Institute of Software Engineering
Software Quality and Architecture

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Integrating Explanation Generation into the Palladio Tool Chain

Jan Haas

Course of Study: Informatik

Examiner: Prof. Dr.-Ing. Steffen Becker

Supervisor: Sarah Sophie Stieß, M.Sc.
Floriment Klinaku, M.Sc.

Commenced: November 4, 2022

Completed: May 4, 2023

Abstract

Context. Modeling and simulating software behavior using Palladio can be an important step in ensuring the softwares quality.
Problem. However, the end results of a simulation may not be sufficient to understand where and why a self-adaptive system decided to reconfigure itself.
Objective. This thesis extends the Palladio tool chain, adding a component that can generate the explanations to these and more questions based on the Slingshot simulator.
Method. For this, we gathered usage scenarios where adaptation decisions required further explanations. Using them as requirements, we implemented our explanation generation component as an expert system with a focus on flexibility and expandability.
Result. The component functions as a modular extension to Palladios Slingshot simulator that can be dynamically configured to give explanations in different forms as required. For our evaluation we performed a user survey on the importance of different questions to be answered.
Conclusion. We believe that the component we have implemented is flexible enough to serve as an easily expanded and adapted tool for explanation generation as well as similar tasks. The scenarios we created as well as the ideas behind them may help others define similar requirements in a more structured way.

Kurzfassung

Kontext Software in Palladio zu modellieren und zu simulieren kann ein wichtiger Schritt darin sein, die Qualität der Software zu gewährleisten.
Problemstellung Die Endergebnisse einer Simulation können allerdings nicht ausreichen um zu verstehen wo und warum sich ein selbst-anpassendes System neu konfiguriert hat.
Ziele In dieser Masterarbeit wird Palladio um eine Komponente erweitert, die diese und andere Fragen beantworten kann. Wir bauen dabei auf den Slingshot Simulator auf.
Vorgehensweise Wir beginnen damit, Szenarien zu sammeln in denen solche Fragen beantwortet werden müssen. Mit diesen als Voraussetzungen implementieren wir unsere Erklärungs-Komponente als Expertensystem. Wir legen besonderen Wert darauf, dass diese flexibel und leicht erweiterbar ist.
Ergebnisse Die erstellte Komponente stellt eine modulare Erweiterung zu Palladios Slingshot Simulator dar, die dynamisch konfiguriert werden kann bestimmte Erklärungen zu geben. In welcher Form diese ausgegeben werden ist ebenfalls leicht anpassbar. Zur Evaluation führen wir außerdem eine Nutzerbefragung durch, in der die Wichtigkeit einzelner zu beantwortender Fragen erörtert wird.
Fazit Wir glauben, dass die von uns implementierte Komponente flexibel genug ist, um als leicht anpassbares Werkzeug zur Erklärungs-Generierung zu dienen. Zusätzlich können die Szenarien und wie wir diese festgehalten haben nützlich für ähnliche Projekte sein.

Contents

1 Introduction **1**

2 Foundations and Related Work **3**
2.1 Literature Research Methodology 3
2.2 Foundations 4
2.3 Related work 6

3 Scenarios **9**
3.1 Motivation 9
3.2 Scenario Sheets 9
3.3 Example Scenarios 10

4 Design **11**
4.1 Terms 11
4.2 Structure 11
4.3 MAB-EX 12
4.4 Case Definitions 13
4.5 Users Guide 15
4.6 Changes we had to make 18

5 Implementation **19**
5.1 General Structure 19
5.2 Monitors 20
5.3 Analyzer 20
5.4 Builder 21
5.5 Interpreters 22
5.6 MABEXStrings 23
5.7 Did we re-implement Complex Event Processing (CEP)? 23

6 Evaluation **25**
6.1 User Survey Design 25
6.2 Results 26
6.3 Discussion 29
6.4 Threats to Validity 32

7 Conclusion **35**
7.1 Summary 35
7.2 Benefits 35
7.3 Limitations 35
7.4 Lessons Learned 36

7.5 Future Work 36

Bibliography **39**

A Scenario Sheets **43**

List of Figures

4.1 The explanation generation component - Architecture Overview 12

5.1 Dataflow during explanation generation 20
5.2 MABEXAnalyzer: workflow 21
5.3 Building an Explanation 22

6.1 Survey Results as a box and whiskers plot 28

List of Tables

6.1 Survey Results: *-marked questions have been adapted, **-marked questions have been combined and adapted 27

List of Listings

4.1 The basic structure of a case definition 14
4.2 An example case definition file 15
4.3 An example explanation text file 17
6.1 Threshold crossed case definition 30
6.2 Model changed case definition 30

Acronyms

CEP Complex Event Processing. vii, 5

EDA Event Driven Architecture. 4

EMF Eclipse Modeling Framework. 4

EPL Event Processing Language. 12

ERT Expertise, Risk and Time Explainability. 13

HCI Human Computer Interaction. 16

IDE Integrated Development Environment. 4

PCM Palladio Component Model. 4

SAS Self-adaptive System. 1, 4

SLA Service Level Agreement. 5

SLI Service Level Indicator. 5

SLO Service Level Objective. 5

SPD Scaling Policy Definition. 29

XAI Explainable Artificial Intelligence. 4

1 Introduction

The Palladio Software Simulation Approach can be used to analyze the performance and reliability of software at design time. This kind of early testing becomes more important as software architectures become more complex and development costs rise as a result. Model-level testing can also help identify faults in distributed or cloud-based applications that are hard to track down without being able to observe the whole system synchronously.
However, the end results of a simulation may not be sufficient to understand where and why a Self-adaptive System (SAS) decided to reconfigure itself. These systems increasingly make use of artificial intelligence [PGB21] to provision servers, correct the course of a drone or other tasks. While one can build such a system to explain itself (e.g. [DW21], [WAM+21]), this is not yet possible when simulating the unfinished software.
With this thesis, we aim to provide one solution to this problem by adding a new component to the Palladio tool chain [BKR09][Pal22]. The component takes the events provided by Palladios new Slingshot simulator [KKB21][Kat21] and generates a human-understandable explanation from them.
We started by gathering requirements for our component using a scenario-based approach. They consist of questions the component should be able to answer. We focused these questions on the field of auto-scaling web applications due to available literature in this field (e.g. [Zil22], [BLV+19], [USCG05]). Next, we performed a user survey to rank the importance of these questions. Finally, we implemented the component over several iterations and evaluated it using the ranked questions as a basis.
The resulting component extends Palladio by an expert system that can be dynamically configured to detect patterns in Slingshots event stream. It can be easily expanded to monitor new events or to provide explanations and similar outputs in different formats, even simultaneously.
The main contribution of this thesis is the new Palladio extension for explanation generation. It additionally contributes a set of self-adaptation scenarios we have used as a source of requirements and the structure they are presented in. Finally, we contribute insights into how important different explanations are to developers through a user survey.

Given the problem described at the beginning of this section in the context of the Palladio approach, we raise the following research goals:

RG1 How can a Palladio component implement an explainability approach?

RG2 For the chosen explainability approach, how should Palladio simulation data be prepared?

RG3 How should a Palladio component access the required knowledge on explanations?

In the following chapters, we show how we worked towards these goals. We introduce the component we have implemented and how we gathered the requirements for it. We evaluate the component and discuss our results before pointing out possible future research and what we have learned.

Thesis Structure

This thesis is structured into the following chapters:

Chapter 2 – Foundations and Related Work In this chapter we explain terms necessary for understanding the topic of this thesis and give an overview of related work.

Chapter 3 – Scenarios Next, we introduce our take on scenario-based requirements engineering.

Chapter 4 – Design Here, we provide insights into how the explanation generation component was designed as well as how to use it.

Chapter 5 – Implementation This chapter gives details on the implementation of the component.

Chapter 6 – Evaluation Here, we go over how we evaluated our contributions and the results of this evaluation.

Chapter 7 – Conclusion We conclude our thesis by indicating whom this thesis might benefit and give an outlook on future work.

2 Foundations and Related Work

In this chapter we give an overview of the existing knowledge and literature relevant to this thesis. It mainly contributes an explanation generation component to the Palladio tool chain but also gives insights into how relevant different kinds of information are to users of similar applications. Additionally, we created structured scenario documents that can be used to quickly construct examples or test cases for explanation generation mechanisms.

Before we introduce the foundations of this thesis, we first review our literature research methodology. Finally, we explore similar works and point out their relevant parts as well as differences to this thesis.

2.1 Literature Research Methodology

We mainly used Google Scholar [Goo] to find relevant papers. In a few cases, we were able to find additional documents from the recommendations given by the digital library hosting one of these papers. The search terms we used are documented in the following list:

- explainability
- explainability adaptive systems
- requirements for explanation generation
- self adaptive systems requirements
- expert systems
- rule based expert systems
- explanation generation
- explainability framework
- self explaining artificial intelligence

We used the information given in a papers abstract as a first point in judging whether it could be relevant for our thesis. Seemingly relevant papers where then further judged according to key chapters such as results and conclusions. For papers we consulted concerning constructing our component, age was also an important point due to information becoming outdated more quickly in this regard.

2.2 Foundations

The component developed during this thesis is an extension to the Palladio Bench and its event-driven simulator Slingshot. These and other terms necessary for understanding this thesis will be explained in this section.

2.2.1 Palladio

Palladio [BKR09][Pal22] is a software architecture simulation approach used for analyzing software at a model level. It can point out performance bottlenecks, threats to the reliability of the software and other issues while allowing for subsequent optimization at design time. For this, software is modeled using the Palladio Component Model (PCM), which itself is implemented within the Eclipse Modeling Framework (EMF). The recommended way to do so is by using the Palladio Bench, an integrated modeling environment based on the Eclipse Integrated Development Environment (IDE). With it, one can create a model of a system, simulate it and see where optimization may be possible or needed.

2.2.2 Slingshot

Slingshot is a performance simulator for PCM models based on Event Driven Architecture (EDA) [BD10] developed by Katić et al. [KKB21][Kat21]. A simulator takes a modeled system and calculates what happens during a given usage scenario while recording performance metrics. Apart from being more extensible than preexisting simulators, the event-based operation of Slingshot allows for clear points of interest that may either require an explanation or can be used for generating one.

2.2.3 Self-adaptive System (SAS)

These are systems that can change their behavior if changes in the environment, requirements or other aspects make their system objectives harder to realize. Self-adaptive software achieves this through constant monitoring and evaluation of its own behavior, triggering an adaptation if it detects non-satisfactory performance (e.g. [BDG+09], [MHdH13]). Depending on how it is set up, the system could for example change its parameters, acquire more resources from a cloud provider or adapt a model to better reflect observed values.

2.2.4 Explainability

With the increasing application of machine learning, especially highly accurate but intransparent deep learning models [VL21], come increasing requirements for these models to explain their decision processes (e.g. [Shi21]). Explainable Artificial Intelligence (XAI) or Explainability in general is needed for users to trust in the decisions of intelligent or adaptive software and thus critical for their acceptance in current times where data protection and safety are increasingly important [HKWT18]. The same concepts can also help in allowing users more quickly understand

complex or not directly exposed processes in other systems. As Slingshot is currently expanding to allow for the simulation of self-adaptive systems, we consider this point the main motivation behind our thesis.

2.2.5 MAB-EX

The MAB-EX framework [BGC+19] is an adaptation of the MAPE control loop [IAA20] describing a structured approach to generating explanations. It separates this process into four parts: *Monitoring* the application in question to get relevant information. *Analyzing* the information to detect cases that require an explanation and to compute e.g. aggregated values as needed. *Building* an explanation from the data by gathering all relevant information into an intermediate format. And *EXplaining*, as in delivering the explanation to a user in an understandable form. This form should be adapted to what a specific user needs or understands, so there may be multiple different outputs for the same explanation depending on which users are present. We chose to implement this framework over other options (see Section 2.3) due to fitting our requirements well (see Section 4.3). Blumreiter et al. suggest different knowledge bases for the analyzer, of which we chose to implement an expert system in this thesis given the lack of available training data and time constraints.

2.2.6 Complex Event Processing (CEP)

EDA describes the general style of working with events to better reflect real processes (e.g. [BD10]). Meanwhile, CEP specifically refers to techniques used to process massive quantities of events in real time. It helps with aggregating information and finding cause-and-effect relationships between events (e.g. [CM12]).

2.2.7 Expert Systems

An expert system employs artificial intelligence to mimic the decisions of an expert on the respective field [Jac86; Shu05]. The knowledge of human experts is encoded into a machine-readable form and then queried to draw conclusions from given information. The system can then act on these conclusions and reason about how it arrived at them if required. With this thesis, we implemented a rule-based expert system. Here, the experts knowledge is encoded into rules that can be followed to determine whether a specific conclusion matches the presented inputs.

2.2.8 Service Level Objective (SLO)

A service provider and their customer agree on a Service Level Agreement (SLA) to define when the service is properly working and how the provider has to respond if it fails to do so [JWNS]. For a metric such as throughput or error rate of the service, a so called Service Level Indicator (SLI), the agreement defines constraints. These constraints are called SLOs as they designate the objectives the service provider has to fulfill. They help in avoiding disputes over the service quality based on misunderstanding between the provider and customer.

2.3 Related work

In this section, we will introduce related works, explain their core concepts and highlight differences to our thesis. We have gathered requirements using scenarios, implemented an explainability framework and evaluated our explanation generation approach.

2.3.1 Scenario-based requirements engineering

Scenario-based requirements engineering is already an established method. As an example, Sutcliffe [Sut03] explains the concept and introduces different existing definitions of scenarios. They review the respective more or less formal representations and where in the requirements engineering process scenarios can play a role. Sutcliffe proceeds by listing advantages and disadvantages of scenarios: They allow us to base our reasoning on specific points, but we may lose generality in the process. Scenarios force attention to details that may be ignored when using more abstract models. On the other hand, generating a sufficient amount for full test coverage may be too costly. Overall, responsibly used scenarios can be an important help in the design process. Next, Sutcliffe shows where scenarios can be used and introduces different methods of scenario-based requirements engineering. However, this does not yet include scenarios specific to explainability and the unique challenges of this field.

Wolf [Wol19] on the other hand introduces the idea of using scenarios for requirements engineering in the field of XAI. They specifically investigate using them for finding requirements related to deploying an XAI system into a complex setting of use. With two short stories as example scenarios, they point out challenges such as having users with different view-points that need to make sense of the same explanation. Wolf argues that using scenarios early on in a design process can help reveal unintended outcomes and possible vulnerabilities. In their scenario design, they refer to four key aspects given by Go and Carrol [GC04]: actors, their background, their goals and the sequence of actions. However, their story-based description of scenarios is more informal and thus unsuited for the additional uses we envisioned.

2.3.2 Explainability Frameworks

While we chose to implement the MAB-EX framework, there are other options as well. We will introduce some here and justify our choice for MAB-EX in Section 4.3.

Parra-Ullauri et al. [PGB21] argue that explanations are often given only for specific situations but that a more general scope is required. They mention that these global explanations help more in promoting trust in the system as a whole compared to only in single decisions. According to their paper, runtime models are already used in SAS, so they propose using an event graph model to generate these global explanations from evolving runtime models. These event graphs allow displaying system behaviour as a state-time diagram while CEP is used to correlate events and to derive conclusions. To promote their approach, Parra-Ullauri et al. then apply it to an example SAS in the domain of mobile communication.

Simkute et al. [SLJ+21] propose the conceptual Expertise, Risk and Time Explainability framework (ERT) based on cognitive psychology and human factors literature. They propose using their ERT framework for deployment and iterative development of effective heuristics for intelligible interface design. The ERT framework is based on three dynamics: How much a decision maker relies on their expertise for solving a given task, the risk environment a decision is made in and time constraints. Simkute et al. argue that these most heavily impact the type of explanation required.

Fernández et al. [RMMH22] introduce the Explanation Sets framework, which combines semifactuals and counterfactuals, two instances of XAI techniques. These techniques explain observations by comparing them to another one. For counterfactuals, the resulting predictions are different, while they are the same for semifactuals. Their framework groups observations into different neighborhoods (the Explanation Sets) where most of them satisfy conditions defined on the neighborhood. An explanation set can either be dissimilar (counterfactuals) or similar (semifactuals). Fernández et al. argue that a mayor strength of their approach is being able to extend the explanations to other tasks by providing an appropriate grouping measure.

2.3.3 Explainability Evaluation

In their masters thesis, Zilch [Zil22] evaluates explainability in autoscaling frameworks. After reimplementing an existing autoscaler, they conducted a user survey on what functionality said framework should offer. They then used the results to build an evaluation scheme for autoscaling frameworks. Apart from explainability properties, they also cover configuration properties and accessibility properties. However, their evaluation of explainability is more concerned with the overall quality of the framework, whereas we looked into what information is important to users.

Sokol and Flach [SF20] on the other hand introduce Explainability Fact Sheets for assessing explainable systems using five categories: Functional, operational, usability, safety and validation. For this, they surveyed XAI literature and collected the criteria authors used explicitly or implicitly. They looked at papers proposing new algorithms as well as those purely suggesting how to evaluate them. Sokol and Flach argue that their framework not only allows one to evaluate an explainability approach, but also to discover differences between its theoretical and implemented performance. However, their evaluation criteria are again more concerned with the quality of the system itself and do not consider the users informational needs.

3 Scenarios

To gather requirements for our explanation generation component, we have used a scenario-based approach. We adapted a suggestion by Wolf [Wol19] to fit our needs and suggest the term "scenario sheets" for the resulting documents.

3.1 Motivation

Wolf [Wol19] suggests the use of explainability scenarios to help in finding requirements that emerge when XAI systems are used in more complex environments. However, their examples are used more to guide a developers thought process and are written as informal stories. We instead used them as a more direct source of requirements in the form of questions our explanation generation component should be able to answer. Additionally, we ranked the importance of these questions in a survey (see Section 6.1) for evaluating our component (see Chapter 6). To ensure the validity of this process, we decided to document the scenarios in a more formal way.

3.2 Scenario Sheets

Inspired by Design Patterns in Software Engineering [GHJ+95], we propose to write scenarios more formally. This would allow others to more accurately understand them. It would additionally give them more weight when used for evaluating a design and make it easier to use them for development and testing. We build on this idea by suggesting a general structure and additional requirements that should be adhered to:

Reproducibility When conducting a study, it is usually desired that one should share its structure, raw data and any additional factors. This makes it possible for others to reproduce the study accurately and to verify the results. We suggest that this should be a requirement when sharing scenarios as well. Ideally, a scenario sheet should contain or point to a model or structured description of the scenario in question. According to our experience, having a model also helps in conveying the scenario more accurately.

Variations While these scenarios can be used for generating requirements or questions an explanation should answer, we suggested also using them for development and testing. To this end, a scenario sheet could include notes on what parts of the example scenario can be changed while keeping its core concept intact.

Document Structure We additionally suggest that a scenario sheet should be structured and concise, similar to how Architectural Patterns are shared in Software Engineering [GHJ+95]. For the scenario sheets created with this thesis, we used the following structure:

Description: A brief description of the scenario that allows for quickly grasping its core concept.

Example Model: The Palladio model we created for this scenario.

Relevant Questions: Which questions we think an explanation should answer for this scenario. These were produced from key aspects of the scenario, inherited from more basic scenarios and taken from our user survey (see Chapter 6).

Variations: Which parts of the scenario can be changed without affecting its core concepts.

Additional Notes: Apart from comments not fitting elsewhere, we introduced which field of research the scenario belongs to here. This allows anyone unfamiliar with the topic to at least classify where to place the scenario.

3.3 Example Scenarios

For this thesis we created the scenario sheets described below. They are focused around auto scaling in the field of web-based applications. At the time of writing this thesis, this field was already well covered by other papers (e.g. [Zil22], [BLV+19], [USCG05]). It therefore provided a good foundation to work on. The scenario sheets are placed in Appendix A at the end of the document. The questions we sourced from them are detailed in Section 6.1.

Please note that at the time of creating these scenarios, the stable Slingshot version did not yet support defining model adaptations, making the models incomplete.

Base Scenario: Threshold Crossing As a basis for other scenarios and as a simple test case, we included a simulated system that monitors the response time of a function and takes an action once a threshold is crossed.

Scenario: Delayed Provisioning Next, we modeled a single-service web-based system that can provision more processing power, but does so too slowly.

Scenario: Overprovisioning On the other hand, a similar system may function correctly, but provision an uneconomical amount of additional processing power. As such, an explanation generation component should be able to answer questions regarding this topic as well.

Scenario: Oscillating Provisioning As a more complex scenario, we modeled a system where the upper threshold is higher than the next lower threshold. This causes it to rapidly increase and decrease the available processing power and requires an explanation that includes multiple events.

Scenario: Cascading Delay If the system is instead structured like a pipeline and uses a different, greedy load balancer for each part, it can take a long time until a spike in requests can be dealt with completely. This happens when the first service in the pipeline processing at initial capacity does not yet cross the upper threshold of the second service and so forth. When a spike in requests then occurs, the second service only detects it after the first service has finished scaling, the third service only when the second is done etc. Depending on the longest path in the pipeline, an SLO violation may not be resolved for a long time with this setup. As such, it raises questions regarding the structure and relations in the system.

4 Design

In this chapter, we will go over the design of the explanation generation component we have implemented with this thesis. First, we will go over the terms we have come to use for different parts of our design. Then, we will showcase the general structure of the component and reason on why we chose to follow the MAB-EX framework [BGC+19]. As a core concept, we introduce the case definitions that define the rules our component follows. We have included a Users Guide (Section 4.5) that explains how to set up and use the component. At the end, we explain design changes we had to make.

4.1 Terms

Case An event chain or pattern that requires an explanation.

Detail One aspect of a Slingshot event that may be required for an explanation.

Explanation An explanation in machine-readable form.

Monitor The first step in MAB-EX is to *monitor*. In this case, we monitor Slingshots event stream.

Analyzer The second step in MAB-EX is to *analyze* the events.

Builder For detected patterns, we *build* an explanation in MAB-EX' third step.

Interpreter While the last step in MAB-EX is to *explain*, we found that to *interpret* described this step better for our component.

4.2 Structure

Our explanation generation component implements the MAB-EX framework suggested by Blumreiter et al. [BGC+19] (see Section 4.3). It is designed as an expert system that focuses on ease of use and expandability while its main workflow is split into four parts:

- Monitoring the events occurring during a Palladio simulation.
- Analyzing whether expert-defined cases occur in this stream of events.
- Building a machine-readable Explanation from the events involved in a case.
- Interpreting these Explanations for the user.

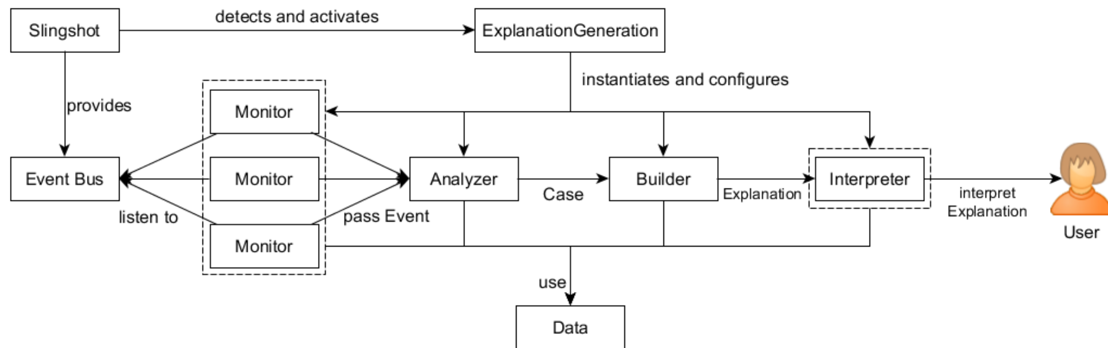

Figure 4.1: The explanation generation component - Architecture Overview

Each part is implemented as a separate module and their connections are shown in Figure 4.1. The diagram also outlines that while there is only one Analyzer and one Builder, there are multiple Monitors and possibly Interpreters. For each event type the component should detect, a different Monitor is used. This allows a Palladio expert to define exactly what information should be extracted from this specific event type. On the other hand, the component also supports having multiple Interpreters. This way different jobs such as logging specific cases to a file, giving explanations to the user or filling plots as the simulation runs can be completed at the same time while maintaining a separation of concerns.

Apart from the Analyzer and Builder sharing a configuration for the experts convenience, different parts generally communicate using a single method per connection. This makes it possible to adapt or replace all parts while we put special care in allowing to rapidly add or adapt Monitors and to add new Interpreters. For more details, see Chapter 5 on the implementation.

Another core aspect of the component is that its main logic is dynamically created. Inspired by the Event Processing Language (EPL) developed by EsperTech [Esp], which itself extends SQL, we have created a language that allows for defining arbitrary cases using a single YAML-file (see Section 4.6 for the connection to Esper). Each case is defined similarly to an SQL statement and describes a chain of events, conditions on these events and what data should be passed on if the case occurs.

Figure 4.1 also shows that a core module (*ExplanationGeneration*) acts as an entry point and registers as a Slingshot extension while the data module collects data classes in one place.

4.3 MAB-EX

We have already introduced MAB-EX in detail in the Foundations section (see Section 2.2) of this thesis. As a reminder, Blumreiter et al. [BGC+19] have suggested the Monitor, Analyze, Build and EXplain steps for an explanation generation component that attaches to an existing system.

The MAB-EX framework has several selling points that made it a good fit for our use case. It is designed as an add-on rather than an integration, meaning it required no changes to Palladio or the Slingshot simulator. If it is present, it simply uses the extension point provided by Slingshot to register itself and install its monitors. Next, the different steps are easily split into separate

modules. This meshes well with the OSGi frameworks approach of modular software [OSG] used by Palladio and the similar trend away from monolithic design. As the modules do not require complex connections either, we could implement message-based interactions that allow for simply exchanging modules if a use case requires it. Finally, the idea of differentiating between building a machine-readable explanation and actually explaining to the user allows our component to be used for one or more tasks at the same time while maintaining a separation of concerns between them.

Compared to MAB-EX, other concepts we have presented in Section 2.3 either serve different purposes or lack advantages we were prioritizing:

Parra-Ullauri et al. [PGB21] suggested the use of event graph models for providing global explanations, but for Palladio, we already have global insights and are rather looking for local explanations. They used CEP as we eventually did, although in a different way. On the other hand, we do not have a counterpart to their event graph, which might be worth looking into if the need for answering questions on demand after a simulation arises.

Simkute et al. [SLJ+21] proposed their Expertise, Risk and Time Explainability (ERT) framework, but their work provides a set of guidelines rather than a frame to implement. They also focus on the human aspect of how an explanation should look like, while with our implementation of the MAB-EX framework, this only becomes relevant for the last step. Their ERT framework could however be considered when implementing a new interpreter.

Fernández et al. [RMMH22] introduced the Explanation Sets framework that combines two different XAI techniques. Referencing MAB-EX, their framework is only a counterpart to the Analyzer however and relies on grouping observations with previous ones. As we do not have a good source of training data and given the time constraints placed on a thesis, training a machine learning model also did not seem feasible to us.

4.4 Case Definitions

Case definitions are a core concept of making this component highly customizable. A case defines a series of one or more events occurring during a Slingshot simulation that together warrants an explanation. Case definitions are written similarly to SQL statements. With the Analyzer and Builder as implemented in this thesis, a single YAML file can be configured as the source to read these case definitions from. The YAML syntax is for example specified on the official website [Yam].

At the point of writing this thesis, the relative or total path to the case definition file had to be set in the *AbstractMABEXAnalyzer* class' *caseDefaultLocation* constant. The case definitions used are updated on simulation start, so it is unnecessary to restart the target Eclipse application if there have only been changes to the already targeted YAML file.

The file must start with a single root tag called **cases:** that all case definitions are subordinated to. A case definition itself has the following structure:

Listing 4.1 The basic structure of a case definition

```
<name>:
    trigger: <event>, <event>,...
    select: <event>.<detail>, <event>.<detail>, ...
    where: <condition> AND <condition> ...
    stop: <condition> AND <condition> ...
```

<name>: identifies the case and must be unique. An interpreter can match their explanations to the correct case using these.

trigger: is a required part of the definition and declares the chain of events leading up to this case as a comma-separated list. For convenience, the component provides a list of all available events and the details they provide at simulation start. The list is currently being printed to the console of the primary Eclipse instance.

- these events must occur in order for the case to be completed, but other events may occur between them
- The same event type may be used more than once, even in succession.
- Events may be given an alias using the **AS** keyword.
- Within one case definition, aliases must be unique among themselves and any non-aliased events. Please note that the ANY alias is reserved for use with *stop*.
- if a case requires two events of the same type, at least one must have an alias

select: is a comma-separated list used by the Builder to determine what details to add to an Explanation. While this may not find use often, select is an optional component. Details are written as **<alias or event type>.<detail>** as is common in SQL or programming languages. Apart from passing on event details, we also support having the Builder compute secondary data from details of the trigger events. You can find a list of the computed values we have implemented in Section 4.5.1.

where: narrows down what events count towards this cases event chain. Conditions can be combined using **AND**, which acts as a logical and as would be expected. While we do not currently support a logical or, there is a **NOT** condition that can be used to simulate this functionality. Again, a full list of the conditions implemented with this thesis can be found in Section 4.5.1.

stop: describes when to abandon this case, which is checked *first* whenever a new event arrives. With this, it serves two purposes: Cases can be cleaned up once they are unlikely to occur, improving performance. They can also be further narrowed by abandoning a case if a certain other event or any event carrying a certain detail arrives. The syntax for *stop* is the same as for *where*, with one exception: The **ANY** alias is available for matching all arriving events.

This answers the first part of our third research goal: How should a Palladio component access the required knowledge on explanations? (compare Chapter 1)

Listing 4.2 An example case definition file

```
#-----------------------------------------------------------------------------------------------
# Feel free to add comments such as this one for readability.
# Empty lines between different tags do not affect the parser either.
#-----------------------------------------------------------------------------------------------
cases:
    UserFinishedEarly:
        trigger: UserFinished
        select: UserFinished.user
        where: UserFinished.time < 10%d     # details in conditions do not have to be selected

    UsersArriveTogether:
        trigger: UserStarted AS us1, UserStarted AS us2
        select: us1.user, us2.user
        where: NOT DIFFERENCE us1.time us2.time > 0.5%d
            # if a condition is not directly available yet, NOT may help

    UsersArriveTogetherAlternative:
        # functionally the same as the case before, but offers better performance
        # as it gets abandoned once the case is not possible anymore
        trigger: UserStarted AS us1, UserStarted AS us2
        select: us1.user, us2.user
        stop: DIFFERENCE us1.time ANY.time > 0.5%d

    JobTooSlow:
        trigger: JobInitiated AS ji, JobFinished AS jf
        select: jf.id, %DIFFERENCE ji.time jf.time AS duration
        where: DIFFERENCE jf.time ji.time > 5%d
            AND jf.id EQUALS ji.id
        stop: ANY.type EQUALS JobFinished%s
            AND ji.id EQUALS ANY.id
            AND NOT DIFFERENCE jf.time ji.time > 5%d
            # the type of an event is always available as a detail as well
```

To give a better impression of these features, we have included an example case definition file in Listing 4.2. The conditions and computed values we have implemented are listed in the Users Guide, starting with Section 4.5.1.

4.5 Users Guide

In this section, we will discuss how to use our explanation generation component. As a core feature, case definitions and how to define them has already been introduced. After a general introduction, we therefore continue with the available conditions, computed values and how to configure our console-based interpreter.

4.5.1 General Use

With the component installed correctly and at least a minimum configuration, simply run a Slingshot simulation and view the explanations given according to the interpreters used.

With our thesis, we have implemented a console-based interpreter that prints explanations to a new console in the target Eclipse Application used to run the simulation. Please note that the component is initialized during the first simulation start and the console is thus not yet present when the Eclipse Application is launched. When using different interpreters, please consult their documentation on how to access the respective output.

In defining the dynamically created rules and explanations, we see two different roles: The domain expert, responsible for encoding cases and setting up what information may be required to explain them. As well as the Human Computer Interaction (HCI) expert responsible for configuring how an Interpreter displays this information.

Available Conditions

Please note that we are using % as a special character and that <event> means the alias set for an event if it exists, otherwise its type. We are using %d to denote double constants and %s to denote String constants.

All of these are defined in the *MABEXStrings* class and possible changes can be reviewed there.

NOT <condition> Negates the following condition. Any condition can be used with the appropriate syntax.

DIFFERENCE <event>.<detail> <event>.<detail> > <double>%d Assumes that the passed details can be cast to Double. Computes their difference and compares it to the given constant.

<event>.<detail> EQUALS <event>.<detail> Checks the given details for equality.

<event>.<detail> EQUALS <String>%s Assumes that the given detail can be cast to String. Checks the given detail for equality with the given String. Hint: The type of an event is also always available as a detail.

<event>.<detail> < <event>.<detail> Assumes that the given details implement the Comparable interface. Checks whether the first detail is less than the second detail.

<event>.<detail> < <double>%d Assumes that the given detail can be cast to Double. Checks if it is less than the given constant.

<event>.<detail> > <event>.<detail> Assumes that the given details implement the Comparable interface. Checks whether the first detail is greater than the second detail.

<event>.<detail> > <double>%d Assumes that the given detail can be cast to Double. Checks if it is greater than the given constant.

Available Computed Values

Please note that we are using % as a special character and that <event> means the alias set for an event if it exists, otherwise its type. Meanwhile, <alias> refers to the key the computed value will be placed under.

%DIFFERENCE <event>.<detail> <event>.<detail> AS <alias> Assumes that the given details can be cast to Double. Computes their difference and places it under the given alias.

4.5.2 Writing Explanation Texts

For the console-based interpreter we have implemented with this thesis, a single YAML file can be configured that explanation texts will be sourced from. At the point of writing this thesis, the relative or total path to the explanation file has to be set in the *ConsoleBasedMABEXInterpreter* class' *explanationFileLocation* constant. The file is read in as a map of Strings with String keys. The texts may contain placeholders for data provided by the Builder in the following form: **${<selected value>}**. The <selected value> may either be an <event>.<detail> or the alias of a computed value as defined in the cases select statement. Note that <event> means the alias defined for an event if it exists and otherwise refers to its type. For longer explanation texts, we recommend becoming familiar with YAMLs block- and flow-style multi-line strings.

For different interpreters, please refer to their documentation for help in configuring them.

This answers the second part of our third research goal: How should a Palladio component access the required knowledge on explanations? (compare Chapter 1)

Assuming that the example case definition file given earlier (Listing 4.2) is used, here is an example explanation file:

Listing 4.3 An example explanation text file

```
#------------------------------------------------------------------------------------------
# Feel free to add comments such as this one for readability.
# Empty lines between different texts do not affect the parser either.
#------------------------------------------------------------------------------------------
UserFinishedEarly: The User with the ID ${UserFinished.user} finished before 10.0 simulation
        time.

UsersArriveTogether: The users ${us1.user} and ${us2.user} arrived within 0.5 simulation time
        of each other.

UsersArriveTogetherAlternative: "Really, the users ${us1.user} and ${us2.user}
        arrived within 0.5 simulation time of each other."

JobTooSlow: The job ${jf.id} was too slow, it took ${duration} simulation time to complete.
```

4.6 Changes we had to make

While it is usually better to reuse existing libraries, the fact that Palladio is based on the OSGi framework [OSG] limited our options in this regard. For our Analyzer, we were originally planning to base it on the Esper complex event processor developed by EsperTech Inc. [Esp]. However, we were unable to resolve some of Espers bundle dependencies that had changed between Java 8, which Esper targeted at the time of writing this thesis, and Java 11, which Palladio was based on at that time. We did however reuse their idea of extending SQL for encoding event patterns to make our language more user-friendly. This is also mentioned in Section 4.2. We additionally tried to base our Analyzer on Flink-CEP [Apa] but were unable to do so due to Flink using reflection to access its configuration. As OSGi uses multiple class loaders, this step failed to work with our setup.

During development, we ourselves also encountered limitations set by using OSGi as we were for example unable to automatically detect all existing Monitors using reflection, requiring a developer to manually make them known. Additionally, solutions for incomplete parts left to future work may already exist, but we could not find documentation on these issues or existing documentation turned out to be outdated.

5 Implementation

Our component offers experts a way to encode their knowledge into cases that require explanations without writing new code. Should the component lack a required function, most parts are expandable and its architecture allows for replacing complete modules should that not be enough. This allows it to additionally function as a tool for extracting other complex data from a Palladio simulation. We decided on these priorities because our understanding of different fields is naturally lacking compared to respective experts. A ready-made explanation generation component would have therefore been respectively limited.

As the framework we have based our implementation on, we are using MABEX [BGC+19] (see 4.3) as an identifying component in most of our class names.

In the following sections, we document implementation decisions and details on different core classes and modules. At the end, we also argue whether we have simply implemented another CEP option.

In the further development of this component, we generally see three roles: A Palladio expert responsible for adding further details to existing monitors and creating new ones for events that are not yet monitored. An Analyzer expert that takes care of expanding the set of available conditions and computed values as well as adding more parts to the case definitions as required. And finally an HCI expert that creates new Interpreters or adapts existing ones to fit new or changing use cases.

Our code has been placed into a Github repository at [Haa23a].

As a whole, this chapter answers our first research goal: How can a Palladio component implement an explainability approach? (compare Chapter 1).

5.1 General Structure

Our component is split into several bundles that match the modules depicted in Figure 4.1. Where feasible, we have concentrated the setup and configuration in the *core* bundles *ExplanationGeneration* class. It is registered as a Slingshot extension and therefore gets activated when a Slingshot simulation starts. More information on how to extend Slingshot can be found at [Sli] and in [Kat21]. As can be seen in Figure 4.1, *ExplanationGeneration* instantiates the actual processing modules, but does otherwise not participate in the explanation generation process. It is also responsible for linking up the other modules, allowing for them to be completely replaced here.

On the other end, the *data* bundle concentrates all intermediary formats and other required data classes. Most are only built to transport needed information from one module to the next, which is depicted in Figure 5.1. We will however go into more detail on the *MABEXStrings* class in Section 5.6, as it represents another option to easily adapt the component.

Figure 5.1: Dataflow during explanation generation

The explanation generation process is handled by the *monitor*, *analyzer*, *builder* and *interpreter* modules, which we will introduce in this order next.

5.2 Monitors

Each Monitor listens to a specific Slingshot Event (see 4.1). It then extracts details from the event and passes them to the Analyzer (depicted in Figure 5.1).

Apart from the type of event which is automatically added for identification, a Palladio expert can freely decide what other details should be extracted. With a different monitor for each event and no restrictions on the type of the details, any information that is accessible via the event can be passed on if needed. This answers our second research goal: For the chosen explainability approach, how should Palladio simulation data be prepared? (compare Chapter 1)

By convention, a monitor must also provide a list of the keys it uses when storing details. These lists are printed to console at the start of a simulation and provide a convenient overview of what events are currently getting listened to and what details are extracted from each event. This way, a domain expert looking to write new case definitions does not have to go through each monitor to check which information they can use in doing so. They can also quickly gauge whether and where they need to update or add a monitor to make the data they need available.

5.2.1 Adding and expanding Monitors

We have created a *TemplateMonitor* class that does not function as a Monitor itself but contains instructions on how to copy and adapt it into a new Monitor.

To make the process of adding new details to an existing Monitor less verbose, we have placed a large part of the required code into a convenience method and allow for method cascading.

5.3 Analyzer

The Analyzer is responsible for detecting the need for an explanation. It collects all necessary *EventWithData* and passes them to the Builder (see Figure 5.1).

With our thesis we have implemented the *MABEXAnalyzer* which offloads most of the required computations to potential cases. Figure 5.2 shows its workflow. If an event arrives that may be the first one in a defined case (see Section 4.4), a potential case is created. Afterwards, all potential cases check if the event is the next one they need. This allows for discarding them if the first event already did not meet some conditions without the analyzer itself having to check these conditions.

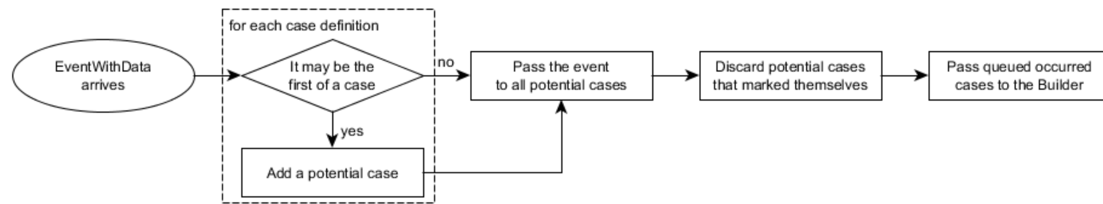

Figure 5.2: MABEXAnalyzer: workflow

5.3.1 Potential Cases

A potential case progresses through the events defined in a specific case definition (see Section 4.4). Upon receiving an event, it first checks whether the cases *stop* conditions have been met so it can be discarded early. This means a performance increase, but a potential case may be discarded on receiving the last event it needed. We chose to accept this trade-off as it can be clearly communicated and then worked around. The potential case then checks the conditions defined in the cases *where* statement and remembers the event if it passes them.

For each condition, a potential case first checks if it can actually be evaluated with the currently available events. Skipped conditions are assumed true, while skipped stop conditions are assumed false.

5.3.2 Conditions

MABEXConditions provide the functionality behind any condition usable in the *where* and *stop* parts of a case definition (see Section 4.4).

We separated them into extra classes so they can be dynamically used according to each case definition. If a required condition is not yet available, it is also simpler to add them.

Their superclass is currently responsible for decoding the conditions given in a case definition. It uses a decision tree to determine the correct Condition according to their keywords detailed in Section 4.5.1.

5.4 Builder

The Builder is responsible for generating machine-readable Explanations from the events relevant to a case (see Figure 5.1). It is not to be confused with the builder pattern [GHJ+95] and instead refers to MAB-EX (see Section 4.2). The Builder filters out only the details and computed values defined in a cases *select* statement (see Section 4.4) as a flat map.

With our thesis, we have implemented a simple Builder that passes on details as they are and outsources computed value calculation. The classes that perform this computing step are detailed in the following subsection while Figure 5.3 shows this process graphically.

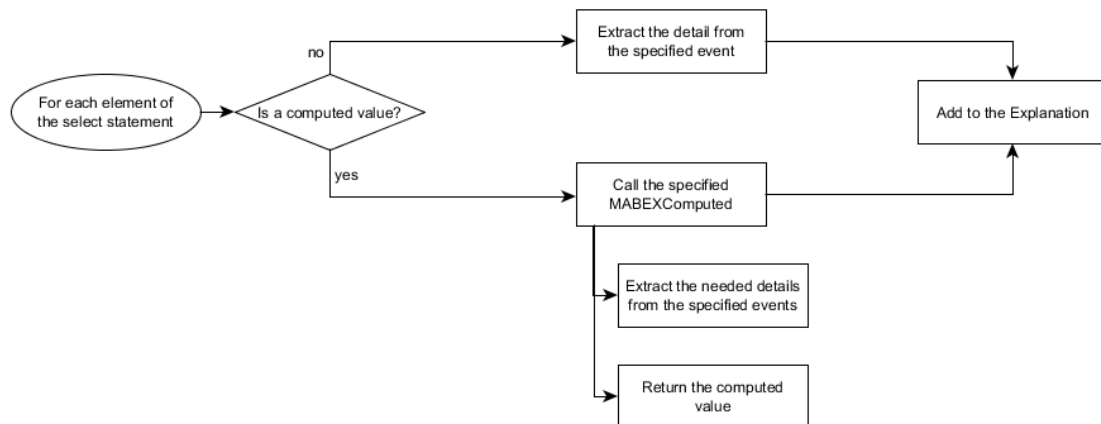

Figure 5.3: Building an Explanation

5.4.1 Computed Values

MABEXComputed are responsible for computing derived values for the Builder. The Computed available with this thesis are detailed in Section 4.5.1. As we had similar requirements for them as we had for the Conditions (Section 5.3.2), their structure is much the same.

They are separated into different classes to be used dynamically and to simplify adding additional ones. Their superclass is currently responsible for decoding the computed values given in a case definition.

5.5 Interpreters

An Interpreter in this component is a data sink that takes Explanations and turns them into a human-readable output or a data format required for further processing.

For this thesis, we have implemented a simple console-based Interpreter that gives freely definable but fixed explanation texts in an extra console view of the target eclipse platform. It is described in the following subsection.

As the Interpreter only acts as a data sink however, it is easily exchangeable depending on what form of output is required for a particular use case. For example, new Interpreters could log specific event details to a file or plot numerical values into a diagram as the simulation runs.

5.5.1 The Console-based Interpreter

This Interpreter can output expert-defined texts in a new console of the target eclipse platform the simulation is ran on. It loads these texts from a YAML file (see Section 4.5.2) and matches their keys with the keys used for defining the case definitions for the analyzer (see Section 4.4).

Please note that Slingshot is currently lacking support for passing YAML files via a launch configuration. Because of this, the path the interpreter grabs its explanation definitions from needs to be set via the *explanationFileLocation* constant.

As the Builder asks all Interpreters to update their definitions on simulation start, there is no need to restart the target Eclipse Application if only the content of the explanation file has changed.

5.6 MABEXStrings

While most of our data classes are simple containers, this class acts as a central storage for used keywords. They are split into detail keys used by Monitors, common delimiters as well as identifiers for conditions (see Section 4.5.1 and Section 5.3.2) and computed values (see Section 4.5.1 and Section 5.4.1). The class also offers static methods for making often used operations on these strings less verbose.

Having all of these strings available in a single class makes it much faster and less error-prone to change them as needed. Where possible, they also refer to each other, so changing the special character we use to mark constants or computed values (currently %) automatically changes all relevant strings.

With the support of an IDE and due to our classes using the provided utility methods, it also becomes possible to e.g. change from commas as delimiters to semicolons, complete with a renamed convenience method in only a few operations.

5.7 Did we re-implement CEP?

As mentioned in Section 4.2, our case definition language is inspired by Espertechs EPL [Esp]. In this section we would like to argue whether this means that we simply implemented another less developed CEP option.

On one hand, our component is used for very similar tasks. It searches for patterns in Slingshots event stream and reacts to them. It filters the events according to conditions attached to the pattern and aggregates them into a single case. Similar to complex or composite events, a case is also inferred from an event pattern rather than directly detected. Our component also extracts and derives values from data contained in the events. Additionally, the fact that we tried to use Esper (see Section 4.6) in the first place is another point.

On the other hand, there are differences in the limitations and the motivation. CEP is used to find patterns in real-time, while our component runs on simulated time. It can therefore use more expensive computations or query for additional information if needed. Given that our component is operating in a fixed system, we are also already aware of any event hierarchies and do not need to find them. Next, there is currently only a single stream of events in Slingshot, while CEP may have to react to many different sources. Lastly, CEP usually results in a composite event that can then be reacted to. Our component however does this part as well via the Interpreters.

Concluding both sides of the argumentation, we propose that our component is not reimplementing CEP, but rather represents another leaf on the same branch. In some parts, our component does less than CEP as limitations and requirements are less strict but it also goes another step that is usually not done in CEP. While the core concepts and thus most of the steps taken are similar, the motivations and limitations are different. This may also become more apparent as our component could get expanded to query model information in the future (see Section 7.5).

6 Evaluation

In this chapter, we will evaluate our component. For this, we have performed a user survey on the importance of questions an explanation generation component should be able to answer. We use the questions we have gathered with our scenario sheets (see Chapter 3). They are listed in the next section together with details on the design of the survey.

6.1 User Survey Design

We used the scenario sheets detailed in Chapter 3 to gather a set of questions our explanation generation component should ideally be able to answer. These served as requirements during development, highlighting what information may be needed and what connections the component should be able to draw. To make them a better indicator for evaluating our component however, we designed a survey and asked participants to rank the importance of each question and to suggest additional ones.

As this was going to be a small survey only, we created it as a single PDF document with inbuilt form fields. The document was created via LaTeX using the *hyperref* package. It starts by introducing the participant to the thesis and what is asked of them. We then added a privacy policy to explain what will happen to the data we collected. Apart from ranking the existing and potential new questions, we only asked the participants to indicate how much, if any, experience they have with Palladio. To make it easier to understand for participants less familiar with the topics discussed, we then go over important terms. Before we gave the questions to rank, we started with describing a scenario of modeling a self-adaptive system in Palladio. With this, we wanted to give the participant an environment they could use to more easily decide on the importance of each question.

For ranking the questions, we used a 5-valued Likert-scale of 1 (not important) to 5 (very important). For other questions a participant would like to see answered, we provided several blank fields with Likert scales as well as a large text field.

The questions themselves were divided by topics:

- Thresholds
  - Which threshold was crossed?
  - When was the threshold reached?
  - What was the threshold value?
- Reaction/Scaling
  - What reaction was caused?

- When did the reaction end?
- When was the updated capacity available?
- What was scaled?

- SLOs
  - Which SLO was violated?
  - How long was the SLO violated and when?
  - What was the agreed upon threshold for each SLO?

On one hand, we intuitively used topics with our scenarios to generate these questions. On the other hand, they make it easier for the participant to decide if they would ask additional questions as well. The topics also give more structure to the survey document.

The results of our survey are used and detailed in the following section while you can find the filled surveys and the original document on Github at [Haa23b].

6.2 Results

We have tallied up the results of our user survey in Table 6.1.
Figure 6.1 shows a box plot of the results, although only the question number has been included for readability. For each question, the darker line marks the median importance value. The box ranges from the first to the third quartile while whiskers show outliers.

In total, we had twelve participants. As we did not collect identifying data beyond their experience with Palladio and computer science, we have simply numbered them in the first line of the table. We have rounded the average values to the nearest decimal for ease of reading. They will be used as an importance indicator for each question. We also shortened “I am” and “yes” to “y” as well as “I am not” and “no” to “n” to fit the table on a single page. The sum of all average values at the bottom will be compared to the sum of the average values of answerable questions in the next section.

New questions suggested by participants have been weighted less according to the limited sample size. Their importance was treated as zero for participants that did not rank them. They are separated by our given questions using a horizontal line. Additionally, some of the new questions were given as comments or could be interpreted in several ways. We have replaced these with more specific questions as we interpreted them. These adapted questions are marked with a * while we combined two similar questions into one marked with ** twice.

Apart from new questions, we also received an additional comment that we will include in the discussion without an attached weight.

The full anonymous surveys with the original questions have been uploaded to Github at [Haa23b].

Question	User	1	2	3	4	5	6	7	8	9	10	11	12	Avg
Q1: Which threshold was crossed?		5	5	5	5	4	4	5	5	4	5	4	3	4,5
Q2: When was the threshold reached?		3	4	4	3	4	5	5	5	3	3	5	5	4,1
Q3: What was the thresholds value?		4	2	3	4	4	3	4	3	5	3	4	3	3,5
Q4: What reaction was caused?		5	5	5	5	4	5	5	5	3	5	4	4	4,6
Q5: When did the reaction end?		3	2	3	2	3	4	5	3	2	3	4	3	3,1
Q6: When was the updated capacity available?		2	4	3	3	4	4	3	5	5	4	5	4	3,8
Q7: What was scaled?		5	5	5	4	4	3	4	4	5	5	5	4	4,4
Q8: Which SLO was violated?		5	5	5	5	5	5	4	5	4	4	5	5	4,8
Q9: How long was the SLO violated and when?		3	4	5	4	4	4	3	4	5	5	5	5	4,3
Q10: What was the agreed upon threshold for each SLO?		4	3	5	3	4	4	3	4	4	3	3	4	3,7
Q11: **What was the velocity of the monitored metric upon crossing the threshold?			4						3					0,6
Q12: *Are there other thresholds that will be crossed at the current velocity?			3											0,3
Q13: *Was the reaction large enough to satisfy the SLOs again?			5											0,4
Q14: **How close to being violated is the SLO after scaling?			5						4					0,8
Q15: *How large was the monitored metric relative to the threshold it crossed?				3										0,3
Q16: Which rule(s) triggered the reaction?								4						0,3
Q17: How often was the threshold reached?									4					0,3
Q18: In which direction was scaled (horizontal/vertical)?									3					0,3
Q19: *Was the reaction executed without errors?									3					0,3
Q20: How much was the SLO violated?									5					0,4
Sum of all average values:														44,8
Are you researching or working in the field of computer science?		y	y	y	n	n	y	y	y	n	y	y	y	
Have you worked with Palladio before?		n	n	n	n	n	n	y	n	n	n	n	n	
If yes, how many years of experience do you have with Palladio?		-	-	-	-	-	-	1	-	-	-	-	-	

Table 6.1: Survey Results: *-marked questions have been adapted, **-marked questions have been combined and adapted

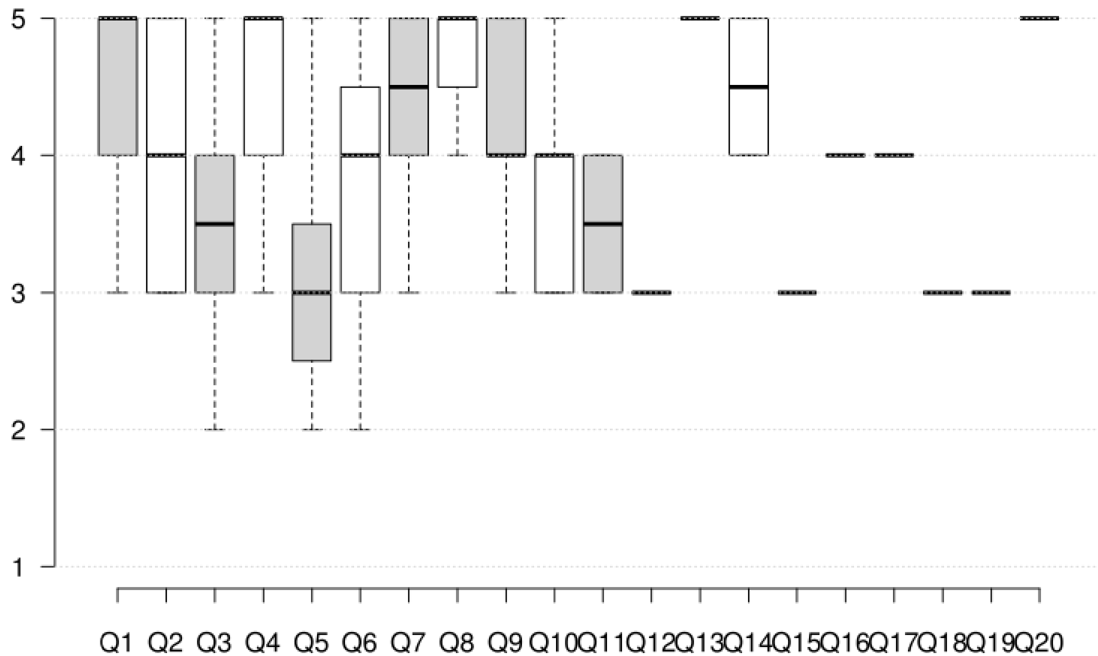

Figure 6.1: Survey Results as a box and whiskers plot

6.2.1 Participant Distribution

In this section, we want to briefly discuss our selection of participants.

While we chose to keep them anonymous as personal information is not relevant for our results and to respect their privacy, we contacted roughly the same amount of researchers and people from the industry. To generalize our results further, we also included people that were not related to the field of computer science. Apart from these, Table 6.1 shows that only one participant indicated that they have experience with Palladio. We do not see this as a problem however. If at all, we may have been able to perform the same survey while excluding any mention of Palladio and assume that we would have still gotten similar results. We would like to argue that between a simulated autoscaling system and one that is already implemented, the relative importance of desired information does not change. As for the total importance, in a live system that runs into the problems mentioned, the given questions will likely be deemed more important than for a simulated system at design time.

6.2.2 General Results

Before we use them to evaluate our component in the next section, we would like to first inspect the survey results themselves.

Reading Figure 6.1, we would argue that our selection of questions was correct as a whole. Of the ten questions we gave, only two have received a median importance value lower than four. Three of them have a median importance value of five, the highest value in the scale. No question was

deemed as non-important by any participant, as in, the lowest importance value given was a two. We also believe that it was a good choice to allow participants to add their own questions given that some participants used this option heavily and thus helped in further adding to our requirements.

While our survey was small and specific to the topic of autoscaling, we conclude that it can still prove useful to similar projects due to these results. They also affirm that the scenario-based approach we used was well-suited for our project.

6.3 Discussion

We will now discuss whether our component is able to answer each of the questions given in Table 6.1. As a metric for evaluating our component, they have been assigned the average importance value as seen in the last column of the table. At the end of this section, we will sum these up for the questions we are able to answer and compare this sum to the sum of all averages as seen in Table 6.1. For each question, we will either give an example case definition that will provide the necessary details or argue why it is currently not possible to do so. Where possible, we will combine multiple questions into a group with a single case definition or reasoning.

At the time of writing this thesis, the Scaling Policy Definition (SPD) Meta Model and the Slingshot simulator were in active development. It is therefore possible that information has moved or that previously unavailable details have been exposed by now. We list the relevant commits we developed against and those we determined these results against in our Github code repository at [Haa23a].

Question Group 1

Q1: Which threshold was crossed? Weight: 4,5
Q3: What was the thresholds value? Weight: 3,5
Q15: *How large was the monitored metric relative to the threshold it crossed? Weight: 0,3
Q16: Which rule(s) triggered the reaction? Weight: 0,3

Slingshot defines thresholds using triggers that activate when the threshold is crossed and adapt the simulated system. However, Slingshot does currently not expose a trigger event. Instead, it checks the triggers following a *MeasurementMade* event and directly makes the changes. Our component can therefore not access the information required to answer these questions. Instead, we have collected these and other requirements in Section 6.3.2 for the developers of Slingshot.

Question Group 2

Q2: When was the threshold reached? Weight: 4,1
Q4: What reaction was caused? Weight: 4,6
Q7: What was scaled? Weight: 4,4
Q18: In which direction was scaled (horizontal/vertical)? Weight: 0,3

Listing 6.1 Threshold crossed case definition

```
ThresholdCrossed:
    trigger: MeasurementMade as mm, StepBasedAdjustor as adj
    select: mm.time, adj.targetgroup, adj.step, adj.time, adj.delay
    stop: DIFFERENCE mm.time ANY.time > 0.1%d
```

While we cannot yet get information on the threshold itself, we can infer when a threshold was crossed by detecting the *StepBasedAdjustor* event as in Listing 6.1. It is the only currently implemented event that represents changes made to the simulated system because of a threshold trigger. It scales horizontally by adding or removing resource containers. The *targetgroup* can be used to acquire the resource that was scaled, while *step* represents how many containers where added or removed. As these events should follow directly after each other if there have been changes, we *stop* the case again after a short time.

Question Group 3

Q5: When did the reaction end? Weight: 3,1
Q6: When was the updated capacity available? Weight: 3,8

Listing 6.2 Model changed case definition

```
ModelChanged:
    trigger: ModelChanged AS mc
    select: mc.time
```

A *ModelChanged* event summarizes the changes made to the simulated system. We can therefore use it as in Listing 6.2 to learn when the reaction ended. Model changes are currently simulated instantaneously, so the capacity is also directly available.

Question Group 4

Q8: Which SLO was violated? Weight: 4,8
Q9: How long was the SLO violated and when? Weight: 4,3
Q10: What was the agreed upon threshold for each SLO? Weight: 3,7
Q13: *Was the reaction large enough to satisfy the SLOs again? Weight: 0,4
Q14: **How close to being violated is the SLO after scaling? Weight: 0,8
Q20: How much was the SLO violated? Weight: 0,4

While Palladio allows for defining SLOs, Slingshot does not currently link triggers defined in the SPD to them. Additionally, looking up the correct SLO would likely require inspecting the simulated system, which is an additional hurdle.

Question Group 5

Q17: How often was the threshold reached? Weight: 0,3
Q11: **What was the velocity of the monitored metric upon crossing the threshold? Weight: 0,6
Q12: *Are there other thresholds that will be crossed at the current velocity? Weight: 0,3

These questions would require querying the past of the simulated system, which we currently do not support. It may be possible to answer the first and second question with an Interpreter that can make use of the stored events after a simulation has ended. For answering the third question however, we would need to inspect the simulated system, wich is an additional hurdle as mentioned earlier.

Question Group 6

Q19: *Was the reaction executed without errors? Weight: 0,3

To answer this question, we would require that Slingshot exposes the success or failure of a model adjustment. This could for example be done in the *ModelChanged* event used in Listing 6.2.

6.3.1 Component Evaluation

In total, our component is currently able to answer 6 out of 20 questions, with a total weight of 20,3 out of 44,8. While this result seems sub-par at first, we would like to highlight that the reasons are largely only twofold. On one hand, we have not implemented the possibility of inspecting the simulated model. On the other hand, Slingshot in its current state of development simply does not expose or use information required for a large part of these questions. This means that they represent new requirements for the Slingshot simulator, which we detail in the following subsection.

We would also like to argue that our component is built with extensibility in mind. Features that are currently lacking can be added. The case definitions are not a fixed construct. If the basic toolbox offered by our component fits a problem, it can likely evolve to solve the problem.

Overall, we were still able to answer all three research goals (compare Chapter 1) we set at the start of this thesis.

Additional Comment

Apart from new questions, one participant added an additional comment that we would like to address. They asked for a graphical explanation instead of a textual one. It should show the modeled system before and after the adaptation as well as highlight the differences between them.

We currently cannot do this. However, this is simply because we only implemented a console-based Interpreter. The necessary details can already be passed on, we are only lacking a graphical Interpreter that presents them as described.

To us, this affirms that our component would be well-suited as a basis for offering different explanations in the context of Palladio.

6.3.2 Resulting Requirements for the Slingshot simulator

This subsection presents the requirements for the Slingshot simulator that have followed from Section 6.3. The concepts behind the Slingshot simulator are documented at [Sli] and it is available at [Pal].

If a trigger fires, Slingshot should produce a corresponding event that references the relevant Scaling Policy and the measured value. This would allow our component to directly answer the first group of four questions. The described event could also be required if SLOs set in Palladio are linked to the Scaling Policies. To report on their (non-)violation would require knowing when or if the corresponding thresholds were crossed.

This brings up the second topic: To avoid querying the simulated model, a trigger event that is relevant for an SLO should result in an SLO event that references the trigger and the corresponding SLO.

To answer whether there are additional endangered thresholds, our component would also require that the target group a trigger references can be queried for other Scaling Policies that target it.

Finally, a *ModelChanged* event or those representing a single change in the model should report on the success of each change. On the other hand, the reasons for a failure can likely already be calculated from the relevant Scaling Policy.

6.4 Threats to Validity

In this section, we will discuss possible threats to the validity of our results. As part of this, we will also address the threat classification as per Runeson and Höst [RH09] that encompasses four different aspects.

There is a point we would like to note first: While the weighted importance metric we used seems sound to us, we are not aware of a scientific publication that proves its suitability as a reference for an explanation generation components effectiveness.

Next, we will address the threat classification from [RH09].

Construct Validity

This aspect concerns itself with how accurately what was studied matches what the researcher had in mind and their research questions [RH09]. We may suffer from this aspect somewhat. While our survey gives a brief introduction of Palladio, SAS and other relevant terms, participants may have different previous knowledge on these topics. This leads to them having different viewpoints on the described scenario which may not match our viewpoint. However, the survey goal was for participants to rank the importance of different questions. While more experienced participants may answer differently than those unfamiliar to the topic at hand, we argue that this was sufficiently conveyed through the design of the survey.

Internal Validity

This aspect concerns itself with possible external factors that affect the investigated subject [RH09]. We cannot guarantee that the order in which participants were asked to rank each question may have affected their view of the later ones, but we believe this effect to be marginal at best. Furthermore, we have surveyed people from different institutes and companies, including some that were less familiar with the field of computer science as a whole (compare Section 6.2.1). As such, we believe that there should be no meaningful external factors that could affect our results as a whole.

External Validity

This aspect concerns itself with how well our findings can be generalized and whether they may be of interest to other people [RH09]. We performed a small survey using questions specific to the topic of autoscaling SAS. As such, we can not claim that our findings are representative for a larger population. The questions ranked are also specific to the field of providing information to users or developers of autoscaling systems. Nonetheless, they may be of use to people working on the explainability of already existing systems where making use of Palladio may not be desired or possible.

Reliability

This aspect concerns itself with how reproducible a study is [RH09]. We can not claim that our survey is completely reproducible as our participants are anonymous and may change their opinions over time as well. However, our survey itself consists of a fixed document that another researcher can hand out in much the same way we did. As such, we believe that our results would not have been different if someone else had performed the survey.

7 Conclusion

In this chapter, we will summarize our thesis and go over these closing topics: We will point out who specifically we believe will benefit from our contributions. Next, we will expose limitations our current work has and what lessons we have learned over the course of our thesis. Finally, we will give our ideas on future work branching off from our thesis.

7.1 Summary

With this thesis, we have implemented an expert system that extends Palladio and allows for explaining patterns in the occurring events. The patterns as well as the explanations given by our console-based interpreter can be dynamically defined in YAML files. The component is easy to extend and modular enough to allow for other use cases that require extracting complex information from Slingshots event stream.

Additionally, we suggested a variant of the explainability scenarios by Wolf [Wol19] that incorporates requirements usually imposed on studies and a defined structure. We also performed a short survey on the importance of questions an explanation generation component should be able to answer in the field of autoscaling web applications.

7.2 Benefits

With this thesis, we set out to provide one possible solution for Palladio users that require additional explanations on simulations of complex software such as self-adaptive systems. While the component cannot answer all questions we have collected, we believe it will definitely be an aid in those cases. We also believe that the dynamic and customizable way we have implemented our component in can serve as a good foundation for similar projects. For the same reasons, we believe the component can also serve as a quick customizable logging tool for Palladio developers. The component is a non-intrusive extension that can be deployed as needed and the ability to change Interpreters as required should be beneficial here as well. Additionally, the system of dynamically usable conditions and computed values may be usable for other components requiring user-defined logic.

7.3 Limitations

Our component currently has some limitations that may be more simple to solve, as well as some that would require more large scale work to get around.

First, we could not find a good way to encode optional components into cases. Our idea was that they could be used to define possible reasons for a case occurring and similar relations. It also does not support the full extent of logical operators when linking different conditions.

On a larger scale, our component can currently only use information that is provided with each event and potentially those prior to it. However, some answers require information on the relations of different entities in the simulated model that we cannot get this way. We also face the same limitation that other expert systems have in that we can only explain what an expert encoded beforehand.

Here, we would also like to refer back to Section 4.6, where we described changes made due to OSGi. While they were important for our design and have thus been placed in Chapter 4, they represent limitations as well.

7.4 Lessons Learned

We found that implementing a new language adds an additional layer of complexity on top of more static coding, as one has to think about not only the end user and the maintainer. There is also the ambition that an expert configuring the system should be able to encode what they envision, as well as that they can do so smoothly. Both of these aspects caused us to rework multiple aspects of our component and to drop some features for a lack of good ideas on how they could be encoded properly.

We also found that the structured approach to defining requirements for our explanation generation component detailed in Chapter 3 provided anchors that made this process much easier. We thus recommend that anyone having to define similar requirements should first look for a structured approach that suits their needs.

7.5 Future Work

If it proves as useful as we think it will, there is much to do in the future of this component. On one side, its existing structure needs to be fleshed out. On the other side there are many possible points that could be expanded upon depending on how requirements change.

First, the sets of conditions and computed values still lack some elements that would likely be used often. As we implemented computed values after the conditions, it may also be worth exploring whether the former should be usable in the later. Additionally, both conditions and computed values currently use a simple tree of conditions to determine the correct instance to create from a String. Depending on how complex parameters are required in the future, a more sophisticated decoding logic may be desirable. Next, we are not yet monitoring all events Slingshot is able to create and for most, we only pass on a limited set of details. Given that there are also new events being developed, we foresee an ongoing need to maintain the different monitors. There are also some options or modifiers that we think may see common use if they were added. For example, an option to make the event chain strict instead of allowing other events in between. Lastly, we were unable to extract the locations of YAML files from our launch configuration tab, leaving it non-functional. Locating and implementing the missing link would allow storing and automatically using a case definition file together with the corresponding launch configuration.

On a wider scope, we also see several more or less ambitious possibilities of expanding our component. For different use cases, new Interpreters could be implemented. They could provide explanations on demand instead of directly, visualize certain data in graphs as the simulation runs or create a report file that can be directly passed to stakeholders among many other possibilities. On the other end, we are currently only accessing information from the events occurring during a simulation. For some of the questions we evaluated our component with however, we already mentioned that accessing the model itself may be required. As such, another larger project could be developing how to encode and access model information in case definitions. Similarly, we did not manage to devise a good way to encode optional relations into cases that could be used to explain e.g. potential reasons for the case. If the component finds more widespread use, we also suggest that a library of case definitions may be useful to share them between users. Finally, if requirements change significantly the Analyzer and or Builder may have to be replaced completely.

Bibliography

[Apa] Apache Software Foundation. *FlinkCEP - Complex event processing for Flink*. URL: https://nightlies.apache.org/flink/flink-docs-release-1.17/docs/libs/cep/ (cit. on p. 18).

[BD10] R. Bruns, J. Dunkel. *Event-driven architecture: Softwarearchitektur für ereignisgesteuerte Geschäftsprozesse*. Springer-Verlag, 2010 (cit. on pp. 4, 5).

[BDG+09] Y. Brun, G. Di Marzo 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*. Ed. by B. H. C. Cheng, R. de Lemos, H. Giese, P. Inverardi, J. Magee. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 48–70. ISBN: 978-3-642-02161-9. DOI: 10.1007/978-3-642-02161-9_3. URL: https://doi.org/10.1007/978-3-642-02161-9_3 (cit. on p. 4).

[BGC+19] M. Blumreiter, J. Greenyer, F. J. Chiyah Garcia, V. Klös, M. Schwammberger, C. Sommer, A. Vogelsang, A. Wortmann. "Towards Self-Explainable Cyber-Physical Systems". In: *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. 2019, pp. 543–548. DOI: 10.1109/MODELS-C.2019.00084 (cit. on pp. 5, 11, 12, 19).

[BKR09] S. Becker, H. Koziolek, R. Reussner. "The Palladio component model for model-driven performance prediction". In: *Journal of Systems and Software* 82.1 (2009). Special Issue: Software Performance - Modeling and Analysis, pp. 3–22. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2008.03.066. URL: https://www.sciencedirect.com/science/article/pii/S0164121208001015 (cit. on pp. 1, 4).

[BLV+19] A. Bauer, V. Lesch, L. Versluis, A. Ilyushkin, N. Herbst, S. Kounev. "Chamulteon: Coordinated Auto-Scaling of Micro-Services". In: *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. 2019, pp. 2015–2025. DOI: 10.1109/ICDCS.2019.00199 (cit. on pp. 1, 10).

[CM12] G. Cugola, A. Margara. "Processing flows of information: From data stream to complex event processing". In: *ACM Computing Surveys (CSUR)* 44.3 (2012), pp. 1–62 (cit. on p. 5).

[DW21] E. Dai, S. Wang. "Towards Self-Explainable Graph Neural Network". In: *Proceedings of the 30th ACM International Conference on Information and Knowledge Management*. CIKM '21. Virtual Event, Queensland, Australia: Association for Computing Machinery, 2021, pp. 302–311. ISBN: 9781450384469. DOI: 10.1145/3459637.3482306. URL: https://doi.org/10.1145/3459637.3482306 (cit. on p. 1).

[Esp] EsperTech Inc. *Esper - EsperTech*. URL: https://www.espertech.com/esper/ (cit. on pp. 12, 18, 23).

[GC04] K. Go, J. M. Carroll. "The Blind Men and the Elephant: Views of Scenario-Based System Design". In: *Interactions* 11.6 (Nov. 2004), pp. 44–53. ISSN: 1072-5520. DOI: 10.1145/1029036.1029037. URL: https://doi.org/10.1145/1029036.1029037 (cit. on p. 6).

[GHJ+95] E. Gamma, R. Helm, R. Johnson, R. E. Johnson, J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Deutschland GmbH, 1995 (cit. on pp. 9, 21).

[Goo] Google. *Google Scholar*. URL: https://scholar.google.com/ (cit. on p. 3).

[Haa23a] J. Haas. *Explanation Generation Code Repository*. 2023. URL: https://github.com/Ulkusus/Palladio-Addon-ExplanationGeneration (cit. on pp. 19, 29).

[Haa23b] J. Haas. *Explanation Generation User Surveys Repository*. 2023. URL: https://github.com/Ulkusus/ExplanationGeneration-UserSurvey (cit. on p. 26).

[HKWT18] A. Holzinger, P. Kieseberg, E. Weippl, A. M. Tjoa. "Current Advances, Trends and Challenges of Machine Learning and Knowledge Extraction: From Machine Learning to Explainable AI". In: *Machine Learning and Knowledge Extraction*. Ed. by A. Holzinger, P. Kieseberg, A. M. Tjoa, E. Weippl. Cham: Springer International Publishing, 2018, pp. 1–8. ISBN: 978-3-319-99740-7 (cit. on p. 4).

[IAA20] M. Imdoukh, I. Ahmad, M. G. Alfailakawi. "Machine learning-based auto-scaling for containerized applications". In: *Neural Computing and Applications* 32.13 (July 2020), pp. 9745–9760. ISSN: 1433-3058. DOI: 10.1007/s00521-019-04507-z. URL: https://doi.org/10.1007/s00521-019-04507-z (cit. on p. 5).

[Jac86] P. Jackson. *Introduction to expert systems*. Jan. 1986. URL: https://www.osti.gov/biblio/5675197 (cit. on p. 5).

[JWNS] C. Jones, J. Wilkes, M. Niall, C. Smith. *Site Reliability Engineering - Service Level Objectives*. Google. URL: https://sre.google/sre-book/service-level-objectives/ (cit. on p. 5).

[Kat21] J. Katić. *Event-driven simulator for the Palladio Component Model*. B.S. thesis. 2021 (cit. on pp. 1, 4, 19).

[KKB21] J. Katić, F. Klinaku, S. Becker. "The Slingshot Simulator: An Extensible Event-DrivenPCM Simulator". In: 2021 (cit. on pp. 1, 4).

[MHdH13] F. D. Macías-Escrivá, R. Haber, R. del Toro, V. Hernandez. "Self-adaptive systems: A survey of current approaches, research challenges and applications". In: *Expert Systems with Applications* 40.18 (2013), pp. 7267–7279. ISSN: 0957-4174. DOI: https://doi.org/10.1016/j.eswa.2013.07.033. URL: https://www.sciencedirect.com/science/article/pii/S0957417413005125 (cit. on p. 4).

[OSG] OSGi Working Group. *OSGi, The Dynamic Module System for Java*. URL: https://www.osgi.org/resources/what-is-osgi/ (cit. on pp. 13, 18).

[Pal] Palladio. The Quality Software People. *The Palladio Simulator Github Organization*. URL: https://github.com/PalladioSimulator (cit. on p. 32).

[Pal22] Palladio. The Quality Software People. *Palladio Software Architecture Simulator*. 2022. URL: https://www.palladio-simulator.com/home/ (cit. on pp. 1, 4).

[PGB21] J. M. Parra-Ullauri, A. García-Domínguez, N. Bencomo. "From a Series of (Un)fortunate Events to Global Explainability of Runtime Model-Based Self-Adaptive Systems". In: *2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. 2021, pp. 807–816. DOI: 10.1109/MODELS-C53483.2021.00127 (cit. on pp. 1, 6, 13).

[RH09] P. Runeson, M. Höst. "Guidelines for conducting and reporting case study research in software engineering". In: *Empirical Software Engineering* 14.2 (Apr. 2009), pp. 131–164. ISSN: 1573-7616. DOI: 10.1007/s10664-008-9102-8. URL: https://doi.org/10.1007/s10664-008-9102-8 (cit. on pp. 32, 33).

[RMMH22] R. R. Fernández, I. Martín de Diego, J. M. Moguerza, F. Herrera. "Explanation sets: A general framework for machine learning explainability". In: *Information Sciences* 617 (2022), pp. 464–481. ISSN: 0020-0255. DOI: https://doi.org/10.1016/j.ins.2022.10.084. URL: https://www.sciencedirect.com/science/article/pii/S002002552201204X (cit. on pp. 7, 13).

[SF20] K. Sokol, P. Flach. "Explainability Fact Sheets: A Framework for Systematic Assessment of Explainable Approaches". In: *Proceedings of the 2020 Conference on Fairness, Accountability, and Transparency*. FAT* '20. Barcelona, Spain: Association for Computing Machinery, 2020, pp. 56–67. ISBN: 9781450369367. DOI: 10.1145/3351095.3372870. URL: https://doi.org/10.1145/3351095.3372870 (cit. on p. 7).

[Shi21] D. Shin. "The effects of explainability and causability on perception, trust, and acceptance: Implications for explainable AI". In: *International Journal of Human-Computer Studies* 146 (2021), p. 102551. ISSN: 1071-5819. DOI: https://doi.org/10.1016/j.ijhcs.2020.102551. URL: https://www.sciencedirect.com/science/article/pii/S1071581920301531 (cit. on p. 4).

[Shu05] Shu-Hsien Liao. "Expert system methodologies and applications—a decade review from 1995 to 2004". In: *Expert Systems with Applications* 28.1 (2005), pp. 93–103. ISSN: 0957-4174. DOI: https://doi.org/10.1016/j.eswa.2004.08.003. URL: https://www.sciencedirect.com/science/article/pii/S0957417404000934 (cit. on p. 5).

[Sli] Slingshot Project. *Slingshot Project Documentation*. URL: https://palladiosimulator.github.io/Palladio-Documentation-Slingshot/ (cit. on pp. 19, 32).

[SLJ+21] A. Simkute, E. Luger, B. Jones, M. Evans, R. Jones. "Explainability for experts: A design framework for making algorithms supporting expert decisions more explainable". In: *Journal of Responsible Technology* 7-8 (2021), p. 100017. ISSN: 2666-6596. DOI: https://doi.org/10.1016/j.jrt.2021.100017. URL: https://www.sciencedirect.com/science/article/pii/S266665962100010X (cit. on pp. 7, 13).

[Sut03] A. Sutcliffe. "Scenario-based requirements engineering". In: *Proceedings. 11th IEEE International Requirements Engineering Conference, 2003.* 2003, pp. 320–329. DOI: 10.1109/ICRE.2003.1232776 (cit. on p. 6).

[USCG05] B. Urgaonkar, P. Shenoy, A. Chandra, P. Goyal. "Dynamic Provisioning of Multi-tier Internet Applications". In: *Second International Conference on Autonomic Computing (ICAC'05)*. 2005, pp. 217–228. DOI: 10.1109/ICAC.2005.27 (cit. on pp. 1, 10).

[VL21] G. Vilone, L. Longo. “Notions of explainability and evaluation approaches for explainable artificial intelligence”. In: *Information Fusion* 76 (2021), pp. 89–106. ISSN: 1566-2535. DOI: https://doi.org/10.1016/j.inffus.2021.05.009. URL: https://www.sciencedirect.com/science/article/pii/S1566253521001093 (cit. on p. 4).

[WAM+21] C. S. Wickramasinghe, K. Amarasinghe, D. L. Marino, C. Rieger, M. Manic. “Explainable Unsupervised Machine Learning for Cyber-Physical Systems”. In: *IEEE Access* 9 (2021), pp. 131824–131843. DOI: 10.1109/ACCESS.2021.3112397 (cit. on p. 1).

[Wol19] C. T. Wolf. “Explainability Scenarios: Towards Scenario-Based XAI Design”. In: *Proceedings of the 24th International Conference on Intelligent User Interfaces*. IUI ’19. Marina del Ray, California: Association for Computing Machinery, 2019, pp. 252–257. ISBN: 9781450362726. DOI: 10.1145/3301275.3302317. URL: https://doi.org/10.1145/3301275.3302317 (cit. on pp. 6, 9, 35).

[Yam] Yaml Project. *The Official Yaml Website*. URL: https://yaml.org/ (cit. on p. 13).

[Zil22] M. Zilch. “Evaluation of Explainability inAutoscaling Frameworks”. M.S. thesis. 2022 (cit. on pp. 1, 7, 10).

All links were last followed on April 27, 2023.

A Scenario Sheets

Base Scenario: Threshold Crossing

Description

A threshold on a monitored value is reached, causing the associated reaction to trigger.

Example Model

An example model for this scenario is maintained at `https://github.com/Ulkusus/Scenario-ThresholdCrossing`.

Relevant Questions

- Which threshold was crossed?
- When was the threshold reached?
- What was the threshold value?
- How fast was the threshold reached?
- Are there more thresholds and will they be reached with the projected development?
- What reaction was caused?

Variations

As this is a very basic scenario, almost everything can be exchanged to create additional variations. So long as the modeled threshold is actually crossed within the simulated time.

Examples

- varying the resource that is both used and monitored
- changing the actual values within bounds
- exchanging the reaction
- adding additional thresholds on the same resource

Additional Notes

This represents the very basics of a self-adaptive system and is mostly included as a basis for more complex scenarios and testing.

Scenario: Delayed Provisioning

Description

A web-based single-service system with a load balancer reaches a response time threshold, but adjusting how much traffic is routed to backup servers is slow enough to cause an SLO violation.

Example Model

An example model for this scenario is maintained at `https://github.com/Ulkusus/Scenario-DelayedProvisioning`.

Relevant Questions

- Which threshold was crossed?
- When was the threshold reached?
- What was the threshold value?
- How fast was the threshold reached?
- Are there more thresholds and will they be reached with the projected development?
- What reaction was caused?
- How long did the reaction take?
- When was the updated capacity available?
- Were SLOs violated?
- Which SLOs were violated?
- How long was each SLO violated?

Variations

Most parts of this scenario can be changed to create more variations. The combined time it takes the system to respond and additional processing power to become available must however stay long enough for the SLO violation to occur.

Examples

- varying the increase in user arrivals between barely too fast and some higher bound
- varying between scaling out a virtual server / routing traffic to a backup server / starting an additional container
- changing the response times of the system and the backup within appropriate bounds

Additional Notes

A basic scenario in the field of Internet-based, usually cloud-based systems. Delayed provisioning could be caused by a slow system or e.g. a slow cloud provider, so both completion times may be interesting.

Scenario: Overprovisioning

Description

Because a single-service system with a load balancer reaches a response time threshold, it routes an overly large percentage of incoming traffic to backup servers.

Example Model

An example model for this scenario is maintained at `https://github.com/Ulkusus/Scenario-Overprovisioning`

Relevant Questions

- Which threshold was crossed?
- When was the threshold reached?
- What was the threshold value?
- How fast was the threshold reached?
- What reaction was caused?
- When was the updated capacity available?
- Were SLOs violated?
- How much bigger was the updated capacity?
- How high was the new response time compared to updated thresholds?

Variations

This scenario can be varied rather freely as long as the updated processing power counts as uneconomical as defined by e.g. stakeholders.

Examples

- varying levels of overly large fixed and/or growth-based increases in processing power
- varying the definition of uneconomical

Additional Notes

A basic scenario in the realm of Internet-based, usually cloud-based systems. Due to an oversight in setting up a load balancer, it responds to a crossed threshold by increasing the available processing power at an uneconomical scale. Note that under-provisioning can be seen as a variation of this scenario as the same questions should be asked in this case.

Scenario: Oscillating Provisioning

Description

A single-service system repeatedly commissions and decommissions additional processing power due to thresholds overlapping.

Example Model

An example model for this scenario is maintained at `https://github.com/Ulkusus/Scenario-OscillatingProvisioning`.

Relevant Questions

- Which threshold was crossed?
- When was the threshold reached?
- What was the threshold value?
- How fast was the threshold reached?
- What reaction was caused?
- When was the last time a threshold on this monitored value was crossed?

Variations

This scenario can be varied quite freely as long as the upper threshold stays higher than the scaled lower threshold.

Examples

- varying the resource that is both used and monitored
- changing the actual threshold values while maintaining their relation
- varying between scaling out a virtual server / routing traffic to a backup server / starting an additional container

Additional Notes

A more complex scenario in the realm of Internet-based, usually cloud-based systems. If a specific workload triggers both an upper threshold on a particular resource and the lower threshold on the same resource after scaling, it can lead to a cycle of adding and removing processing power.

Scenario: Cascading Delay

Description

A multi-service pipeline uses a separate autoscaler for each service, causing large delays when the services provision additional processing power one after the other due to a spike in workload.

Example Model

An example model for this scenario is maintained at `https://github.com/Ulkusus/Scenario-CascadingDelay`.

Relevant Questions

- Which threshold was crossed?
- When was the threshold reached?
- What was the threshold value?
- How fast was the threshold reached?
- What reaction was caused?
- When was the updated capacity available?
- How high was the workload before and after scaling?

- Were SLOs violated?
- Which SLOs were violated?
- How long was each SLO violated?

Variations

This particular scenario occurs only within fairly tight bounds and should be varied carefully, if at all. Thresholds must remain greedy so later services do not provision new processing power too early, solving the problem the scenario is meant to simulate.

Examples

- varying the resource that is both used and monitored
- changing the actual threshold values while keeping them proportional to each other

Additional Notes

A complex scenario in the realm of Internet-based, usually cloud-based systems. If a system uses separate autoscalers for each service in a pipeline and greedy thresholds, a spike in workload can take many times the duration it takes a single service to provision additional processing power until it can be dealt with. This happens due to each service only detecting the spike after its predecessor in the pipeline has completed scaling up and is actually capable of pushing the increase in workload further.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature