the operating system. Such event can for example be I/O interrupts or starts and terminations of processes. While *dtrace* allows the user to write programs that can be run when an event occurs, GTF traces always show the same output fields, which can then be used for further analysis. Component level traces are explicitly defined by the components that write to the trace. This means especially, that the included information can be used to analyze problems and also gain insights into the reasoning process.

[5] https://www.ibm.com/docs/en/zos/2.4.0?topic=determination-ipcs
[6] https://www.gnu.org/software/gdb/documentation/
[7] https://www.ibm.com/docs/en/zos/2.4.0?topic=aids-generalized-trace-facility-gtf
[8] http://dtrace.org/blogs/about/

Overall, IPCS is not meant to aid in long-term performance analysis scenario. The main scope is problem debugging in either an error state or another predefined state of interest. Aggregation and formatting of data is available to shorten the process while visualizations are not. Performance metrics can be accessed to gain a short-term insight into recent events on the system. The availability of all data and variables of the active system in combination with the detailed traces allows to reconstruct the recent behavior and decision process. Explanations have to be generated manually as there is no automatic or integrated support to do so. The assignment of IPCS into the classes can be found in Table 9.5.

5.2.6 IBM Health Checker for z/OS

The IBM Health Checker for z/OS[9] follows a concept that differs from the ones of the previously discussed tools. Instead of collecting, aggregating or visualizing performance or debug data, its main objective is to identify and predict potential problems by monitoring certain aspects of the system. The goal here is to give the administrator or user enough time to prevent a critical situation that might, in the worst case, cause an outage or data loss.

The predefined checks cover multiple areas. The settings and configuration values of the operating system which can adjust dynamically are monitored for unexpected changes. Performance and workload metrics are compared against threshold values and are reported when being approached, including predictions whether these limits will be exceeded or violated. In addition to the included checks, it is possible to provide customized ones for user-specific scenarios.

While there is no support for performance analysis or insights into the behavior and reasoning process of the system, the IBM Health Checker for z/OS aims to predict critical situations of the system. Once such a prediction comes to an undesired outcome a warning is issued accompanied with a brief explanation. These explanations contain information on the nature of the identified problem and recommendations on how to prevent or fix the issue but do not explain the internal reasoning process itself. It is also not possible to perform complex what-if predictions in order to evaluate for example how a certain configuration change would affect the system performance in the future. Real-time and long-term analysis are not within the scope of the IBM Health Checker for z/OS. The assignment of IBM Health Checker for z/OS into the classes can be found in Table 9.6.

5.2.7 Predictive Failure Analysis and Runtime Diagnostics

Predictive Failure Analysis (PFA)[10] and Runtime diagnostics[11] are two tools that work closely together to help an IBM z/OS admin or user to identify and understand problems and potentially critical situations.

Runtime Diagnostics allows to shorten the analysis of system memory dumps by automating re-occurring manual tasks. It will for example go through the system logs to look for critical messages while also evaluating the core memory for certain patterns. Problems that can be identified and

[9] https://www.ibm.com/docs/en/zos/2.4.0?topic=level-health-checker-zos-users-guide
[10] https://www.ibm.com/docs/en/zos/2.4.0?topic=management-predictive-failure-analysis
[11] https://www.ibm.com/docs/en/zos/2.4.0?topic=management-runtime-diagnostics

reported range from simple cases like a missing mount of a file system to potentially looping processes or lock conditions. The outcome is a set of messages that point to a large, yet limited list of problems that can be potentially identified in a system memory dump. The output is purely text based, without support of visualizations.

PFA makes use of the IBM Health Checker for z/OS and Runtime Diagnostics insights to create a machine learning model that represents the overall behavior and performance of the system given the current workload. This model is then used to predict the further development of the system to indicate potentially critical situations before they actually happen, e.g. memory exhaustion. Furthermore, the model is used to identify anomalies during workload processing based on resource usage or message issuance for example. These capabilities can be used to get insights into long-term behavior of the system, but this is not within the main scope. Neither is real-time analysis.

Overall, the combination of these two tools aims more towards prediction and explainability than the previously discussed ones do. By autonomously identifying, reporting and explaining potential error scenarios, the user is able to safe precious time. The explanation aspect does not specifically cover the reasoning itself though. An indication on what happened and a brief message on the nature of the problem and in some cases also references on how to fix it are provided. This does still not necessarily allow to follow the reasoning process that led to the scenario. The assignment of PFA and Runtime Diagnostics into the classes can be found in Table 9.7.

5.2.8 IBM OMEGAMON

The OMEGAMON tooling family[12] is another vehicle that allows to monitor the performance of IBM z/OS systems. One sample scenario that is covered by this tool is the assistance in identifying bottlenecks. This is done by providing data aggregation and visualizations of various performance metrics. These can for example be the resource usage on different abstraction layers, e.g. on process level or on a subsystem layer. Like previously mentioned tools there is also support available that allows loop detection by identifying modules that take particularly long to run within a larger complex.

Another aspect is the calculation of necessary capacity which allows to adjust for resource shortages in advance. This is also supported by predefined scenarios that allow quick insights into workloads that are missing there performance goals or workload types that are competing for the same set of resources.

Overall OMEGAMON can be seen on a similar level as *sysstat*. While the differences in the underlying architectures are significant, the goal is the aggregation of various performance metrics in both cases. OMEGAMON has additionally built-in support for visualizations and predefined scenarios as well as resource calculation tools which allow a rough estimate of future capacity requirements. The assignment of OMEGAMON into the classes can be found in Table 9.8.

[12] https://www.ibm.com/it-infrastructure/z/omegamon

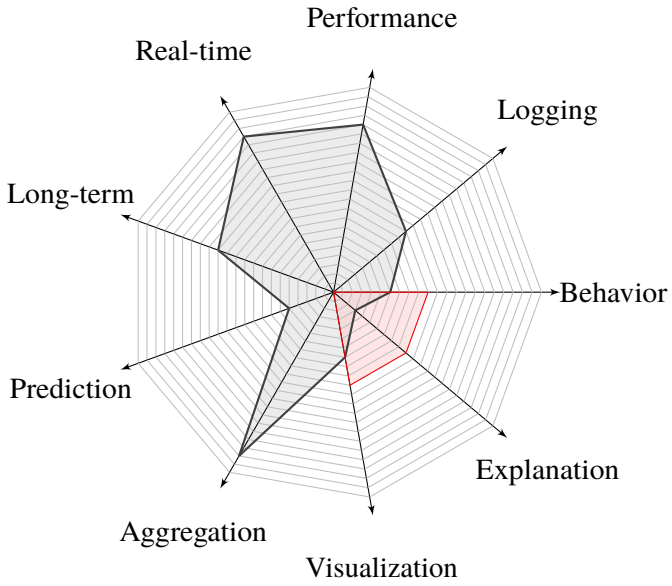

Figure 5.1: Summary of classification results

5.3 Summary

As performance and error analysis is a substantial aspect of operating system development, it is not surprising that there is a wide variety of tools available that offer assistance in that regard. The focus hereby lies on real-time and long-term performance analysis. Message and event log monitoring and evaluation is used to support the process by a subset of the discussed tools. It is also possible to derive insights into the internal behavior of an operating system in order to debug its reasoning process. Nevertheless, this scenario requires detailed knowledge of the operating system in combination with trawling through large amounts of data.

Central concepts are the aggregation and visualization of data to make it easier for humans to evaluate log and performance data efficiently. By supplying comprehensible graphs and charts, or just summaries and averages of relevant metrics, it is possible to gather insights into problems quickly for real-time and long-term perspectives. The set of discussed tools are predominantly text based and only a subset offers built-in visualizations. Nevertheless, the availability of such graphical aids depends on the use-case of each tool. Tools that mainly support long-term analysis, where it is for example important to recognize patterns, usually offer such support.

Prediction support to anticipate problematic scenarios is available in a subset of tools, but limited to a specific set of predefined scenarios. Simulation is not common for operating systems. Software simulation itself is a research topic and might be transferred to the area of operating systems. This transfer is out of scope for this thesis and will therefore not be further discussed.

One shortcoming is the explanation of the internal reasoning process. While there are tools that identify problem scenarios and explain why they are problematic and how to fix them, there are no explanations on the reasoning process that lead to the error, which would potentially help in root cause analysis.

All scores were summed up and are visualized in Figure 5.1.

The presented overview highlights two aspects. First, there is a wide array of helpful tools that deliver support in the analysis process that allow to derive explanations of the operating system behavior. This is of no surprise as performance and error analysis are re-occurring, non-trivial tasks. Second, while the presented tools allow detailed insights into the behavior of the operating system, it is still necessary to have profound knowledge of the internal reasoning processes to gather conclusions for more complex scenarios. This means especially, that the derivation of explanation is in general possible, but remains a non-trivial manual task which could benefit from an increased level of explainability of the internal reasoning process.

This finally concludes the answer to Research Question Q2:

> **Q2**: Which approaches are used in the industry to explain the operating system decision process today?

The following Chapter will propose approaches that can be used to make the internal reasoning process of an operating system more transparent and hence more explainable. These approaches will then be presented to a set of professionals in order to determine the potential usefulness. The results will be discussed in Chapter 7.

5.3.1 Limitations

The presented capabilities of the discussed tools are only covered on a high abstraction layer. A detailed analysis and description is not feasible in the limited time frame of this thesis, due to the extensive amount of features and documentation for the tools. Nevertheless, their core functionalities and concepts are covered in the sections above.

Additionally, there is a large amount of additional tools available to support users in the process of performance and error analysis. These range from commercial tools to community-based ones. Again, due to the limited time frame of this thesis, it is not possible to evaluate a larger number of tools. This will require further efforts in the future. Based on the outcome of the user study in Section 4.3, the covered tools can be considered to be part of the standard tool set used in the industry and provide an initial overview of the state-of-the-art analysis process.

6 Explainable Software Approaches

The previous sections showed how inherently important and at the same time non-trivial it is to analyze performance or error scenarios in the context of such complex pieces of software as operating systems. The following section will first introduce a new way to describe and visualize the internal reasoning process of such a system by using decision graphs in Section 6.1. Subsequently, six additions to the proposed concept will be presented in sections 6.1.4 through 6.1.9 which will allow for more specific insights.

The presented approaches will serve as answers to Research Question Q3:

> **Q3**: Which approaches could be used to increase the explainability of operating system decisions?

6.1 Decision Graphs

The proposed approach will make use of the fact that decision trees are widely considered to be easily interpretable and explainable [Mol20] [CBF+05]. Due to their straight forward structure that aggregates the involved data and their correlation, decision trees offer a human-friendly representation of reasoning processes. The ability to easily create visualization, that can be understood intuitively, is an additional argument which leads to the usage in the proposed approach.

The core concept lies in the idea to allow a user to identify a certain value within the logs of the operating system and to gather information on how and especially why this values was set to this particular value. To do so, a decision graph will be generated that contains information on which constants, other values or objects contributed to the decision.

Because it is well possible that some of the generated structures, especially larger configurations that will be described later, violate tree properties, the more generalized form of a graph will be used in this thesis. Such a decision graph in its basic form, consists of four main parts:

1. **Input**: A set of nodes that represent the input values for the decision.
2. **Input Correlation**: A set of comparison nodes that connect the input values to show why a decision was taken.
3. **Output**: A single node that represents the output value of the decision.
4. **Output Computation**: An addition to the output node that contains the computation description to show how the output of the decision was generated.

An example is shown in Figure 6.1. This particular decision graphs visualizes multiple aspects. First it shows that *valueO* is changed by incrementing its previous value by one. Furthermore it reveals that the values *valueA*, *valueB*, *valueC* and *valueE* as well as a constant and a set are input to this decision. Finally it discloses that the input nodes are correlated in the following way by connecting the input nodes with correlation nodes:

1. *valueA* is strictly greater than *valueB*.
2. *valueC* is equal to a given constant.
3. *valueE* is part of *set1*

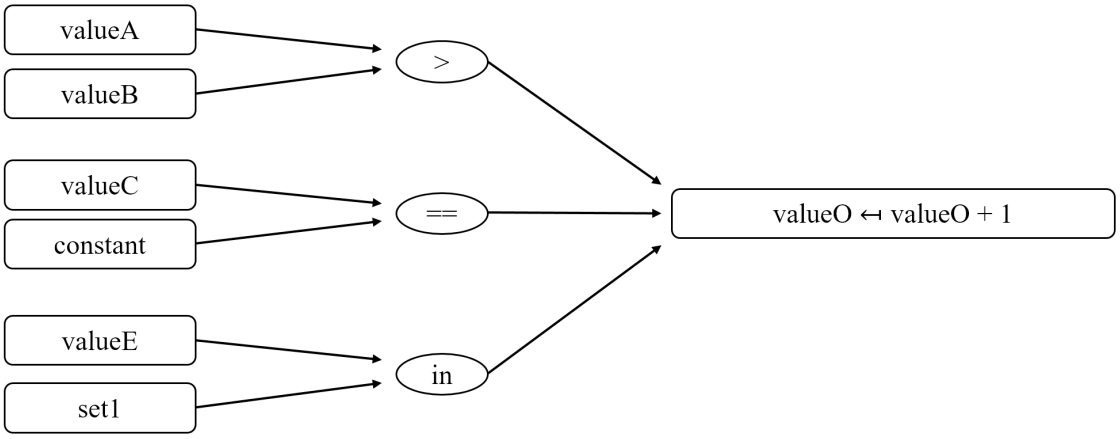

Figure 6.1: A sample decision graph

It shall be noted that the computation instruction is not limited to arithmetical expressions. It is also possible to use operations on objects, e.g. sorting a list, as the outcome of a decision.

This initial approach allows to understand why and how a certain operating decision was taking and allows for a more natural and more intuitive analysis process as it circumvents the research of documentation, design documents or potentially source code. Additions to this approach will be presented in sections 6.1.4 through 6.1.9. These additions will allow to put a single decision into a broader context or deliver more detailed information.

6.1.1 Graph Creation

A core question that arises when considering the proposed approach is how the decision graph can be generated. Storing and persisting such a graph will require large amount of storage to be allocated as some operating system, especially in an commercial environment, might run for years without powering off. Therefore it is impractical to hold the complete decision graph available at all times. Fortunately, operating systems are capable of providing extensive log information which allow to recreate a local decision graph when needed. To successfully create such a graph, five pieces of information have to be available in the log files:

1. **Output value**: Result of the decision

2. **Input values**: Values and their correlation that caused the decision to be made
3. **Computation values**: Values that were used to generate the decision output
4. **Time stamp**: Point in time when the involved values were last changed
5. **Decision identification**: One value can be set by multiple decisions, it is necessary to provide a mean to distinct those

Since this approach is not integrated into any modern operating system, the log files will not contain all necessary information at the current state. It is therefore assumed, that the necessary changes will be made to allow the generation of the decision graph. It shall be noted that very complex scenarios might make it necessary to store large amounts of data in the log files. In this case a trade-off between log file size and a correlating performance drop versus debug and analysis capabilities has to be considered.

Generating the decision graph on the fly comes with more advantages over a fully persisted one. Continuous generation of the graph makes it necessary to implement means that permanently monitor how multiple decisions intersect with one another. This means, it is possible that multiple decisions might depend on the same input values which would make it necessary to identify overlaps in the decision graph in order to properly aggregate nodes that represent the same object. Furthermore, an ongoing graph construction has to handle cycles in order to not create unnecessarily large artifacts. Creating only local sub-graphs, circumvents these problems, as decision that are not relevant for the analyzing person can be ignored, while cycles or loops can be expanded if necessary.

In order to implement an actual generator for decision graphs it is necessary to consider how to take the enriched log files and turn the gathered information into the desired format. Two of the possible options will be outlined briefly without ruling out alternative concepts.

One possible approach is to include the meta information on the decisions into the design of the operating system itself. This can be done by source code annotations, source code scanners, additional property files or modeling software, similar to the XPLAIN framework proposed by Swartout [Swa83]. The advantages here are a close coupling between the actual implementation and the explanations which allows the developers to make sure that the generated explanations match their expectations. Down-sides are the aspect that it is necessary to make sure that source code and annotations for example match at all time which introduces an additional variance when servicing or extending an existing piece of code. Source code generators will require the source code to follow certain conventions which is potentially troublesome as well.

Exploiting machine learning to identify the correlations within the internal reasoning of the operating system is another possible approach. This is a highly non-trivial task and will therefore not be discussed in detail in this thesis. Problems that arise are the generation of sufficient training data as well as the verification of the precision of the generated model. Such an machine learning approach will also be hard to debug and it will be necessary to explain how the model derived a certain explanation which creates a dilemma that requires a complex solution.

Reducing Operating Systems to Graphs

Another aspect worth discussing is the question whether it is possible to reduce the reasoning process of an operating system into a decision graph. The idea behind this assumption is that it is possible to reduce an operating system down to a series of if-clauses. Current computers can be modeled as finite state machines and can therefore also be represented with a Turing machine [HMU01]. A piece of software that can be simulated by a Turing machine is Turing computable. Turing computable software is also WHILE-computable which is equivalent to GOTO-computable [Sch92]. This means that it is possible to reduce an operating system to a set of IF/ELSE and GOTO statements. Finally, this allows to create the decision graph as these transitions can be visualized as a graph in a trivial way.

6.1.2 Selection Criteria for Represented Decisions

It should be obvious that it is not feasible nor useful to map every single if-statement of the source code to a decision graph node. This would lead to an enormous construct where many nodes might not contain critical information. To avoid such a cluttered graph, it will be necessary to select relevant decisions and aggregate the underlying sub-decisions into a meaningful and helpful generalization. The selection and aggregation of decisions to be represented in the final graph pose a design challenge. The following paragraphs will propose a set of criteria that could be used to assist in the process.

Decisions that should be available for analysis are surely those which have immediate consequences that are visible form an external perspective. This means especially such decisions that lead to a change of a metric that can be monitored by a system administrator or a performance analyst for example. In case of an error or an unexpected behavior, an error report will probably reference those values as they are the indicators for the users whether something went wrong in the first place. Therefore, it will be beneficial to be able to easily identify and understand the decision that is responsible for the change of such an externally visible metric.

Another set of decisions that might be of special interest are those that are directly influenced by external inputs, such as a user input or a configuration change. These are particular interesting as they disrupt the system from its current behavior and potentially cause it to transition into an unexpected state. By being enabled to check those influences during the analysis process in an efficient way, it becomes easier to evaluate whether a user interaction had impact on a system that is not performing as intended.

A third set of decisions which can be interesting during the analysis process are those which affect internal variables whose purpose is non-obvious to the user, but nevertheless have a critical role in the overall reasoning process of the operating system. These variables are of particular interest for the developers of the operating system as it might shorten their analysis process significantly. The ability to add these variables and its coming about to the debug data can prevent the need to put a puzzle of other values together in order to derive how such an internal variable might have been set in the error case.

Some decisions on the other hand might not be as interesting during analysis scenario. It would be obviously problematic to completely abandon them per se, but in many case it will be possible to aggregate multiple decisions into a single decision graph. This needs to be done very carefully though in order to make sure that the explanation still matches the actual decision process.

These criteria are of course only suggestions and could be continued. In general it will be necessary though to evaluate the choice whether to select and/or aggregate a decision on a case-by-case basis.

6.1.3 Example

A simple example taken from the *Complete Fair Scheduler* implemented in the Linux Kernel will now be presented to provide an idea on how the proposed approach can be realized by adding source code annotations and evaluate a log file. The code snippet that will be discussed is shown inListing 6.1[1]. It shall be noted that one case will be omitted for simplicity.

```
/*
* Pick the next process, keeping these things in mind, in this order:
* 1) keep things fair between processes/task groups
* 2) pick the "next" process, since someone really wants that to run
* 3) pick the "last" process, for cache locality
* 4) do not run the "skip" process, if something else is available
*/
static struct sched_entity *
pick_next_entity(struct cfs_rq *cfs_rq, struct sched_entity *curr)
{
  struct sched_entity *left = __pick_first_entity(cfs_rq);
  struct sched_entity *se;

  /*
  * If curr is set we have to see if its left of the leftmost entity
  * still in the tree, provided there was anything in the tree at all.
  */
  if (!left || (curr && entity_before(curr, left)))
    left = curr;

  se = left; /* ideally we run the leftmost entity */

  <...> // neglecting one case here for simplicity

  if (cfs_rq->next && wakeup_preempt_entity(cfs_rq->next, left) < 1) {
    /*
    * Someone really wants this to run. If it's not unfair, run it.
    */
    se = cfs_rq->next;
  } else if (cfs_rq->last && wakeup_preempt_entity(cfs_rq->last, left) < 1) {
    /*
    * Prefer last buddy, try to return the CPU to a preempted task.
```

[1] https://github.com/torvalds/linux/blob/3a7956e25e1d7b3c148569e78895e1f3178122a9/kernel/sched/fair.c

```
    */
    se = cfs_rq->last;
  }
}
```

Listing 6.1: selection of next $sched_entity$ in kernel/sched/fair.c

This particular scheduler is using a red-black-tree to determine which process should be granted CPU-time next. As a default, the left most child of this tree is selected for the next scheduling time slice. Two alternative cases will be briefly discussed in the following. First, the currently active run-queue (cfsrq) can explicitly select a process that should be run next. If this selection does not interfere with the overall fairness assumption of the scheduler, this particular process will be scheduled instead of the default one. Second, the active run-queue can specify which process was selected before the current one and thereby indicate that this process should continue with the idea to maintain cache locality, i.e. to be able to reuse cache entries. Again, if this selection does not interfere with the overall fairness assumption of the scheduler, this particular process will be scheduled instead of the default one. The decision whether the fairness assumption of the scheduler is violated or not is taken in the helper function *wakeuppreemptentity* which takes into account the processes run-times and their priorities. The exact proceedings of this function will not be covered here to keep this example short and simple. At first glance, it can be non-obvious to understand why a certain process was chosen in this scenario. If the default expectation to see the left-most entry of the red-black-tree being selected is not fulfilled, it takes an increased level of debugging to collect all necessary data to verify why another process was selected, for example in the ”nextcase described above.

If an immediate explanation on how the next process to be scheduled would be chosen is available, the manual aspect of searching this information can be avoided. To do so, Listing 6.2 shows a possible annotation for this particular decision. The shown annotation is obviously of very basic nature and will need further thought for more complex scenarios. The annotations are:

1. **ID**: Decision identification
2. **out**: Output metric of the decision
3. **cond**: Condition under which the decision is taken
4. **calc**: Calculation rule for the output metric
5. **alt**: alternative calculation rule if the condition is not fulfilled

It shall be noted that only one if-condition will be shown to keep the example as easy to comprehend as possible. Furthermore the comparison of priorities is not fully accurate as there are more complicated penalizing criteria involved that would make the example more elaborate then necessary. In reality, this priority computation is dependent on the already consumed execution-time which is simplified into the simple use of a constant scaling factor in this example. At the same time, this highlights a possible advantage of the explicitly provided explanations as they would save the analyst the manual work to understand the full priority computation. Instead they can immediately get an explanation why a process was chosen over another one. If a more detailed analysis is required, it will become necessary to debug the underlying algorithm in full detail, which would make it unnecessary to have a higher level of explainability in the first place.

```
<...>

//!ID=42
//!out=se
//!cond=cfs_rq->next & (cfs_rq->next->vruntime <= left->vruntime
//!       || cfs_rq->next->load->weight > left->load->weight / scaling_factor)
//!calc=cfs_rq->next
//!alt=left
if (cfs_rq->next && wakeup_preempt_entity(cfs_rq->next, left) < 1) {
  se = cfs_rq->next;
} else {
  se = left
}

<..>
```

Listing 6.2: sample code annotation for simplified code

A sample log file section that allows to generate a decision graph for this particular decision can be found in Table 6.1.

Timestamp	Decision ID	Metric	Result
...	...	...	...
2021-04-10 15:20:48	10	left	some process
2021-04-10 15:20:49	11	cfsrq→next	another process
...	...	...	...
2021-04-10 15:23:48	42	se	cfsrq→next
...	...	...	...

Table 6.1: Sample excerpt from a log file that can be used to create a decision graph

This sample log contains the final decision (with ID 42) on the chosen process to be executed next, in this case the next entry of the current running queue, hence a deviation from the default behavior. In order to generate an explanation on why the observed behavior does not match the expected one, it is assumed that the decisions on the input values are also part of the log file. In this case, the last decisions on the left-most entry of the scheduling tree (ID 10) and the selection of the next process within the running queue (ID 11) are also shown in the sample log. With this information it becomes possible to access all data necessary to generate the decision graph shown in Figure 6.2.

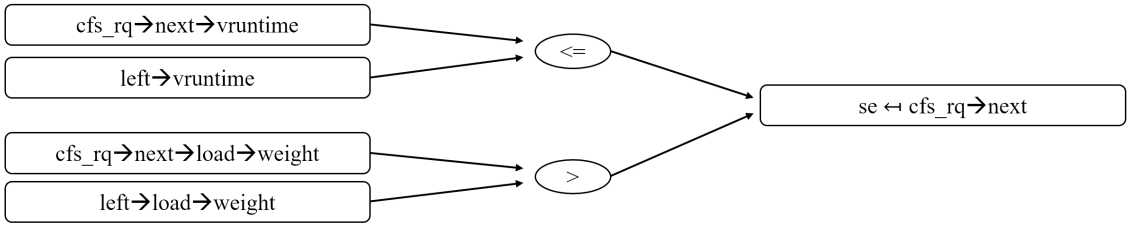

Figure 6.2: Sample graph for the decision on the next scheduled process

This example incorporates the idea of using annotations to generate the necessary code to write all information into the system logs that will be needed to generate the decision graph later on. This idea picks up on the concept of Aspect-Oriented Programming [IKL+97]. It allows the developer to naturally express the aspects of explanation without having to worry about the implementation details that feed the information back into the decision graphs.

This is obviously a simplified scenario, which was chosen to demonstrate the very basic idea behind the decision graphs. The following sections will show additions and enhancements which have the potential to make the concept applicable in more complex scenarios as well.

6.1.4 Browsable History

Analysis of decision can require to evaluate a broader context. It might for example be necessary to not only understand how a certain decision was made, but also how its input values were determined. This can of course be done by generating a set of graphs that all represent a single decision. Having such a collection of figures can become difficult to navigate. Therefore, the first addition to the decision graphs is to make them browsable, i.e. it is possible to expand a single graph by appending the inputs, correlations and computation information for a previous decision.

An example is shown in Figure 6.3. It picks up the graphs from Figure 6.1 and shows an expansion for *valueA*. This allows for the insight that *valueA* is set to the constant value of 2, if *valueX* is strictly greater than *valueY*. The green trapezoid indicates that further expansions are of course possible. This holds for all shown input values.

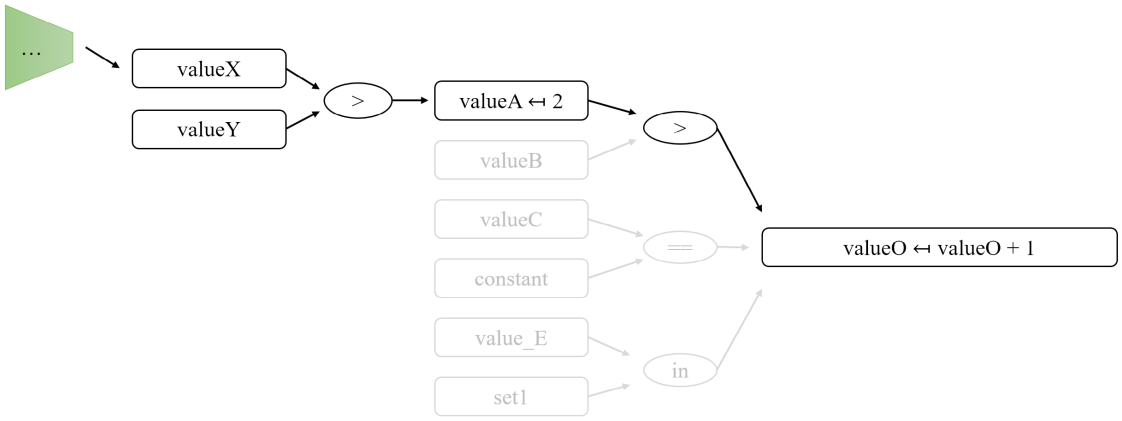

Figure 6.3: Decision graph with expanded history

This visualization allows to put a single decision into a larger context if necessary. In order to avoid a cluttered image that becomes incomprehensible it will be essential to use techniques that keep the graph well arranged. Expanding a graph to a large degree will obviously result in a large number of nodes and edges which need to be displayed in a way that it is still possible to focus on the actual analysis without being overwhelmed.

6.1.5 Component Affiliation

The scope of decision making is not limited to the complete operating system. Within the system, a multitude of sub-components can be responsible for the change of a certain value. Potentially this can even be done via user input or for example middleware software. During analysis, it is relevant to know which party sets a value in order to determine where to continue with a more detailed research or to which team a problem needs to be re-routed.

An example is shown in Figure 6.4. In this graph, *valueO* is incremented by one if *valueA* is equal to *valueB*. Additionally, the color coding of the nodes provides information about the component that set the values. In this case *valueO* is set by component *green*. The input value *valueB* was also set by component *green* while *valueB* was set by component *blue*. If further analysis draws the conclusion that *valueA* contains an unexpected value, it becomes obvious that further analysis needs to be done for component *blue*.

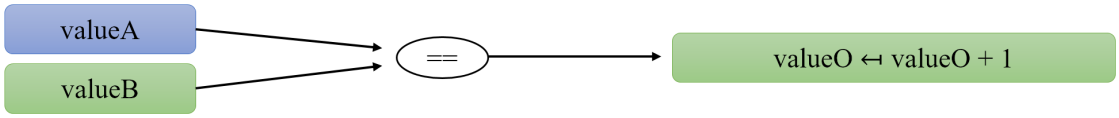

Figure 6.4: Decision graph with component affiliation highlighting

The distinction on a component level allows for a quicker identification of further analysis targets. It furthermore allows to potentially gather a better overview on the interaction of multiple operating system components. This can be especially helpful if someone with less experience wants to get an initial overview of the internal reasoning process. The easy identification of user input, by a user or other software, also allows to distinguish between an operating system issue and external ones.

In order to provide this kind of insight, the decision identification must either include an identifier for the component or must be unique across all sub-components.

6.1.6 Natural Language Explanations

A decision graph on its own will not allow a person who is not familiar with the specific topic to fully understand the internal reasoning process in all cases. Decisions might require conceptual knowledge of the involved component or a certain decision might be non-intuitive due to a flawed software design. In these cases, additional information and context can be appended to a decision in form of natural language.

An example is shown in Figure 6.5. The decision graph in focus reveals that *valueO* is incremented by one if *valueC* is equal to a certain constant. The reason why this constant was chosen for the comparison remains non-obvious. To tackle this ambiguity an explanation in natural language is provided that explains the selection of the particular constant.

Explanations in natural language offer a convenient way to provide additional information to a particular decision within the larger reasoning process. A challenging aspect of this addition lies in the problem that drafting precise and concise documentation is non-trivial. Maintaining the consistency between source code and the natural language string also poses a risk.

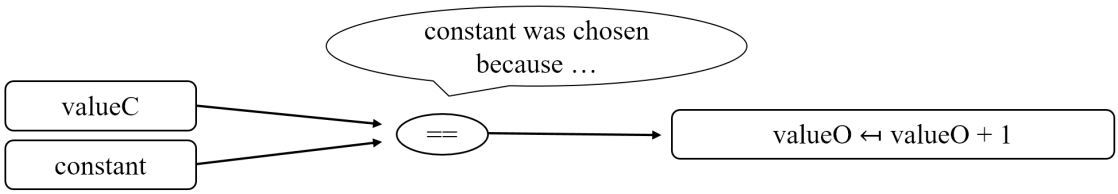

Figure 6.5: Decision graph with explanation in natural language

6.1.7 Evaluation of Alternatives

Knowing the actual sequence and outcome of decisions allows to understand and analyze error scenarios or performance issues. Additionally, insights into potential alternative decisions can be helpful to understand where and how a system can be tweaked to reach a potentially more desired behavior or how to increase confidence that unwanted decision are taken. To do so, the decision graphs can be extended to highlight different outcomes if a correlation node would return an alternate result. This alternatives can be seen as a mapping to an if-else construct within the source code.

An example is shown in Figure 6.5 which matches Algorithm 6.1. This particular graph represents a decision where *valueO* is incremented by one if *valueA* is greater or equal to a given constant. The alternative is highlighted in red and shows that if *valueA* is strictly less than the constant, *valueO* gets decremented by one.

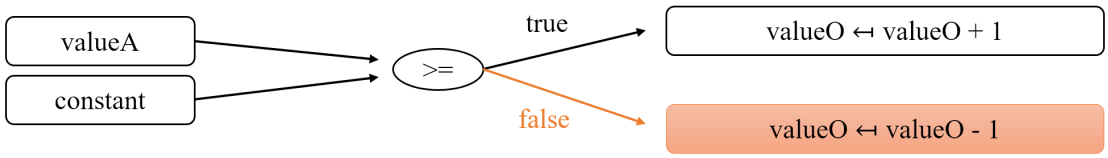

Figure 6.6: Decision graph that shows potential alternatives

Algorithm 6.1 Algorithm that creates alternate outcomes depening on *valueA*

```
if valueA ≥ constant then
    valueO ← valueO + 1
else
    valueO ← valueO − 1
end if
```

By being enabled to evaluate alternative outcomes of a decision the analyzing person has the chance to better understand the internal reasoning process of the operating system. Knowing how the system would behave if a certain values would be changed, allows to get an intuition how the system will react if it encounters a different scenario. Additionally, this concept allows to identify whether a decision can lead to a potentially better outcome when fed with the according input values or whether a slight change on the input side might cause an outcome that would lead to an error. While being a useful addition, it will be non-straightforward to always provide alternative outcomes. Problems will arise if no else-clause is available for example or if the else-clause is of a much higher complexity and does not immediately lead to a decision.

6.1.8 Impact Analysis

Another aspect of interest during problem analysis is the determination whether a given variable contributes to the decision under investigation. This can be achieved by iteratively expanding the decision graph until a previous decision makes use of the variable in question.

An example is shown in Figure 6.7. This graph highlights that the variable to be investigated *valueQ*, has an impact on the decision to increment *valueO* by one. The impact can be traced back to a certain path within the graph. In this case, *valueQ* had influence on the decision how *valueA* was set. The dotted node in between signals that there are potentially more nodes involved. The length of the path can be used as a metric to determine the impact of *valueQ* on the decision on *valueO*.

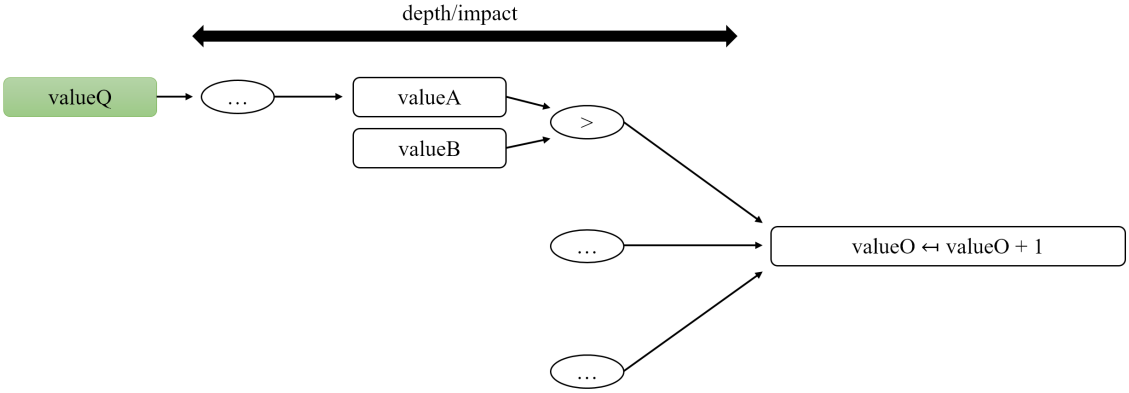

Figure 6.7: Decision graph that shows the impact of *valueQ* on the given decision

Having the opportunity to query whether a specific value has influence on a decision allows to narrow down the set of variables that need to be considered during analysis. Values with high impact are good candidates for more detailed research, while such with little influence can be neglected initially.

The definition of impact is unfortunately non-trivial. The length of the path back to the queried value can be an initial estimate. Decision that were taken far in the past, might have less consequences for the more recent ones. This can not be guaranteed though. A critical decision in the past, might have brought the system in to a non-expected state which it could not recover from, which might not allow the system to behave as intended. This in turn would pose the question if the impact is defined based on if the decision under investigation had the observed result or whether the decision could be reached in the first place.

Another case that needs to be considered is the possibility that the queried value has no influence on the current decision. In this situation, it would be necessary to either abort the search at a certain depth or to generate the graph up to the moment the operating system was started. The latter can come with high resource and time consumption and is therefore to be used carefully.

6.1.9 Backtracking

The final addition to decision graphs presented in this thesis is a concept that aims to assist in the identification of root-causes. If an initial analysis finds an value that does not match the expectations it is necessary to identify when the operating system left the expected path within the decision graph. This can be of help if there is a specific decision within the graph that leads to a branch into one of two (or more) sub-graphs. One of these sub-graphs leads to the observed value and does not contain any path to the expected value. In this case the identified decision, that creates such a deviation, can be seen as a good starting point for a root-cause analysis.

An example is shown in Figure 6.8. In this scenario *valueO* was observed to have the value in the *blue* box. The expected behavior, given the scenario, would be the value in the *green* one. By exploring the decision graph, it is determined that *valueD* had a critical impact on the decision path. The decision based on *valueD* let to a branch within the graph that would no longer allow to reach the *green* box.

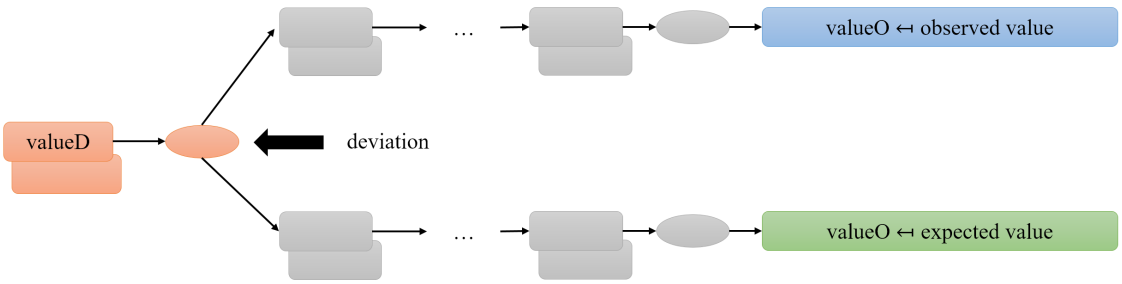

Figure 6.8: Decision graph that shows where a deviation occurred

Having such a capability can give indications on where to continue with a root-cause analysis. It is not a tool that miraculously solves analysis problems though. On the one hand, it is still necessary to understand why the observed value is not the one that would be expected. At the same time, it is also necessary to have an idea on what that expected value would be. This knowledge still requires a good intuition of the internal reasoning process of the operating system. On the other hand, this approach comes with a significant implementation complexity as it will be potentially necessary to generate large decision graphs.

It shall be noted that the backtracking approach, as well as the impact analysis approach, have similarities with the concept of program slicing [Wei84]. The common idea is the attempt to reduce the program or set of programs to those steps necessary to end up in the observed scenario. Just as in program slicing, the goal of backtracking and impact analysis are the identification of entities that contributed to a certain event. While program slicing focuses on the instructions, the approaches proposed in this thesis work on a slightly higher abstraction level, the decisions.

6.1.10 Computability and Complexity

Up to Figure 6.6, the proposed approaches only rely on the necessary data to be available within the log files of the operating system. Creating the according visualizations remains a trivial technical task.

Figure 6.7 and Figure 6.8 on the other hand are more challenging as they require to be able to explore the decision graph on a potentially larger scale. If the graph is completely generated, the approaches reduce to the task to find a path from the decision under investigation to another decision, which maps to the *PATH* problem [Sip96] which is proven to be solvable in an efficient manner, e.g. by using Dijkstra's algorithm [Dij+59]. Restriction do of course apply if the path does not exist or can only be found by evaluating a giant graph. In these cases, a trade-off needs to be considered between the run-time required to find the given path and the usefulness for the analysis process. If a path cannot be found within a defined distance from the decision under review, it can be argued that the path has no relevance in the specific scenario.

This local restriction of the graph generation must also be considered when generating a graph for these approaches. Creating the full decision graph can easily yield a time and resource consuming task, especially if the operating system runs in a large scale setup and has not been rebooted for a long period of time. It must also be considered to create the graph in an iterative manner, i.e. to explore one further level of decision notes at a time and verify whether the target node has been found. There is a high chance that further optimizations can be done to make the graph generation more efficient, but evaluating this topic would go beyond the scope of this thesis.

It shall be noted, that the backtracking approach implies a form of simulation of the operating system. As this is a very complex issue, this technique might only work in a very localized context. Finding root cause decision becomes significantly harder with each step that has to be additionally simulated. The complexity in this task arises in the challenge to anticipate the consequences of the taken decisions for other components.

6.2 Application of Approaches

The following section will discuss how the presented approaches can be applied to the scenarios gathered in Section 4.4. To do so, the essential problem behind the given issues will be briefly reiterated each, followed by proposals which decision graph concepts can help in the analysis process. Furthermore limitations and remaining manual tasks will be given. The application details will remain at a very high level in order to convey an intuition of the potential usage scenario without having to go into great technical depth.

Scenario 1 - Unexpected Internal Adjustments

In the first problem case, the communication to an external device did not perform as expected due to a wrongly set bit in an data structure. Determining that said bit was the cause for the unexpected behavior is a non trivial task as the critical impact of the bit is non-obvious.

A decision graph, generated for the decision on how to run the communication, can shorten this analysis step, as it would be immediately observable, that the reasoning process depends on the bit. To allow that such a graph can be generated it need to be ensured that the assignment of the bit as well as its influence on the communication decision need to be available. Identifying which decision needs to be analyzed remains a manual task though. Mapping the miscommunication with

the device to the correct value/location in the log file, that can then be used for the decision graph analysis also remains a non-trivial task. This step can be potentially supported by existing tools, e.g. PFA, which can identify patterns and anomalies in the log files.

Determining the decision that initiated the miscommunication and deriving that the mentioned bit is responsible represents only a subset of the necessary analysis process. In addition to explaining the symptom, it is also needed to identify the root-cause of the problem, i.e. which component set the bit to the unexpected value and why. To do so, enhancing the decision graph with component affiliation details allows to quickly identify where the most recent change of the bit occurred. This information can then be used for further analysis

Once the decision is identified that changes the bit to the unexpected value, an explanation in natural language, together with the information about input, correlation and computation, can be of help to understand why the bit is set the way it is observed. Ideally, it is obvious why the bit was set based on the decision graph. Nevertheless, if it is the case that the bit is set in a non-intuitive way, additional explanations can save the analyst from researching more documentation or even the source code.

Scenario 2 - Unexpected Hardware Interactions

The second scenario was caused by a change on the side of an external hardware device. The operating system adapted in a way it is designed to. As a consequence, the performance of the system was impacted negatively, which was not obvious to foresee.

In order to better understand the potential impacts of such changes to the system, two of the previously described approaches can help to increase the comprehensibility of the operating system. On the one hand, being able to put decisions into context by investigating the history of preceding ones allows to get better insights into the internal reasoning processes without requiring to step through the source code. On the other hand, being able to gather information on how the system would potentially behave if certain values were to be changed, i.e. having the ability to query for alternative decisions, can increase the confidence in implementing such changes as the consequences would be more predictable.

Finally, the backtracking approach can be used to identify the point in time and cause of the deviation from the expected behavior. This can point the analyst into the direction to investigate the change on the external device side in order to evaluate whether this change was necessary or how the performance consequences could have been circumvented.

Scenario 3 - Unexpected Use of Resources

Finally the third scenario occurred in a highly virtualized environment where workload is competing for and sharing a common set of resources. The problems found during analysis were network issues and an unexpected scheduling of workload onto a specific unit due to an unknown defaulting mechanism.

The network problems are not under control of the operating system and are hence also not necessarily observable in the decisions taken. This highlights a limitation of the proposed approach as there is still a remaining set of tasks to be done manually or with the help of other tools. Namely, identifying

unexpected metrics in the log data. Once such a metric is found, the proposed approach can be used to understand why the metric was set to the observed value and which path of decision lead up to it.

The unexpected dispatching of workload on the other hand, can be mapped to a decision within the operating system. When the decision is spotted, further analysis can begin at this point. Apart from simply following the decisions and understanding the internal reasoning process, impact analysis can bring some value here. At first glance, it is necessary to evaluate whether a problem is caused by an user error. By checking whether a user input, e.g. performance goals, has impacted the given decision, the analyzing person can get an intuition whether the user has an influence on the operating system behavior at this point. In this specific case, it would be possible to identify that a user input is included in the reasoning process but the user did not specify a value. In the following steps, the assignment of this input can be investigated and it can be seen that a defaulting mechanism sets the value.

6.2.1 Summary

The proposed approach aims to show how explainable software can be of help during problem analysis in an operating system context. The high level concept relies on representing the internal reasoning process by using a decision graph, which allows to easily follow and comprehend decisions taken by the system. By adding a variety of potential enhancements the capabilities and explanatory power of the concept is augmented in a way that allows for more detailed and specialized analysis efforts.

Finally, the application of the proposed approaches in Section 6.2 shows that these concepts are a valid answers to Research Question Q3:

> **Q3**: Which approaches could be used to increase the explainability of operating system decisions?

It shall be noted that technical feasibility is out of scope of this thesis. Due to the fact that it is necessary to change the internals of an operating system in a significant way, implementing this approach will pose a major effort. Nevertheless, it has been shown, that the proposed solutions can be implemented if the resource investment is taken.

As the previous sections are merely a proposal, it is necessary to verify that these approaches are appealing to professionals who can potentially profit from them. Therefore, the conceptual approach itself will be verified in the following evaluation in Chapter 7.

7 Evaluation

This chapter will build on the approaches presented in Chapter 6 and aims to evaluate whether these concepts are useful in a professional environment and if so to which degree. To do so, a user study will be presented in the following sections which shows the perception of actual professionals who were confronted with the mentioned approaches. First the goals of the studies will be described in Section 7.1. The design of the study, including its format and choice of participants, will then be shown in Section 7.2. Lastly, the results will be presented and discussed in Section 7.3.

This chapter will then finally conclude with a statement to the following evaluation goal:

> **EG**: Evaluate whether an increased explainability of the internal reasoning process would be a valid enhancement for operating systems.

7.1 Study Goals

As mentioned before, the overall goal of this study is the evaluation whether the presented approach in Chapter 6 is considered to be of value for professionals. To draw conclusions on this question, a set of actual professionals will be interviewed. Three evaluation questions will be addressed.

EQ1 Do the presented approaches bring advantages for the interviewed professionals in their daily work?

EQ2 Do the presented approaches allow the professionals to derive explanations on the operating system behavior more easily?

EQ3 Do the presented approaches allow the professionals to derive complete explanations on the operating system behavior?

The first evaluation question aims to evaluate the practical use of the decision graph concept to determine whether this approach can be of general value in the industry or might just remain a research topic. The second evaluation question then targets whether the proposed approach allows to speed up the mostly manual analysis processes gather in the first user study in Section 4.3. Finally, the third evaluation question challenges the thought whether the new concept can be used to completely explain the operating system behavior and which aspects will require further research.

7.2 Study Design

This section will provide the overall study design. The structure will be identical to Section 4.2 in first presenting the format of the study, then the rationale on the participant choice and finally a discussion on the questions asked.

7.2.1 Format

The format of the conducted study is very similar to the first study shown in Section 4.2. During a personal interview with a single professional, a set of questions is presented which acts as a guideline throughout the conversation to make sure, that all intended points are covered in the end. The answers are recorded together with the participants in written form. The interviews were scheduled to be 30 minutes long and held online[1]. The set of questions is on the one hand designed to explicitly query structured answers of the proposed approaches but on the other hand also aims to encourage a more detailed reflection to generate non-structured feedback. The gathered structured data is supposed to provide a metric that allows concrete claims on the usefulness and potential value and capabilities of the decision graph concept. The non-structured answers allow a more diverse insight into more specific aspects that are received particularly well while also allowing the interviewees to openly state potential blind spots.

Other potential formats were dismissed for the same reasons as discussed in Section 4.2. Just as in the first user study, this second one aims to gather qualitative data over quantitative ones.

7.2.2 Participants

The set of interviewed professionals is taken from the same background setting as in the first study described in Section 4.2. Apart from one individual, all other interviewees were not participating in the first study to ensure the coverage of a broader spectrum. Unfortunately, it was not possible to arrange interviews with the groups of system administrators and support professionals due to an unexpected spike of workload on their end. As the results of the first study showed that the analysis approaches across all job roles are similar, this negligence of the other groups in this study is not to be seen as a major issue.

7.2.3 Questions

A brief set of two background questions was chosen to ensure that the interviewed people work in the desired field and are confronted with problem analysis task in their daily jobs while also giving the opportunity for an easy start of the conversation. The full set of questions and the introduction for the participants can be found in the appendix in Section 9.3.

Questions 1 through 7.2, shown in Table 7.1, cover the approaches shown inChapter 6. The simple nature of Yes/No questions aims to gather intuitive reactions and encourage the interviewees to think and talk about the concepts. The split into the potential use cases of actual problem analysis and the additional field of understanding/learning the reasoning process of the operating system was chosen to further stimulate the interviewees to provide feedback. The purpose of these questions is to engage the interviewees during the presentation of the approaches. A scale ranging from 0 to 5 was chosen. Generating no additional value at all maps to a score of 0, while a score of 5 relates to a potentially significant improvement in the daily work of the interviewees.

[1] due to the COVID-19 pandemic

X.1	Would the described approach be helpful during problem analysis?
X.2	Would the described approach help you to better understand the decision process of the operating system?

Table 7.1: Questionnaire questions 1 through 7.2

Questions 8.1 through 8.9 then aim to collect more detailed feedback while also leaving the interviewees with the task to consider the analysis and learning aspect at the same time. First, question 8.1 challenges the visual representation of the approaches in order to gather information on potential issues that need to be addressed in the future. Questions 8.2 and 8.3 check if the approach can be used to fully understand or generate complete explanations for the operating system. These questions particularly aim to engage the interviewees to think critically about the proposed approaches and name aspects and areas that cannot be covered. Furthermore questions 8.4 and 8.5 are designed to engage the interviewees to compare the respective approaches and distinguish their usefulness. Finally, questions 8.8 and 8.9 gather the relevant distinction between the approaches on a more explicit level by challenging the interviewees to provide an actual rating. Questions 8.x are shown in Table 7.2.

8.1	Do you find the visualizations helpful?
8.2	Would you consider that the described approaches provide enough detail for you to understand the reasoning process of the operating system?
8.3	Would you consider that the described approaches allows to generate complete explanations for the reasoning process of the operating system?
8.4	Which aspects would you consider to be most valuable during problem analysis?
8.5	Which aspects would you consider to be essential to understand the reasoning process of the operating system?
8.6	How useful would the described base approach in Question 1 on its own for problem analysis?
8.7	How useful would the described base approach in Question 1 on its own to understand the internal reasoning process of an operating system?
8.8	How useful are the additional described approaches from Question 2 through Question 7 during problem analysis?
8.9	How useful are the additional described approaches from Question 2 through Question 7 to understand the internal reasoning process of an operating system?

Table 7.2: Questionnaire questions 8.1 through 8.9

7.3 Study Results

The following paragraphs will present and discuss the answers obtained by the user study. First an overview of the gathered feedback is shown in Section 7.3.1. The derived advantages and disadvantages, according to the interviewees will be described in Section 7.3.2. Enhancements proposed by the participants will also be presented in Section 7.3.2. In Section 7.3.3, the scores for each approach will be aggregated to form a value assessment. Finally, Section 7.3.4 discusses

the perception of the participants on the ability to generate explanations based on the proposed approaches. Transcripts of the answers provided by the interviewees can be found in the appendix in Section 9.3.

7.3.1 Overall Reception

The proposed approach received good reception. Most aspects were rated as potentially helpful and sparked ideas for possible usage scenarios among the interviewees. It proved to be effective to ask the participants to look at the approaches from two perspectives, the problem analysis and the learning aspect. This encouraged them to spend more thought on their rating and understanding of the approaches. This was especially helpful as the concept was initially hard to grasp. The gradual introduction of different aspects and variations allowed the interviewees to form a sufficient understanding throughout the ongoing interview.

7.3.2 Advantages, Disadvantages and Enhancements

The following paragraphs will first discuss the aspects perceived positive by the interviewees. Subsequently, mentioned shortcomings will be presented as well as enhancements proposed by the participants.

Advantages

One set of advantages mentioned by the interviewees revolves around the potential to save time during the analysis process. One aspect contributing to possible time savings is the chance to gather a better overview on a higher level in the debugging process. By being able to ignore source code at first, it is possible to follow the reasoning process at first and avoid a more detailed and hence more time consuming analysis on source code level. Furthermore, the ability to quickly get insights whether certain variables have an impact on the decision in question is seen to be a chance to bootstrap the analysis process by allowing a simple prioritization of values that need further analysis. Gathering these priorities is otherwise a question of experience in the area as in general, a large set of values might have had an impact. An additional chance to accelerate the analysis process lies in the opportunity to identify the component which took the critical decision more efficiently and therefore avoiding a lengthy analysis.

Another set of advantages emerged in the area of improved learning chances. One positive aspect is the created ability to verify and build intuition on the internal reasoning process of the operating system. This means especially, that by looking at a certain scenario and picking a metric of interest, it is easy to verify if one knows how the given value is set. At the same time, during problem analysis it is possible to quickly check whether the current assumptions on the decision process that the analyst has are correct. Especially less experienced professionals are assumed to benefit from the proposed approaches. If done properly and with the correct scope, the attachment of context and further explanations in natural language is deemed to be of value in this case. Furthermore, the proposed approaches increase the transparency of the operating system in general which results in better predictability and a reduced overhead and skepticism when analyzing a less familiar sub-component for example.

Disadvantages

While the overall concept of the proposed approaches was well received in general, some aspects leave room for improvement according to the interviewees. One study participant raises concern about the representation of more complicated problems. Visualization of non-trivial and complex decision steps will need to be addressed in the future as a suitable visual representation will be necessary to not overwhelm the analyst with output. Another participant formulated the prerequisite that it is still necessary to first manually identify the decision which caused an unexpected scenario. This hold especially true in more abstract analysis scenarios, e.g. if a system missed its performance goals. Additional software or manual analysis will be necessary to determine the decision. Another potentially problematic aspect are external influence that cause the operating system to take non-optimal decisions. One example for this case is a malfunctioning network device which would require to identify that an external input was wrongly calculated. One interviewee emphasizes the described non-trivial problem to define the impact metric in the impact analysis approach. Finally, a participant mentions that the proposed approaches might not be suitable for well experienced analysts as they might be able to immediately pinpoint what they need to look for. An additional interface to navigate the decisions of the operating system might be a burden to them.

Enhancements

In addition to advantages and disadvantages, two interviewees also propose potential additions to the general approach.

The first addition mentioned by an interviewee is the combination of displaying the decision graph alongside the operating system source code. The idea behind this proposal is to allow a quicker navigation and orientation through the code, which could create speedup in the development process as well.

The second idea explicitly focuses on the natural language explanations. The explanations should not only be available on the decisions themselves but also on the input metrics per decision. This point was stressed by one interviewee as the understanding of the decision alone can be worthless if the input values are unclear. Such explanations are not explicitly excluded by the already proposed approaches, but also not clearly mentioned. It shall therefore be noted that this aspect needs to be additionally mentioned when considering such explanations in natural language in the future.

7.3.3 Value Assessments

The decision graph concept was rated very well in general. On average, the interviewees considered it to be a very helpful addition to the already existing tool set, as shown by the score of 4.4 out of 5 for problem analysis and 4.6 out of 5 for an overall learning goal.

The more specialized additions to the approach got more differentiated scores as shown in Table 7.3. This is due to the fact, that the interviewees showed very different viewpoints upon the topic as the variety of advantages and disadvantages show in the previous section. Nevertheless, some trends and patterns can be derived.

The backtracking approach is deemed as particularly helpful by all interviewees for problem analysis while being rated significantly low for learning efforts. The opposite yields for impact analysis, with it being seen very useful for learning, but less useful for problem analysis. The browsable history approach and the component affiliation representation is received good ratings for both application scenarios while natural language explanations and the display of alternatives were the least popular concepts.

Approach	Analysis	Learning
Decision Graph	4.4	4.6
Natural Language Explanations	2.4	3.6
Browsable history	4.2	4.2
Component affiliation	3.6	4
Alternatives	2.6	3.2
Impact Analysis	2.8	4.4
Backtracking	4.8	2,4

Table 7.3: Results for study Questions 8.6 through 8.9

These scores show, that the professionals prefer to be able to understand given data by being able to browse it and highlight the involved parties. These two approaches focus in particular on the explanation aspect of the proposed approach, i.e. to be able to explain which component changed a metric and which values were used to take this decision.

If the goal is defined as to learn how the operating system works, the focus expands to other approaches which also allow to gather further context and information while also evaluating alternatives. This indicates that an additional support of prediction or simulation techniques would bring value to the professionals.

The strong support for the backtracking concept in problem analysis scenarios shows the immense interest in finding the root cause of a problem as quickly as possible. This is obviously not surprising but shows the potential value in further research in this area which combines explainability with simulation capabilities.

7.3.4 Explanations

A more abstract scope of the proposed approaches is the general capability to derive explanations on the internal reasoning process of the operating system. The interviewees mostly agreed that the approach can help to derive explanations more easily and more efficiently. The proposed approaches on their own are deemed to be not sufficient on their own, i.e. they are not perceived as a stand-alone solution that allows a complete problem analysis.

The complexity of the operating system will still require further analysis tools to aggregate information in complex and more abstract scenarios, like performance analysis. Identifying the metric that causes the unexpected behavior remains non-trivial because the symptom might not correlate to a specific metric that can be found in the log files. Once the problematic value is found, the approach helps to explain why the problem occurred.

An important distinction mentioned by the interviewees are the scopes of the derivable explanations. Aspects that can be explained are those that cover why the operating system what it did. Circumstances that cannot be fully explained by the proposed approaches are more abstract ones that cannot be immediately linked to a specific value but require to analyze a combination of issues at once.

Another potential shortcoming is mentioned in the area of integrating external influences into the explanations. As soon as a problem is caused by an malfunctioning external device for example, it is no longer possible to derive complete explanations with the proposed approaches. This means especially that only problems that fully reside within the scope of the operating system itself can be explained using the proposed approaches. Valid external input on the other hand is acceptable.

One additional point is that while it is possible how the operating system takes a decision and which metrics are involved, it remains not obvious why the operating system was implemented this way. To address this issue, it would be necessary to ensure that the natural language explanations are maintained properly, such that it is possible to gain insights into the decision process. Gathering this information by algorithmic means is usually not possible.

7.4 Threats to Validity

The threats to validity of this second study are almost identical to those of the first study as described in Section 4.4.1. The main critical aspects are the small sample size of 5 interviewees and the lacking experience of the interviewee.

7.5 Summary

This final contribution chapter concludes with a statement on how the conducted user study contributes to the fulfillment of the evaluation goal:

> **EG**: Evaluate whether an increased explainability of the internal reasoning process would be a valid enhancement for operating systems.

The presented study showed that approaches which aim to increased the explainability of the internal reasoning process of an operating system are considered to be helpful by the interviewed professionals. To do so the three initially mentioned evaluation questions were answered.

The first Evaluation Question EQ1 was answered by the shown value assessment of the approaches. The highlighting of advantages and disadvantages underlines the high potential in saving time which also answers Evaluation Question EQ2. With regard to Evaluation Question EQ3, the participants were mostly skeptical about capabilities to generate complete explanations, but stated that further tool support will remain necessary to do so.

In summary, all study participants agreed that the presented approaches in particular would bring additional value for the problem or performance analysis process. The study also showed that while the decision graph would be a valid enhancement of the operating system, it cannot be seen as a complete replacement of the existing tools. Hence providing an overall positive answer statement with regard to the evaluation goal.

8 Conclusions

This chapter will summarize the overall conclusions of this thesis followed by a brief outlook on further research opportunities.

8.1 Summary

The overall research goal of this thesis was to

> **RG**: Evaluate how the explainability of operating system decisions can be increased and how this improves the analysis process.

To do so, four research questions were answered. The answers to those questions will be summarized in the following paragraphs, followed by a conclusion how those answers contribute to the research goal.

The first research question was answered in Chapter 4 by evaluating the actual analysis of professionals in a user study. This study highlighted two things. One, generating explanations for the operating system reasoning process pose a highly non-trivial and time consuming task. Second, there is a multitude of scenarios in which these explanations are necessary. Especially, it is required to gather explanations on very specific decisions of the operating system. A total number of three scenario types were presented, namely, unexpected adjustments of internal data structures, unexpected interactions with peripheral hardware and an unexpected use of internal resources, which were used to answer Research Question Q1.

> **Q1**: Which scenarios exist in the area of operating system that require explanation?

The second research question was answered in Chapter 5 by presenting tools that are used by professionals on a regular basis and discussing to which degree they support the generation of explanations or which other capabilities they offer during the analysis process. It was shown, that the gathered tools offer a large variety of support for the analyzing professional. The main focus lies on the aggregation of real-time and long-term performance data. This requires the analyzing professional to have a profound background knowledge of the operating system internals in order to interpret and understand why the operating system behaves in a certain way. This means especially, that the generation of explanations is a very manual process that requires the analyst to manually connect data points within the log data, concluding the answer to Research Question Q2.

> **Q2**: Which approaches are used in the industry to explain the operating system decision process today?

The third research question was answered in Chapter 6 by introducing the concept of decision graphs. These visual representations of the internal reasoning process of the operating system allow to integrate the highly interpretable nature of decision trees into the problem analysis process. By being able to follow each change of relevant metrics in an intuitive and comprehensible way, the generation of explanations becomes easier and more time-efficient. A variety of possible extensions and use-cases was presented alongside an example out of the complete fair scheduler used in the Linux kernel to show how the level of explainaibility can be increased by the proposed approach, hence answering Research Question Q3:

> **Q3**: Which approaches could be used to increase the explainability of operating system decisions?

A final evaluation of the proposed approach was shown in Chapter 7 by presenting the concept of decision graphs to a set of professionals and collecting their feedback in a second user study. This study highlighted, that the proposed approach would actually improve the explainability of the operating system while also demonstrating that such an increased explainability would be an actual improvement for the analysis process of the interviewed professionals. This result was then used to highlight the contribution to the evaluation goal:

> **EG**: Evaluate whether an increased explainability of the internal reasoning process would be a valid enhancement for operating systems.

In conclusion, the answers to the research questions show that an increased explainability of operating system decisions would be of value for professionals. It was shown, that there are scenarios that require the explanation of specific operating system decisions. It was also shown, that existing tools and approaches allow to analyze such scenarios but still require a significant amount of background in the specifics of the internal reasoning process of the operating system. Therefore, an approach was presented that visualizes the decision process in a comprehensible way in order to increases the explainability. Finally it was shown, that such an approach is perceived very well by professionals who have to conduct problem analysis regularly.

8.2 Outlook

This thesis has shown that there is value in increasing the explainability of the internal reasoning process of operating systems. Further research can be done in multiple directions, some of which will be described in the following paragraphs.

First, it should be considered to redo the conducted user studies with a larger number of participants. This would bring two advantages. One, it would allow to confirm the results of this thesis on a broader scale. Second, it would potentially identify further aspects of interest in the area or additional enhancements for the decision graph concept proposed in this thesis. Another direction for further investigation is the design of the proposed code annotations and the potential implementations to enhance the existing log files of operating system with the necessary data to generate the decision information. Another implementation aspect that can be considered as future work is the implementation of the graph generation. Challenges here are the efficiency and complexity of the necessary algorithms.

Another aspects worth further investigating is the adjustment of the annotations proposed in Section 6.1.1 to align more properly with the concepts of Aspect-Oriented Programming. This would allow for a better usability and potentially enriched feature set to create more sophisticated representations of explanations. One example for such an enrichment could be to re-think the visualization of input values. It might for example be possible, that a set of time-based data points is used as input to a decision, which could be represented as a graph. Depending on the content of the input data, it might also be possible to make use of visualizations like box plots or histograms. Finding examples for such decisions and evaluating whether those would bring additional value is also a possible starting point for future work.

9 Appendix

9.1 Study of the Problem Analysis Process in the Industry

The following pages show the content of the questionnaire used for the user study presented in Chapter 4. The intent of the questionnaire that was given to the participants is also shown. Finally, the transcripts of the conducted interviews are shown.

Intent of this questionnaire

The answers to this questionnaire will be used as input to a Master's thesis which aims to investigate the potential value of combining the two computer science areas of *operating systems* and *explainable software*. *Explainable software* is a relatively new branch of research and attempts to investigate how software can make its internal reasoning process more transparent and more comprehensible for its users.

This is especially interesting if the software in focus is non-trivial. A common target of explainable software is the area of machine learning, because the internal processes there are often of black-box nature. Operating systems have usually grown over decades and have similarly black-boxy reasoning processes. The thesis, which will use the results of this survey, aims to evaluate the potential impact of adding approaches of the explainable software field to the design step of operating systems, to make them a core principle during development.

The answers to this questionnaire will provide insight into the overall analysis problems in the operating system environment and will assist in determining if there is a potential value for additional integrated explainability in operating systems.

Sections

The following sections are part of this survey

- **General**: Background of the interviewee
- **Problem analysis**: Time consumption, error types and strategies
- **Scenarios**: Approaches used by the interviewee

General

1.1 What is your current primary job role?

- Operating system developer
- System administrator/programmer
- Performance analyst
- Other

1.2 How frequently do you work with the following operating systems in your current job role?

Note: This questions aims at the operating systems which you are maintaining/developing/configuring in your primary job role. Please do not refer to the operating systems on your local machine that you use to log into the remote machines which you are working with.

Rate the following from 0 to 5, with 0 meaning *not working with the operating system at all* and 5 meaning *working with the operating system on a daily basis*:

- Linux: 0 - 5
- Linux on Z: 0 - 5
- IBM z/OS: 0 - 5
- IBM z/VM: 0 - 5
- Others: 0 - 5

1.3 How much time do you spend working with the following operating systems in your current job role?

Note: This questions aims at the operating systems which you are maintaining/developing/configuring in your primary job role. Please do not refer to the operating systems on your local machine that you use to log into the remote machines which you are working with.

Rate the following from 0 to 5, with 0 meaning *spending no time with the operating system* and 5 meaning *all available time is spent with the operating system*:

- Linux: 0 - 5
- Linux on Z: 0 - 5
- IBM z/OS: 0 - 5
- IBM z/VM: 0 - 5
- Others: 0 - 5

1.4 What level of operating system knowledge does it take to do your current job properly?

Note: This questions aims at the operating systems which you are maintaining/developing/configuring in your primary job role. Please do not refer to the operating systems on your local machine that you use to log into the remote machines which you are working with. You can use the following mapping as an orientation:

- 0: not working with the operating system.
- 1: application skills, e.g. I can use tools, but have no knowledge on how to set them up.
- 2: setup skills, e.g. I know how to set up tools or middleware like a tomcat server or a messaging system.
- 3: configuration skills, e.g. I know how to set up workload priorities or network configurations.
- 4: analysis skills, e.g. I know how to identify performance bottlenecks or how to identify errors.
- 5: debugging skills, e.g. I understand the internal reasoning process for an operating system component.

- Linux: 0 - 5
- Linux on Z: 0 - 5
- IBM z/OS: 0 - 5
- IBM z/VM: 0 - 5
- Others: 0 - 5

Problem analysis

2.1 How much of your work time do you spend on average analyzing problems per week?

Please make sure, to have a sum of less or equal to 100% in the end.

- Linux: 0% - 100%
- Linux on Z: 0% - 100%
- IBM z/OS: 0% - 100%
- IBM z/VM: 0% - 100%
- Others: 0% - 100%

2.2 How much time do you usually spend on an average problem?

- Linux: minutes / hours / days / weeks / months
- Linux on Z: minutes / hours / days / weeks / months
- IBM z/OS: minutes / hours / days / weeks / months
- IBM z/VM: minutes / hours / days / weeks / months
- Others: minutes / hours / days / weeks / months

2.3 How much time do you usually spend on a complex problem?

- Linux: minutes / hours / days / weeks / months
- Linux on Z: minutes / hours / days / weeks / months
- IBM z/OS: minutes / hours / days / weeks / months
- IBM z/VM: minutes / hours / days / weeks / months
- Others: minutes / hours / days / weeks / months

2.4 How common are the following reasons to be the cause of the problems you have to analyze?

Rate the following from 0 to 5, with 0 meaning *not occurring at all* and 5 meaning *occurring multiple times a day*:

- User error : minutes / hours / days / weeks / months
- Operating system setup/configuration error : minutes / hours / days / weeks / months
- Operating system bug / unexpected behavior: minutes / hours / days / weeks / months
- Unexpected workload / external reasons: minutes / hours / days / weeks / months
- Others: minutes / hours / days / weeks / months

2.5 How much time does a problem analysis take on average given the error cause?

You may just leave blanks for error types which you do not encounter in your job role.

- User error : minutes / hours / days / weeks / months
- Operating system setup/configuration error : minutes / hours / days / weeks / months
- Operating system bug / unexpected behavior: minutes / hours / days / weeks / months
- Unexpected workload / external reasons: minutes / hours / days / weeks / months
- Others: minutes / hours / days / weeks / months

2.6 How much time do the following factors consume during your average problem analysis?

Rate the following from 0 to 5, with 0 meaning *no big impact on time consumption* and 5 meaning *major time-consuming aspect of problem analysis*:

- Actual analysis: 0 - 5
- Setting up an environment to recreate scenario: 0 - 5
- Waiting for responses from experts to assist in analysis: 0 - 5
- Waiting for feedback from user who reported the error: 0 - 5
- Others: 0 - 5

2.7 Which strategies and tools do you usually use to analyze problems?

Proposed options:

- General strategies
  - Log/trace analysis
  - Debugging source code
  - Recreate error scenario
- IBM z/OS specific tools
  - IBM Z Decision Support
  - IBM Resource Measurement Facility (RMF)
  - IBM z/OS Workload Interaction Correlator (zWIC) and IBM z/OS Workload Interaction Navigator (zWIN)
- Linux specific tools
  - top
  - dtrace
  - perf
- Other tools and strategies (you may specify multiple entries here)
  - Homegrown, selfwritten tools (please provide brief description of their purpose)
  - Other publicly available tools

Scenarios

NOTE: Please make sure to avoid the use of any potentially confidential or personal information.

The following scenarios attempt to recreate the situation in which a user encounters an unexpected message by the operating system and the analysis process which aims to identify the cause of that message. The responses here will be used to identify actual scenarios and strategies for problem analysis in the area of operating systems. The results will then be used to validate existing tools that aim to provide insights into the reasoning process of the operating system, as well as to evaluate the potential of approaches from the field of explainable software.

If you have not encountered one of the described scenarios, you may omit them. The final section *Your experience* will allow you to specify a scenario that you would deem to be typical in your daily work.

Please highlight any specific tools that you would use in the scenarios.

Scenario 1 - Is it a user error?

Assume the following scenario: A user encountered an error message and is reporting to you that he/she does not know how to proceed. Furthermore, the user insists that he/she has not done anything wrong. What are your first steps to determine whether the user has done something wrong or it is an actual problem on the operating system side? Which tools do you use? Please describe the steps and tools briefly:

- free text field

Scenario 2 - The user is innocent!

Assume the following continuation of scenario 1: You were not able to find any wrongdoing on the user side. What are your next steps to find out if the system has potentially been wrongly setup and identify which configuration might be in error? Which tools do you use? Please describe the steps and tools briefly:

- free text field

Scenario 3 - Something's fishy!

Assume the following continuation of scenario 2: You were not able to find any flaws in the configuration of the system. What are your next steps to find out if this is an actual operating system bug and to understand what went wrong? Which tools do you use? Please describe the steps and tools briefly:

- free text field

Your experience

Have you encountered a similar scenario recently in your daily? If so, please describe what happened briefly:

- free text field

9.1.1 Transcript: Interviewee 1

NOTE: Interviewee stated that Linux and Linux on Z are the same from his/her perspective. Therefore only Linux will be used in the given answers.

Question	Answer
1.1	Performance analyst
1.2	Linux(5) ; IBM z/VM(2)
1.3	Linux(4) ; IBM z/VM(1)
1.4	Linux(3 → performance measurements , 4 and 5 → error scenarios) ; z/VM is not measured nor analyzed in error cases, the latter is done by another team
2.1	Linux(80%), interviewee mentioned that meta problems might occur that origins from interactions between multiple operating systems
2.2	Linux(days)
2.3	Linux(weeks)
2.4	Setup(1) ; Bug(1) Comment: Most performance measurements run without problems. Rare cases with errors, but those are hard to analyze
2.5	Setup(days) ; Bug(weeks)
2.6	Analysis(5) ; Environment(2) ; Experts(3) ; User(1)
2.7	Strategies: Logs, Debug, Recreation Linux tools: top, dtrace (less relevant) Other tools: sysstat (contributions to open source made when necessary)
3.1	get access to system, get benchmarks, not really relevant
3.2	gather log data
3.3	recreate scenario, perf, sysstat, requires a lot of experience, identify team in charge for failing component
3.4	multiple workers run on the same OS, all process the same workload, one node is significantly slower. Cause were network issues but also unexpected scheduling of infrastructure workload onto that particular node due to a default behavior.

9.1.2 Transcript: Interviewee 2

Question	Answer
1.1	Operating system developer
1.2	IBM z/OS(5) ; IBM z/VM(4) ; Linux on Z(1)
1.3	IBM z/OS(5) ; IBM z/VM(1)
1.4	IBM z/OS(4 and 5)
2.1	IBM z/OS(20%)
2.2	IBM z/OS(hours)
2.3	IBM z/OS(days)
2.4	User error(2) ; Setup(1) ; Bug(3); External(1) Comment: Unexpected Workload often caused by tests that contain unrealistic scenarios.
2.5	User error(hours) ; Setup(days) ; Bug(hours); External(days) Comment: Bugs are often pre-analyzed by Support Team
2.6	Analysis(2) ; Environment(2) ; Experts(3) ; User(4) Comment: Delays if experts in other timezone, Delay due to long connection chain to user
2.7	Strategies: Logs, Debug, Recreation Homegrown tools: REXX post-processor to filter data ; Excel spreadsheets to visualize data (e.g. identify peaks) Other tools: IBM IPCS
3.1	request memory dump, recreate on arbitrary system
3.2	request long-term logs/data (SMF), investigate decision trace (only contains decision, explanations only implicit)
3.3	request test system to recreate actual user environment, very rare, steps from 3.2 usually sufficient
3.4	Abends and endless loops, Support Team usually identifies error type and location

9.1.3 Transcript: Interviewee 3

Question	Answer
1.1	System administrator/programmer (mainly development systems, not much production)
1.2	IBM z/OS(5) ; IBM z/VM(4) ; Linux on Z(2); IBM z/VSE(2)
1.3	IBM z/OS(4) ; IBM z/VM(3) ; Linux on Z(2); IBM z/VSE(2)
1.4	IBM z/OS(4) ; IBM z/VM(2) ; Linux on Z(2); IBM z/VSE(2)
2.1	IBM z/OS(50%) ; IBM z/VM(10%)
2.2	IBM z/OS(days) ; IBM z/VM(hours)
2.3	IBM z/OS(weeks) ; IBM z/VM(hours)
2.4	User error(2) ; Setup(3) ; Bug(1) ; External(1) ; Hardware problems(2) ; Network problems(2)
2.5	User error(minutes) ; Setup(hours) ; Bug(days) ; External(minutes) ; Hardware problems(days) ; Network problems(hours)
2.6	Analysis(3) ; Environment(3) ; Experts(2) ; User(1) ; Approvals/Access permissions(1) ; Time difference(1)
2.7	Strategies: Logs,Recreation IBM z/OS tools: IBM RMF Homegrown tools: REXX post-processor to filter and aggregate data Other tools (all IBM): Health Checker, Omegamon, Candle, HMC, SDSF, IPCS
3.1	request scenario details and logs (SDSF, HMC, LOGRECs), verify error description
3.2	collect and analyze detailed error data (system dump and traces)
3.3	in case of abend/exception/signal processing, identify failing module and forward to development team
3.4	interfaces to network/hardware issues (hard to evaluate because scope is outside of operating system itself, can the operating system detect anomalies though?); unclear autonomous adjustments of operating system on hardware and network changes; real-time vs. long-term analysis; tons of messages → which are relevant for a specific scenario, filtering depending on scenario; pro-active signals if something is about to go wrong, if done manually, this requires a lot of manpower

9.1.4 Transcript: Interviewee 4

Question	Answer
1.1	Support professional
1.2	IBM z/OS(5)
1.3	IBM z/OS(5)
1.4	IBM z/OS(5) ; IBM z/VM(1)
2.1	IBM z/OS(90%)
2.2	IBM z/OS(hours)
2.3	IBM z/OS(days)
2.4	User error(3) ; Setup(4) ; Bug(3) ; External(2)
2.5	User error(minutes) ; Setup(hours) ; Bug(days) ; External(minutes)
2.6	Analysis(5) ; Environment(2) ; Experts(1) ; User(3)
2.7	Strategies: Logs,Debug,Recreation IBM z/OS tools: IBM RMF Homegrown tools: REXX post-processor to filter and aggregate data, test programs Other tools (all IBM): HMC, SDSF, IPCS, PFA, MFA, runtime diagnostics
3.1	theoretical research to verify what is supposed to happen in the described scenario
3.2	which component is wrong, search known problems
3.3	analyze system dumps and traces
3.4	specific bit was overridden by an operating system component, no trace/indication who did it and why; peripheral hardware signals a change on its side and operating system reacts to it in an unforeseen way

NOTE: Interviewee stated that for questions 3.x, that the partitioning into the given subscenarios does not necessarily apply to him/her. A user error can as well be determined in very late stages of the analysis process.

9.1.5 Transcript: Interviewee 5

Question	Answer
1.1	Operating system developer
1.2	IBM z/OS(5) ; Linux(4) ; IBM z/VM(2)
1.3	IBM z/OS(3) ; Linux(3) ; IBM z/VM(2)
1.4	IBM z/OS(1,3,4,5) ; Linux(1,2,3,4) ; IBM z/VM(1,5)
2.1	IBM z/OS(40%) ; IBM z/VM(10%)
2.2	IBM z/OS(days) ; Linux(minutes)
2.3	IBM z/OS(weeks) ; Linux(days)
2.4	User error(1) ; Setup(2) ; Bug(2)
2.5	User error(minutes) ; Setup(hours) ; Bug(days)
2.6	Analysis(4) ; Environment(5) ; Experts(1) ; User(3)
2.7	Strategies: Logs,Debug,Recreation IBM z/OS tools: IBM RMF Homegrown tools: MS Excel Chart generator for visualization; formatter for core data areas of operating systems Other tools: IBM IPCS
3.1	identify last steps, ask for dump and SMF data, verify user's claims
3.2	check things that are known to cause problems (e.g. configs and definitions), identify precise location of error
3.3	analyse source code and compare with traces, recreate environment if necessary
3.4	CPUs not utilized although available → a special combination of configuration settings caused an unexpected program path. User did not follow recommendations, but improvised.

9.1.6 Transcript: Interviewee 6

Question	Answer
1.1	Performance analyst
1.2	IBM z/OS(5)
1.3	IBM z/OS(5)
1.4	IBM z/OS(1,2,5)
2.1	IBM z/OS(60%)
2.2	IBM z/OS(days)
2.3	IBM z/OS(weeks)
2.4	User error(5) ; Setup(3) ; Bug(1) ; External(1)
2.5	User error(hours) ; Setup(hours) ; Bug(days) ; External(weeks)
2.6	Analysis(3) ; Environment(2) ; Experts(4) ; User(3) ; Identify where to start analysis(5)
2.7	Strategies: Logs,Recreation IBM z/OS tools: IBM RMF Homegrown tools: SMF processing → fetch, aggregate visualize
3.1	understand normal/expected behavior, understand deviation, analyze user setup and debug data → pin-point location/cause of deviation
3.2	request more data for more detailed analysis, identify component in error, recreate scenario, recreate environment (iterative process because a full recreation at once is usually not necessary/not feasible)
3.3	forward to development teams
3.4	Hardware migration/upgrade performance expectancy vs. reality → hard to explain potential discrepancies Workload executes on unexpected nodes/CPUs → hard to analyze why not executed on expected resources General issue with predictability if something (configuration, workload, etc.) is changed

9.2 Tool Classification Scores

The following tables show the scores for the tools discussed in Chapter 5.

Category	Comment	Score
Performance	basic information on resource usage and active process properties	2 / 3
Behavior	no insights into actions taken by the system	0 / 3
Logging	no access to logs and events	0 / 3
Real-time	continuous display and refresh of performance metrics	3 / 3
Long-term	load averages allow rough estimate on an overall performance trend	1 / 3
Prediction	none	0 / 3
Aggregation	core metrics across processes	3 / 3
Visualization	negligible	0 / 3
Explanation	none	0 / 3

Table 9.1: Classification for top

Category	Comment	Score
Performance	very detailed information on resource usage and active process properties	3 / 3
Behavior	no insights into actions taken by the system	0 / 3
Logging	access to basic system logs	2 / 3
Real-time	continuous collection of data	3 / 3
Long-term	only accessible logs contain historic data	1 / 3
Prediction	none	0 / 3
Aggregation	detailed metrics across processes	3 / 3
Visualization	none	0 / 3
Explanation	none	0 / 3

Table 9.2: Classification for SDSF

Category	Comment	Score
Performance	very detailed information on resource usage and process performance	3 / 3
Behavior	no insights into actions taken by the system	0 / 3
Logging	no log access	0 / 3
Real-time	continuous collection of data	3 / 3
Long-term	only accessible logs contain historic data	3 / 3
Prediction	none	0 / 3
Aggregation	detailed metrics across processes	3 / 3
Visualization	data exportable, large amount of software available that generates graphs and charts	2 / 3
Explanation	none	0 / 3

Table 9.3: Classification for sysstat

Category	Comment	Score
Performance	very detailed collection of performance metrics and system state	3 / 3
Behavior	insights available, but have to be derived manually	1 / 3
Logging	no access to logs	0 / 3
Real-time	continuously updated display of current performance data	3 / 3
Long-term	continuous collection and persistence of extensive performance data	3 / 3
Prediction	none	0 / 3
Aggregation	core metrics across processes	3 / 3
Visualization	pre-configured graphical reports available + xml reports for custom analysis	2 / 3
Explanation	none	0 / 3

Table 9.4: Classification for RMF

Category	Comment	Score
Performance	no progress measurable, but possible to identify how the system performed recently	1 / 3
Behavior	insights available, but have to be derived manually	2 / 3
Logging	full access to system and component logs	3 / 3
Real-time	only the very moment when snapshot taken	1 / 3
Long-term	analysis via log info	2 / 3
Prediction	none	0 / 3
Aggregation	overlays, filters and short-cuts available	3 / 3
Visualization	none	0 / 3
Explanation	none	1 / 3

Table 9.5: Classification for IPCS

Category	Comment	Score
Performance	not the main scope of the tool, conclusions can be drawen based on the occurence of checks	1 / 3
Behavior	not the main scope of the tool, conclusions can be drawn based on the occurrence of checks	1 / 3
Logging	allows to monitor and react on the content of messages and operating system event	3 / 3
Real-time	warn about urgent, potentially critical issues	2 / 3
Long-term	none	0 / 3
Prediction	potentially critical situations can be anticipated	2 / 3
Aggregation	checks base on aggregated data, not possible to further investigate it	1 / 3
Visualization	none	0 / 3
Explanation	info why check hit, how to fix/prevent, no explicit insights into reasoning	1 / 3

Table 9.6: Classification for IBM Health Checker for z/OS

Category	Comment	Score
Performance	predefined scenarios can be analyzed	2 / 3
Behavior	debugging for predefined scenarios possible	2 / 3
Logging	system log and its messages are being monitored	2 / 3
Real-time	notifications if system is in critical state	1 / 3
Long-term	not within the main scope	0 / 3
Prediction	predicts the occurrence of predefined potentially critical scenarios	2 / 3
Aggregation	aggregates data to display predefined messages	1 / 3
Visualization	none	0 / 3
Explanation	explanations are given on the nature of a problem and how to fix it, not the reasoning process itself	1 / 3

Table 9.7: Classification for PFA and Runtime Diagnostics

Category	Comment	Score
Performance	very detailed collection of performance metrics and system state	3 / 3
Behavior	no insights into actions taken by the system	0 / 3
Logging	no immediate access	0 / 3
Real-time	continuous refresh of data	3 / 3
Long-term	continuous collection of data	3 / 3
Prediction	calculation of future capacity requirements	1 / 3
Aggregation	detailed metrics across processes	3 / 3
Visualization	pre-configured graphical reports available	3 / 3
Explanation	none	0 / 3

Table 9.8: Classification for IBM OMEGAMON

9.3 Evaluation

The following pages show the content of the questionnaire used for the evaluations study presented in Chapter 7. The intent of the questionnaire that was given to the participants is also shown. Finally, the transcripts of the conducted interviews are shown.

Intent of this questionnaire

This questionnaire aims to evaluate if a set of proposed approaches can increase the explainability of the internal reasoning process of operating systems. Furthermore, it will be explore whether these approaches would be considered to be beneficial during the problem analysis process in the operating system context. To do so, a set of approaches will be presented and then rated on their helpfulness.

Operating systems are able to collect and report an extensive amount of data that allows analysis of their current state and how they performed in the past. This analysis is facilitated by tools that are either integrated into the operating system itself or by additional tools that evaluated the gathered performance and debug data. Interpreting this information to gain insights into the reasoning process of the operating system is a non-trivial task that requires either a significant amount of time or years of experience. The presented approaches aim to reduce this overhead, thereby, making it easier to analyze and understand decisions taken by the operating system.

It shall be noted, that the technical feasibility will not be evaluated by this questionnaire. It will be assumed that these approaches are already available. An actual implementation will require potentially significant changes within in the operating system itself or a presumably complex external modeling of its behavior.

Background

What is your current primary job role?

- Operating system developer
- System administrator/programmer
- Performance analyst
- Other

How much of your work time do you spend on average analyzing operating system problems?

Note: This question explicitly excludes problems that concern your workstation device or default user id and setup task. Please restrict to problem analysis on the component you are developing, administrating or analyzing the performance of.

- 0% to 100%

Scenario

Assume that it is necessary to analyze a complex problem. After some research, it turns out that a particular system metric, $value_O$ is identified which has been set to a non-expected or invalid value. An example for this would be the increase or decrease of the priority of a certain process. The approaches presented in the following sections aim to assist in the task to understand and explain why this value was set to the observed value. It is further assumed that the component to be analyzed provides and persists the necessary data in some form, e.g. sysstat on Linux or IBM SMF on IBM z/OS.

Decision graphs

The internal reasoning process of the operating system is converted into a large decision graph[1].

The following graph is generated when the user requests an explanation why $observedvalue$ has been assigned to $value_O$ at a specific point in time. It is not necessary to consult the source code. This information can be gathered by just referencing the metric in question and the point in time in a to-be-defined user interface, e.g. by selecting the value in sysstat on Linux or IBM RMF on IBM z/OS.

Each circular node represents an operator that returns a boolean result. Each squared node represents a decision where a value was assigned to a given metric or a constant. It might be necessary to aggregate multiple metrics into a intermediate variable to accurately represent the internal reasoning process. This maps to the use of internal variables within the source code.

It shows that the following set of conditions were met, which lead to the final decision:

1. $value_A$ was greater than $value_B$
2. $value_C$ was equal to a certain constant
3. $value_E$ was part of set_1

Additional similar boolean expressions can also be considered to be available.

1.1 Would the described approach help you to better understand the decision process of the operating system?

- Yes / No

[1] In contrast to a decision tree, it might be possible to have multiple paths lead to the same decision, hence violating tree properties.

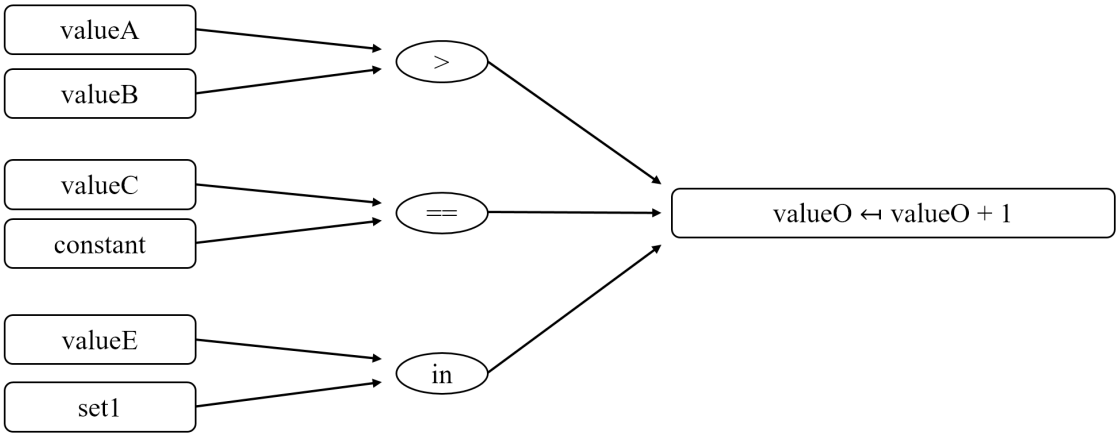

Figure 9.1: A sample decision graph

Browsable history

As each squared decision node represents an assignment of a metric, each non-constant input to a decision is also the output of a previous decision. The graph in Figure 9.2 shows how a browsable version of the decision graph can look like. In this example, which extends the on in Figure 9.1, the decision graph was extended to understand and explain how $value_A$ was assigned to the observed value. In this case, this assignment was done because $value_X$ was greater than $value_Y$.

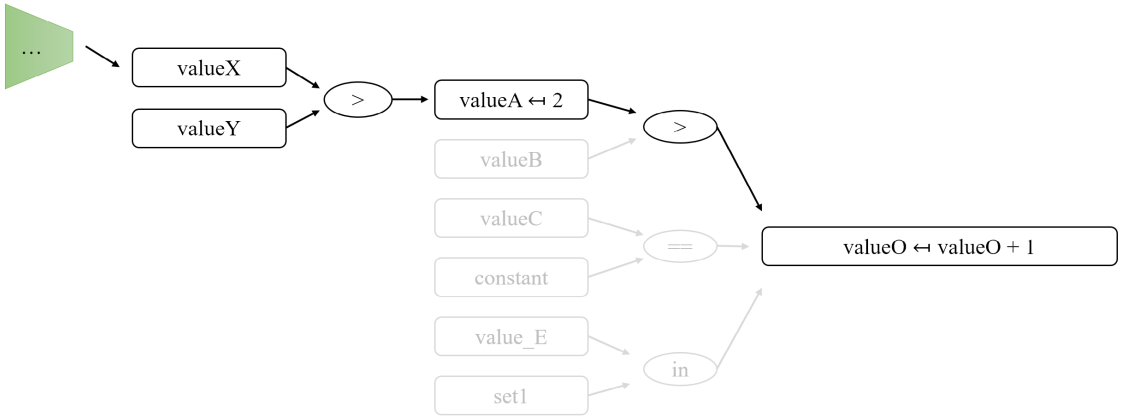

Figure 9.2: A decision graph that can be expanded to also show previous decisions

2.1 Would an extendable decision graph be helpful during problem analysis?

- Yes / No

2.2 Would the described approach help you to better understand the decision process of the operating system?

- Yes / No

Involved components

In some scenarios, it might be the case, that some decisions are made by different components. The approach shown in Figure 9.3 highlights the idea to provide information whether a certain value was set by another component. In the example, one component (green) is currently executing. The decision to set $value_O$ depends on $value_A$ and $value_B$. While $value_B$ has been set by the component itself, $value_A$ has been set by another component (blue).

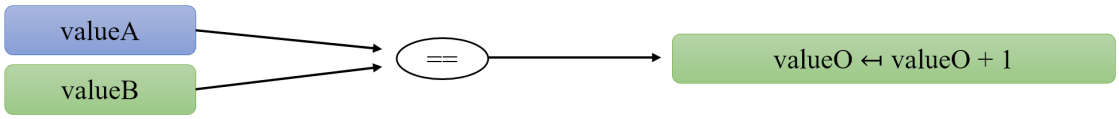

Figure 9.3: A decision graph that shows which component has set the input values.

3.1 Would an overview of involved components be helpful during problem analysis?

- Yes / No

3.2 Would the described approach help you to better understand the decision process of the operating system?

- Yes / No

Explanations in natural language

In addition to the decision graph described above, it is also possible to obtain explanations of the decision in natural language. This explanation could contain further information on the context of the decision and provide reasons why the comparison of two values has been implemented as shown in the graph. In source code this would map to an explanatory comment.

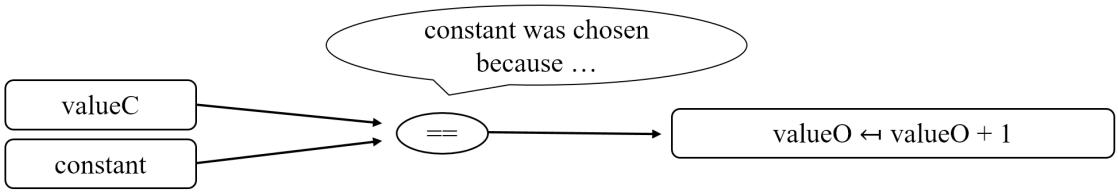

Figure 9.4: A decision graph that includes an explanation in natural language

4.1 Would natural language explanations be helpful during problem analysis?

- Yes / No

4.2 Would the described approach help you to better understand the decision process of the operating system?

- Yes / No

Show alternatives

The following approach adds the ability to analyze alternative results. It is possible to negate the output of one or multiple boolean operations. This allows to understand how the operating system would behave if a certain aspect in the environment would be different. The example below shows that $value_O$ would be set to $alternatevalue$ instead of $observedvalue$ if $value_A$ would not be equal to the given constant.

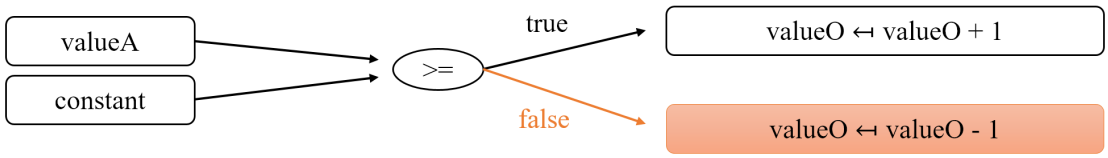

Figure 9.5: A decision graph that shows an alternative outcome for a given decision

5.1 Would an evaluation of alternatives be helpful during problem analysis?

- Yes / No

5.2 Would the described approach help you to better understand the decision process of the operating system?

- Yes / No

Impact analysis

Additionally, consider the possibility to identify whether a certain metric has influenced the decision which is currently under observation. The example in Figure 9.6 highlights a query which requests information on how the assignment of $observedvalue$ to $value_O$ is influenced by the metric of $value_Q$. The depth of the graph to be analyzed might be one indicator how relevant the value of $value_Q$ was for the decision on $value_O$. A metric that has appeared in the recent decision path is more likely to have had a significant impact on the current decision than one that has been taken

further in the past. Unfortunately, there is no guarantee for that as a critical decision in the past might have decided between execution in disjoint sub-graphs. Therefore, the described impact can only be taken as an indication unless additional metrics are applied that identify such critical decisions.

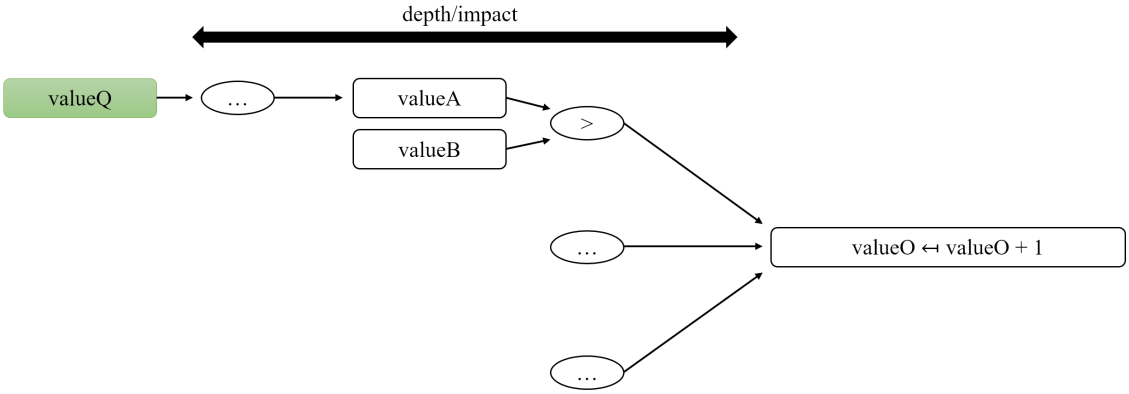

Figure 9.6: A decision graph that shows the impact of another metric to a given decision

6.1 Would it be helpful during problem analysis to gain insights whether a certain metric influenced a decision?

- Yes / No

6.2 Would the described approach help you to better understand the decision process of the operating system?

- Yes / No

Backtracking

Finally, the following approach will handle the question where the operating system deviated from the expected path. This means, $observedvalue$ was reported to be set for $value_O$ but $expectedvalue$ would be the correct behavior. Figure 9.7 shows an example where the decision graph is generated up to the point where the graph branches into a sub-graph from which it is no longer possible to reach the expected value. The value that would have need to be set differently is $value_D$. Further analysis can commence at this decision to identify why $value_D$ was set in the presented way.

This view can obviously only be generated if it is possible to reach the expected value within the decision graph. It is also necessary to consider time and resource boundaries, as the worst case would be to backtrack within the decision graph up to the root node, where the system was started.

7.1 Would backtracking capabilities be helpful during problem analysis?

- Yes / No

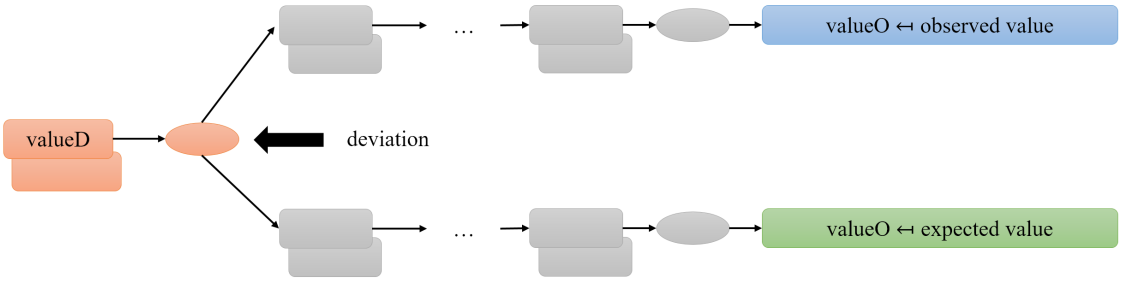

Figure 9.7: A decision graph that shows an alternative outcome for a given decision

7.2 Would the described approach help you to better understand the decision process of the operating system?

- Yes / No

General

This section contains several questions that aim to collect general feedback across the proposed approaches.

8.1 Do you find the visualizations helpful?

- Yes / No

8.2 Would you consider that the described approaches provide enough detail for you to understand the reasoning process of the operating system?

- Yes / No

8.3 Would you consider that the described approaches allows to generate complete explanations for the reasoning process of the operating system?

- Yes / No

8.4 Which aspects would you consider to be most valuable during problem analysis?

Select one or multiple:

- Natural language explanations
- Browsable history
- Involved components
- Show alternatives
- Impact analysis
- Backtracking

8.5 Which aspects would you consider to be essential to understand the reasoning process of the operating system?

Select one or multiple:

- Natural language explanations
- Browsable history
- Involved components
- Show alternatives
- Impact analysis
- Backtracking

8.6 How useful would the described base approach on its own for problem analysis?

Rate the following from 0 to 5, with 0 meaning *not useful* and 5 meaning *significant improvement for problem analysis*:

- Decision graphs: 0 - 5

8.7 How useful would the described base approach in on its own to understand the internal reasoning process of an operating system?

Rate the following from 0 to 5, with 0 meaning *not useful* and 5 meaning *significant help*:

- Decision graphs: 0 - 5

8.8 How useful are the additional described approaches during problem analysis?

Rate the following from 0 to 5, with 0 meaning *not useful* and 5 meaning *significant improvement for problem analysis*:

- Natural language explanations: 0 - 5
- Browsable history: 0 - 5
- Involved components: 0 - 5
- Show alternatives: 0 - 5
- Impact analysis: 0 - 5
- Backtracking: 0 - 5

8.9 How useful are the additional described approaches to understand the internal reasoning process of an operating system?

Rate the following from 0 to 5, with 0 meaning *not useful* and 5 meaning *significant help*:

- Natural language explanations: 0 - 5
- Browsable history: 0 - 5
- Involved components: 0 - 5
- Show alternatives: 0 - 5
- Impact analysis: 0 - 5
- Backtracking: 0 - 5

9.3.1 Transcript: Interviewee 1

Question	Answer
B1	Performance analyst
B2	70%
1.1	Yes
2.1	Yes, but only sometimes, often need for very specific information instead of broad history context
2.2	Yes, when trying to understand, everything is interesting
3.1	Yes
3.2	Yes, identify where things are changed, identify correlations between components
4.1	No, at least for experienced professionals, this might not be necessary
4.2	Yes, if no or little experience
5.1	Yes, for a fast analysis, for more details probably need to check code, not helpful for performance bottlenecks, more for debugging
5.2	Yes
6.1	Yes, good to filter initial set of other metrics to look at
6.2	Yes, useful to gather an intuition
7.1	Yes
7.2	No
8.1	Yes
8.2	Yes
8.3	Yes
8.4	History, components, backtracking
8.5	Natural language, components, impacts
8.6	5
8.7	5
8.8	Natural language(2), history(5), components(5), alternatives(3), impacts(3), backtracking(5)
8.9	Natural language(3), history(4), components(5), alternatives(3), impacts(4), backtracking(3)

9.3.2 Transcript: Interviewee 2

Question	Answer
B1	Operating system developer
B2	70%
1.1	Yes
2.1	Yes, but only sometimes, often need for very specific information instead of broad history context
2.2	Depends on the scenario and representation, might be non-consumable in very complex situations
3.1	Yes
3.2	Yes, identify which components are involved
4.1	No, need to do more research anyway
4.2	Yes
5.1	No
5.2	Yes
6.1	No, hard to know the value to be queried beforehand
6.2	Yes, curiosity
7.1	Yes, offers a chance to find root causes
7.2	No, very specific info, might give arbitrary output that is hard to classify
8.1	Yes, but need to be careful about representation of complex scenarios, might become overwhelming
8.2	Would help in general, but might provide too much information at once
8.3	If all decisions are available, yes, external influences will remain a problem
8.4	History, components, backtracking
8.5	Natural language, components, impacts
8.6	4
8.7	5
8.8	Natural language(2), history(4), components(4), alternatives(1), impacts(1), backtracking(5)
8.9	Natural language(4), history(4), components(3), alternatives(2), impacts(4), backtracking(0)

9.3.3 Transcript: Interviewee 3

Question	Answer
B1	Operating system developer
B2	30%
1.1	Yes, this graph has to be derived manually at the moment (or something similar)
2.1	Yes
2.2	Maybe, good for exploration, but would need to correlate this to source code as well
3.1	Yes
3.2	Yes
4.1	Yes, can help in special cases (non-intuitive choices, unclean code)
4.2	Yes
5.1	Maybe, in complex situations
5.2	Yes
6.1	Yes, allows to find odd one outs
6.2	Yes
7.1	Yes
7.2	Yes
8.1	Yes
8.2	No, need more context usually, using it next to source code would help though (more efficiency)
8.3	Operating system might be to complex in some areas, helpful in simpler cases
8.4	History, alternatives
8.5	Alternatives, components, impacts
8.6	4
8.7	4
8.8	Natural language(1), history(5), components(3), alternatives(3), impacts(1), backtracking(4)
8.9	Natural language(1), history(5), components(4), alternatives(3), impacts(5), backtracking(5)

9.3.4 Transcript: Interviewee 4

Question	Answer
B1	Operating system developer
B2	70%
1.1	Yes
2.1	Yes, allows to go back in history to find a set of estimated values
2.2	No, need more information on the actual meaning of the involved variables and values
3.1	Yes, allows to identify the component more easily that needs to do further analysis
3.2	Yes, allows to understand inter-dependencies within sub-components
4.1	No, at least experienced professionals will be hindered by this if they wanna quickly step through the decisions
4.2	Yes, very much
5.1	Yes
5.2	Yes, good to identify differences
6.1	Yes, allows verify/falsify intuition
6.2	Yes, allows to verify overall understanding
7.1	Yes, potential to save manual work
7.2	No, not trivial to know estimated value beforehand, manual analysis probably has more educational value
8.1	Yes, but can do without
8.2	No, would help in general by increasing transparency, but depending on the complexity of the problem, more detailed analysis (e.g. source code) will be necessary
8.3	No, can explain how the decision unfolded, but not why the particular decision was taken, so it is comprehensible how the system decided but not necessarily why the decision was taken (decision might be wrong in case of a bug for example)
8.4	History, components, impacts, backtracking
8.5	Natural language, history, components, alternatives
8.6	5
8.7	4
8.8	Natural language(2), history(5), components(3), alternatives(2), impacts(5), backtracking(5)
8.9	Natural language(5), history(5), components(5), alternatives(4), impacts(5), backtracking(2)

9.3.5 Transcript: Interviewee 5

Question	Answer
B1	Operating system developer
B2	80%
1.1	Yes
2.1	Yes, allows a certain level of code visualization
2.2	No
3.1	Yes
3.2	No
4.1	Yes
4.2	Yes, but highly dependent on quality of comments
5.1	Yes, understand unknown areas
5.2	No
6.1	Yes, but doubts on feasibility, definition of impact
6.2	Yes
7.1	Yes
7.2	Yes
8.1	Yes
8.2	Yes
8.3	No, complexity of operating system might be a problem, performance analysis might remain tricky, external influences might not be covered
8.4	combination of all
8.5	Natural language
8.6	4
8.7	5
8.8	Natural language(5), history(2), components(3), alternatives(4), impacts(4), backtracking(5)
8.9	Natural language(5), history(3), components(3), alternatives(4), impacts(4), backtracking(2)

Bibliography

[18] *Reform of EU data protection rules 2018*. European Commission. May 25, 2018. URL: https://eur-lex.europa.eu/legal-content/EN/ALL/?uri=celex%3A32016R0679 (visited on 06/12/2021) (cit. on p. 16).

[AB18] A. Adadi, M. Berrada. "Peeking inside the black-box: A survey on Explainable Artificial Intelligence (XAI)". In: *IEEE Access* 6 (2018), pp. 52138–52160 (cit. on p. 21).

[ABC+19] V. Arya, R. K. Bellamy, P.-Y. Chen, A. Dhurandhar, M. Hind, S. C. Hoffman, S. Houde, Q. V. Liao, R. Luss, A. Mojsilović, et al. "One explanation does not fit all: A toolkit and taxonomy of ai explainability techniques". In: *arXiv preprint arXiv:1909.03012* (2019) (cit. on pp. 7, 19).

[All08] J. Allspaw. *The art of capacity planning: scaling web resources*. Ö'Reilly Media, Inc.", 2008 (cit. on pp. 7, 15).

[BAC00] B. Boehm, C. Abts, S. Chulani. "Software development cost estimation approaches—A survey". In: *Annals of software engineering* 10.1 (2000), pp. 177–205 (cit. on p. 21).

[BC05] D. P. Bovet, M. Cesati. *Understanding the Linux Kernel: from I/O ports to process management*. Ö'Reilly Media, Inc.", 2005 (cit. on p. 13).

[BDG+04] D. Binkley, S. Danicic, T. Gyimóthy, M. Harman, A. Kiss, L. Ouarbya. "Formalizing executable dynamic and forward slicing". In: *Source Code Analysis and Manipulation, Fourth IEEE International Workshop on*. IEEE. 2004, pp. 43–52 (cit. on p. 26).

[BF08] R. Böhme, F. C. Freiling. "On metrics and measurements". In: *Dependability metrics*. Springer, 2008, pp. 7–13 (cit. on p. 19).

[BR08] R. Böhme, R. Reussner. "Validation of predictions with measurements". In: *Dependability metrics*. Springer, 2008, pp. 14–18 (cit. on p. 19).

[BSG+09] Y. Brun, G. D. M. Serugendo, C. Gacek, H. Giese, H. Kienle, M. Litoiu, H. Müller, M. Pezzè, M. Shaw. "Engineering self-adaptive systems through feedback loops". In: *Software engineering for self-adaptive systems*. Springer, 2009, pp. 48–70 (cit. on p. 10).

[BVK+14] J. Bourgeois, J. Van Der Linden, G. Kortuem, B. A. Price, C. Rimmer. "Conversations with my washing machine: an in-the-wild study of demand shifting with self-generated energy". In: *Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing*. ACM. 2014, pp. 459–470 (cit. on p. 16).

[CB05] C. Courage, K. Baxter. *Understanding your users: A practical guide to user requirements methods, tools, and techniques*. Gulf Professional Publishing, 2005 (cit. on pp. 18, 19).

[CBF+05] K. Cheverst, H. E. Byun, D. Fitton, C. Sas, C. Kray, N. Villar. "Exploring issues of user model transparency and proactive behaviour in an office environment control system". In: *User Modeling and User-Adapted Interaction* 15.3-4 (2005), pp. 235–273 (cit. on pp. 7, 23, 48).

[CLG+09] B. H. Cheng, R. de Lemos, H. Giese, P. Inverardi, J. Magee, J. Andersson, B. Becker, N. Bencomo, Y. Brun, B. Cukic, et al. "Software engineering for self-adaptive systems: A research roadmap". In: *Software engineering for self-adaptive systems*. Springer, 2009, pp. 1–26 (cit. on p. 11).

[CLRS09] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. *Introduction to algorithms*. MIT press, 2009 (cit. on p. 13).

[CLV+06] M. G. Core, H. C. Lane, M. Van Lent, D. Gomboc, S. Solomon, M. Rosenberg. "Building explainable artificial intelligence systems". In: *AAAI*. 2006, pp. 1766–1773 (cit. on p. 24).

[CO72] W. W. Chu, H. Opderbeck. "The page fault frequency replacement algorithm". In: *Proceedings of the December 5-7, 1972, fall joint computer conference, part I*. 1972, pp. 597–609 (cit. on p. 13).

[Coc18] I. Coca-Vila. "Self-driving cars in dilemmatic situations: An approach based on the theory of justification in criminal law". In: *Criminal Law and Philosophy* 12.1 (2018), pp. 59–82 (cit. on p. 16).

[Coo+04] A. Cooper et al. *The inmates are running the asylum:[Why high-tech products drive us crazy and how to restore the sanity]*. Vol. 2. Sams Indianapolis, 2004 (cit. on p. 18).

[CRP18] L. M. Cysneiros, M. Raffi, J. C. S. do Prado Leite. "Software transparency as a key requirement for self-driving cars". In: *2018 IEEE 26th International Requirements Engineering Conference (RE)*. IEEE. 2018, pp. 382–387 (cit. on pp. 16, 22).

[DC06] B. DiCicco-Bloom, B. F. Crabtree. "Making sense of qualitative research". In: *Medical Education* 40.4 (2006), pp. 314–321 (cit. on p. 18).

[DGM+13] R. De Lemos, H. Giese, H. A. Müller, M. Shaw, J. Andersson, M. Litoiu, B. Schmerl, G. Tamura, N. M. Villegas, T. Vogel, et al. "Software engineering for self-adaptive systems: A second research roadmap". In: *Software Engineering for Self-Adaptive Systems II*. Springer, 2013, pp. 1–32 (cit. on p. 11).

[Dij+59] E. W. Dijkstra et al. "A note on two problems in connexion with graphs". In: *Numerische mathematik* 1.1 (1959), pp. 269–271 (cit. on p. 60).

[Fow13] F. J. Fowler Jr. *Survey research methods*. Sage publications, 2013 (cit. on p. 18).

[Gre13a] B. Gregg. *Systems performance: enterprise and the cloud*. Pearson Education, 2013 (cit. on pp. 7, 14, 20).

[Gre13b] B. Gregg. "Thinking methodically about performance". In: *Communications of the ACM* 56.2 (2013), pp. 45–51 (cit. on pp. 15, 20).

[HBH88] E. J. Horvitz, J. S. Breese, M. Henrion. "Decision theory in expert systems and artificial intelligence". In: *International journal of approximate reasoning* 2.3 (1988), pp. 247–302 (cit. on p. 24).

[HBK+17] N. Herbst, S. Becker, S. Kounev, H. Koziolek, M. Maggio, A. Milenkoski, E. Smirni. "Metrics and Benchmarks for Self-aware Computing Systems". In: *Self-Aware Computing Systems*. Springer, 2017, pp. 437–464 (cit. on pp. 7, 19, 25).

[HBPK17] A. Holzinger, C. Biemann, C. S. Pattichis, D. B. Kell. “What do we need to build explainable AI systems for the medical domain?” In: *arXiv preprint arXiv:1712.09923* (2017) (cit. on pp. 16, 22).

[HMKL18] R. R. Hoffman, S. T. Mueller, G. Klein, J. Litman. “Metrics for explainable ai: Challenges and prospects”. In: *arXiv preprint arXiv:1812.04608* (2018) (cit. on pp. 7, 25).

[HMU01] J. E. Hopcroft, R. Motwani, J. D. Ullman. “Introduction to automata theory, languages, and computation”. In: *Acm Sigact News* 32.1 (2001), pp. 60–65 (cit. on p. 51).

[IKL+97] J. Irwin, G. Kickzales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. Loingtier. “Aspect-oriented programming”. In: *Proceedings of ECOOP, IEEE, Finland* (1997), pp. 220–242 (cit. on pp. 26, 55).

[Jai90] R. Jain. *The art of computer systems performance analysis: techniques for experimental design, measurement, simulation, and modeling*. John Wiley & Sons, 1990 (cit. on pp. 7, 15).

[JLT85] E. D. Jensen, C. D. Locke, H. Tokuda. “A time-driven scheduling model for real-time operating systems.” In: *Rtss*. Vol. 85. 1985, pp. 112–122 (cit. on p. 12).

[KC03] J. O. Kephart, D. M. Chess. “The vision of autonomic computing”. In: *Computer* 1 (2003), pp. 41–50 (cit. on p. 10).

[KL90] B. Korel, J. Laski. “Dynamic slicing of computer programs”. In: *Journal of Systems and Software* 13.3 (1990), pp. 187–195 (cit. on p. 26).

[KLSZ92] A. Kiebach, H. Lichter, M. Schneider-Hufschmidt, H. Züllighoven. “Prototyping in industriellen Software-Projekten: Erfahrungen und Analysen”. In: *Informatik Spektrum* 15.2 (1992), pp. 65–77 (cit. on p. 18).

[LBS99] S. Lu, V. Bharghavan, R. Srikant. “Fair scheduling in wireless packet networks”. In: *IEEE/ACM Transactions on networking* 7.4 (1999), pp. 473–489 (cit. on p. 10).

[LD02] C. Lacave, F. J. Díez. “A review of explanation methods for Bayesian networks”. In: *The Knowledge Engineering Review* 17.2 (2002), pp. 107–127 (cit. on p. 17).

[LFW15] P. Lenberg, R. Feldt, L. G. Wallgren. “Behavioral software engineering: A definition and systematic literature review”. In: *Journal of Systems and software* 107 (2015), pp. 15–37 (cit. on p. 18).

[MHS17] T. Miller, P. Howe, L. Sonenberg. “Explainable AI: Beware of inmates running the asylum or: How I learnt to stop worrying and love the social and behavioural sciences”. In: *arXiv preprint arXiv:1712.00547* (2017) (cit. on pp. 17, 25).

[Mol20] C. Molnar. *Interpretable machine learning*. Lulu. com, 2020 (cit. on p. 48).

[NAC02] H. Núñez, C. Angulo, A. Català. “Rule extraction from support vector machines.” In: *Esann*. 2002, pp. 107–112 (cit. on p. 17).

[NL14] L. Null, J. Lobur. *Essentials of Computer Organization and Architecture*. Jones & Bartlett Publishers, 2014 (cit. on p. 9).

[RS10] P. Rogers, A. Salla. *ABCs of Z/OS System Programming, Volume 12*. Vol. 12. IBM Redbooks, 2010 (cit. on pp. 11, 15).

[SBC+12] F. Sironi, D. B. Bartolini, S. Campanoni, F. Cancare, H. Hoffmann, D. Sciuto, M. D. Santambrogio. “Metronome: operating system level performance management via self-adaptive computing”. In: *Proceedings of the 49th Annual Design Automation Conference*. 2012, pp. 856–865 (cit. on p. 11).

[Sch92] U. Schöning. *Theoretische Informatik-kurz gefasst*. BI Wissenschaftsverl., 1992 (cit. on p. 51).

[Sip96] M. Sipser. “Introduction to the Theory of Computation”. In: *ACM Sigact News* 27.1 (1996), pp. 27–29 (cit. on p. 60).

[SPG91] A. Silberschatz, J. L. Peterson, P. B. Galvin. *Operating system concepts*. Addison-Wesley Longman Publishing Co., Inc., 1991 (cit. on p. 10).

[SSS+17] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, et al. “Mastering the game of go without human knowledge”. In: *nature* 550.7676 (2017), pp. 354–359 (cit. on p. 22).

[Swa83] W. R. Swartout. “XPLAIN: A system for creating and explaining expert consulting programs”. In: *Artificial intelligence* 21.3 (1983), pp. 285–325 (cit. on pp. 7, 24, 50).

[SWM17] W. Samek, T. Wiegand, K.-R. Müller. “Explainable artificial intelligence: Understanding, visualizing and interpreting deep learning models”. In: *arXiv preprint arXiv:1708.08296* (2017) (cit. on pp. 7, 17, 22).

[TB15] A. S. Tanenbaum, H. Bos. *Modern operating systems*. Pearson, 2015 (cit. on pp. 10, 12, 13).

[WDH+20] A. Wan, L. Dunlap, D. Ho, J. Yin, S. Lee, H. Jin, S. Petryk, S. A. Bargal, J. E. Gonzalez. “NBDT: neural-backed decision trees”. In: *arXiv preprint arXiv:2004.00221* (2020) (cit. on p. 23).

[Wei84] M. Weiser. “Program slicing”. In: *IEEE Transactions on software engineering* 4 (1984), pp. 352–357 (cit. on pp. 26, 59).

[WPZZ07] C. Weiss, R. Premraj, T. Zimmermann, A. Zeller. “How long will it take to fix this bug?” In: *Fourth International Workshop on Mining Software Repositories (MSR’07: ICSE Workshops 2007)*. IEEE. 2007, pp. 1–1 (cit. on p. 21).

[WRK+17] S. F. Weng, J. Reps, J. Kai, J. M. Garibaldi, N. Qureshi. “Can machine-learning improve cardiovascular risk prediction using routine clinical data?” In: *PloS one* 12.4 (2017), e0174944 (cit. on p. 16).

[YN13] R. Yang, M. W. Newman. “Learning from a learning thermostat: lessons for intelligent systems for the home”. In: *Proceedings of the 2013 ACM international joint conference on Pervasive and ubiquitous computing*. ACM. 2013, pp. 93–102 (cit. on p. 16).

[Zel09] A. Zeller. *Why programs fail: a guide to systematic debugging*. Elsevier, 2009 (cit. on pp. 16, 20, 35).

[ZKZH12] F. Zhang, F. Khomh, Y. Zou, A. E. Hassan. “An empirical study on factors impacting bug fixing time”. In: *2012 19th Working Conference on Reverse Engineering*. IEEE. 2012, pp. 225–234 (cit. on p. 21).