



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

# Managing Operational Uncertainties in Deployed Systems Using Logic Reasoning

**Scientific Students' Association Report**

Author:

Márton Tarnay

Advisor:

András Földvári

Dr. Imre Kocsis

2023

# Contents

<b>Kivonat</b>	<b>i</b>
<b>Abstract</b>	<b>ii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Application . . . . .	2
1.2 Structure of the report . . . . .	3
<b>2 Deployed Systems</b>	<b>4</b>
2.1 Monolithic Applications . . . . .	4
2.2 Distributed Systems . . . . .	4
2.3 Microservice Architecture . . . . .	5
2.4 System deployment . . . . .	6
<b>3 Distributed Tracing</b>	<b>7</b>
3.1 Modern Tracing Frameworks . . . . .	7
3.1.1 Dapper . . . . .	7
3.1.2 OpenTelemetry . . . . .	8
3.2 Trace and Span Structure . . . . .	9
3.3 Benchmark System: Train Ticket . . . . .	11
<b>4 Qualitative Reasoning</b>	<b>12</b>
4.1 Overview . . . . .	12
4.2 Modeling . . . . .	12
<b>5 Allen’s Interval Algebra</b>	<b>15</b>
5.1 Requirements for Time Representation . . . . .	15
5.2 Temporal Intervals . . . . .	16
5.3 Example . . . . .	16
<b>6 Answer Set Programming</b>	<b>18</b>

6.1	Stable Solution . . . . .	18
6.2	Reasoning . . . . .	18
6.2.1	Incomplete Information . . . . .	19
6.2.2	Additional Building Blocks . . . . .	19
6.3	Difference Logic Extension . . . . .	20
6.3.1	Clingo-DL . . . . .	22
6.4	Interval Logic in ASP . . . . .	23
<b>7</b>	<b>Proposed Approach: Temporal Reasoning over Distributed Traces</b>	<b>24</b>
7.1	Distributed Traces in Allen’s Interval Algebra . . . . .	24
7.1.1	Spans as Intervals . . . . .	24
7.1.2	Modeling from Timing data . . . . .	25
7.1.3	Modeling from Relation Tree . . . . .	25
7.2	Applications . . . . .	27
7.2.1	Missing Span Substitution . . . . .	28
7.2.2	Behavior Consistency Check . . . . .	29
7.2.3	Trace Comparison . . . . .	30
7.2.4	Merge . . . . .	31
<b>8</b>	<b>Proposed Approach: Qualitative Diagnosis over Distributed Traces</b>	<b>32</b>
8.1	Qualitative Approach . . . . .	32
8.2	Propagation Analysis . . . . .	32
8.3	Mapping Traces and Spans . . . . .	33
8.4	Applications . . . . .	34
8.4.1	Example Trace . . . . .	34
8.4.2	Diagnostics . . . . .	35
8.4.3	What-ifs . . . . .	36
8.4.4	Co-deployment Design and Evaluation . . . . .	37
<b>9</b>	<b>Summary</b>	<b>39</b>
9.1	Conclusion . . . . .	39
9.2	Further Work . . . . .	39
	<b>Acknowledgements</b>	<b>40</b>
	<b>Bibliography</b>	<b>41</b>

# Kivonat

A modern, nagyméretű, elosztott rendszerek egyre összetettebbé válnak, és akár több ezer számítógépre kiterjedő szoftvermodulok gyűjteményéből állnak. Az elosztott rendszerek tranzakciói számos erőforrást használnak lefutásuk során, ami rendkívül nehézé teszi nyomon követésüket. Bár az elosztott alkalmazások megfigyelhetősége és a rendszerrel kapcsolatos bizonytalanságok csökkentése egyre jobban javul, mégis akadnak olyan megfelelési és megfigyelhetőségi követelmények, amiknek vizsgálata csak az éles rendszerben lehetséges.

A dolgozat bemutat egy logikai következtetés alapú módszert az elosztott rendszerekben lévő működési bizonytalanságok feltárására és csökkentésére. A megoldás kihasználja az elosztott rendszerekben használt nyomkövetésből származó információkat, amik lehetővé teszik a hibaterjedés elemzést és a konzisztencia ellenőrzést a rendszer működése során rögzített tranzakciók útvonalai és a rendszerrel szemben támasztott feltételezések és követelmények között.

A dolgozatban bemutatott módszer a működési nyomok fölötti következtetéshez, a folyamatok közötti temporális kapcsolatok összefüggéseit vizsgálja. A megoldás kiértékeléséhez a dolgozat mintapéldákon bemutat több következtetési célt a bizonytalanságok csökkentéséhez. Továbbá a megoldás beillesztését az elosztott nyomkövetési munkafolyamatba.

Egyetlen tranzakció egy nagyméretű elosztott rendszerben számos szolgáltatást érinthet az útvonala mentén. A dolgozatban bemutatott megközelítés az elosztott rendszerekben elterjedt nyomkövetési megoldásra épít, ahol intervallumokra bontva vizsgálhatók a tranzakció kiszolgálásához kapcsolódó folyamatok és igénybe vett szolgáltatások. Ezáltal a kiértékelésre az Allen intervallum-algebrájának széleskörű eszköztárát használhatjuk. Allen intervallumalgebrája egy olyan kalkulus, amely meghatározza az időintervallumok közötti kapcsolatokat, és alapot nyújt az események időbeli leírásaival kapcsolatos következtetésekhez.

A megoldás a logikai következtetés a deklaratív Answer Set Programming, Difference Logic kiterjesztését használja, amivel kezelhetők a temporális kapcsolatok a működési nyomvonalak között. Az intervallumok és nyomvonalak feletti következtetés rugalmas és adaptálható eszközöket biztosít a fejlesztők és az üzemeltetési csapatok számára a nyomvonalak elemzéséhez és értékeléséhez. Ez pedig a rendszerrel kapcsolatos bizonytalanságok csökkentésére, hibakeresésre, teljesítményoptimalizálásra és anomáliák észlelésére használható.

# Abstract

Modern large-scale distributed systems are becoming more and more complex, comprised of a collection of software modules spanning up to thousands of services. With modern tools, observability and reducing uncertainties within systems are becoming more manageable. However, some compliance and observability requirements remain, which can only be examined in production systems.

This report presents a logical reasoning-based approach to uncover and reduce uncertainties in distributed systems. The solution takes advantage of context from tracing frameworks already common in distributed systems. This information makes error propagation analysis and consistency checking between the system's actual behavior and the prior assumptions possible.

The method shown in the report uses the temporal relations between processes for logical reasoning over operational traces. In order to evaluate the solution, the report presents several examples of use cases in reducing uncertainties. It also presents how these methods can be used in conjunction with distributed tracing workflows.

A single transaction in a large-scale distributed system may involve many services along its path. The approach presented in the report is based on tracing frameworks common in distributed systems, where process calls and services can be examined as interval spans. We can evaluate and reason across these spans using the extensive toolset of Allen's interval algebra. Allen's interval algebra is a calculus for temporal reasoning, defining the relationships between intervals and providing a basis for temporal reasoning.

The solution is based on a form of declarative programming called Answer Set Programming, extended by Difference Logic, which allows the handling of temporal relations between operational traces. Reasoning over spans and traces provides flexible and adaptable tools for developers and operations teams to analyze and evaluate traces. This approach can reduce uncertainties related to the system, performance optimization, and anomaly detection.

# Chapter 1

## Introduction

Modern large-scale distributed systems are becoming more and more complex, comprised of a collection of software modules spanning up to thousands of services. With modern tools, observability and reducing uncertainties within systems are becoming more manageable. However, some compliance and observability requirements remain, which can only be examined in production systems. This report presents a logic reasoning-based approach to uncover and reduce uncertainties in distributed systems. The solution takes advantage of context from tracing frameworks already common in distributed systems. Using temporal and qualitative reasoning methods to increase knowledge and reduce uncertainties in the system.

The methodologies I developed, and their use cases are shown in Figure 1.1.

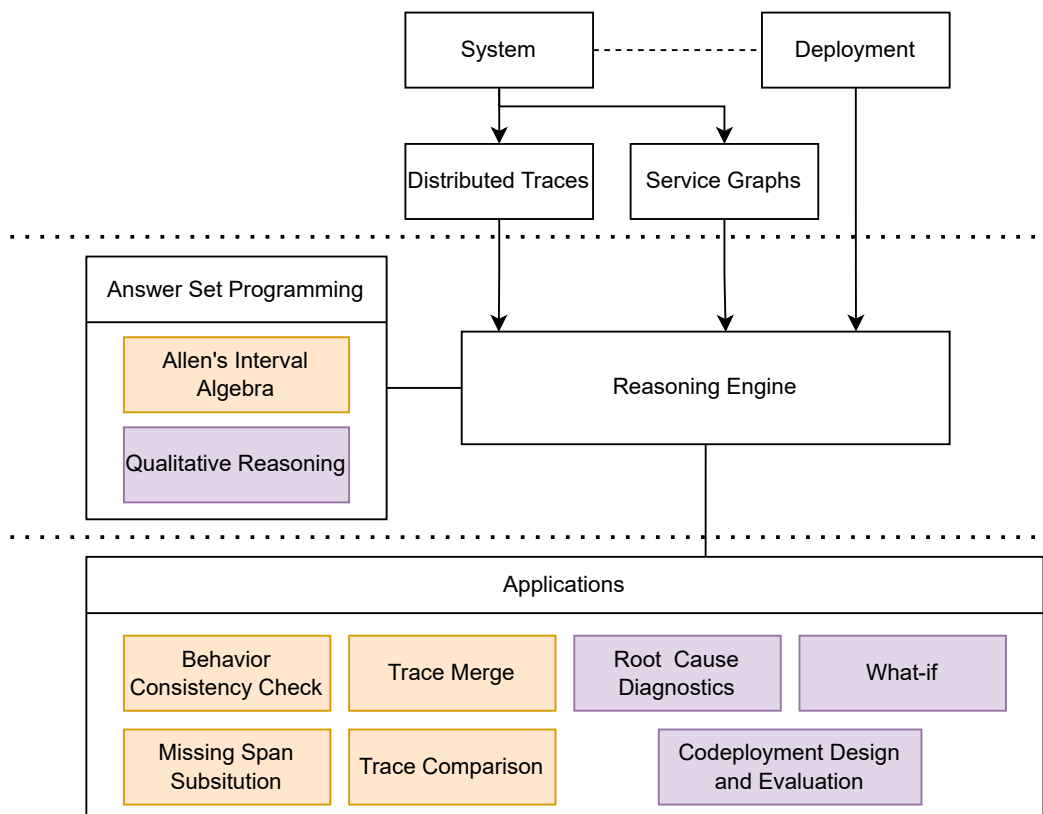


Figure 1.1: Overview

## 1.1 Application

Modeling and representing the data collected and known about these systems is parsed to a logic reasoning engine. This logic reasoning engine will provide the tools to reason and evaluate over this information in the methods shown in this report. This is showcased in Figure 1.1 in the upper section.

The bottom section of Figure 1.1 shows the applications of the methodologies outlined in the report. The yellow color represents the methods based on the temporal knowledge extracted from the traces and Allen's Interval Algebra. The applications marked with purple are based on Qualitative Reasoning and Error Propagation. The following paragraphs highlight the key features and use cases of these applications.

**Temporal methods** In tracing the path of a request and its sub-processes can be described in temporal space. Observing the timing and temporal relationships between these calls provides valuable insight into the inner workings of these systems. The following subsections describe use cases where the temporal methods of this report can be employed.

- **Missing Span Substitution:** Incomplete traces may be caused by a number of defects in the tracing framework, from missing instrumentation to faults in the recording and storage systems. The Missing Knowledge application introduced in this report reduces the uncertainty caused by these issues, allowing engineers to extend these partial traces with the missing spans based on system knowledge.
- **Behavior Consistency Check:** The expected normal execution and behavior of request is defined and known during system design. During testing and operation, identifying abnormal or unexpected behavior is extremely beneficial from a large data set. To aid in this, the report introduces trace consistency checking based on the temporal relations of executions. This provides flexibility to only mark recorded requests where the requirements set towards the system are broken.
- **Trace Comparison:** The execution of requests in a system may differ for a number of reasons. The comparison of traces allows engineers to understand the behavior of these requests better. It is also a valuable tool to analyze faulty requests and the effect of changes in the system.
- **Trace Merge:** Real-world systems are often made up of multiple sub-systems developed by separate teams and organizations. These sub-systems may be provided with their own tracing and instrumentation. Still, the connection points between these sub-systems can lead to situations where the detailed trace information is hidden by only a call to the other sub-system. To increase observability and promote modularity, this report introduces a temporal relation-based method to merge traces related to the same high-level request but recorded on different sub-systems.

**Propagation-based methods** Services, spans, and physical or virtual nodes are all interconnected. Exploring the propagation and spread of information, faults, and other metadata can be beneficial during the planning, modeling, and operating of these large-scale systems. Often, however, there is very little information to use in this type of analysis. This report presents methodologies based on the minimum and maximum execution time of calls and functions while also taking into account other known properties of the system, such as service structure and both physical and virtual architecture.

- **Root Cause Diagnostics:** Based on known timing data and the architecture, abductive reasoning can find the possible root causes of faults. Abductive reasoning involves inferring the possible causes of system behavior from measured and known symptoms.
- **What-if:** System designers and operators can find insight into possible future issues or improvements by evaluating changes and abnormal behavior without testing on real operational systems. To support this task, this report introduces deductive reasoning to model system behavior based on introduced faults or modifications.
- **Co-deployment design and evaluation:** Called Affinity Check in the report. Services often need to be separated into different physical or virtual nodes to provide the required performance and safety needed in distributed systems. This ensures that two subsystems may not interfere with each other even under heavy loads or possible fault conditions. This report introduces tracing-based affinity and anti-affinity checking to validate the correct configuration and behavior of these separated services and functions.

## 1.2 Structure of the report

The following chapters describe the basics of microservice-based deployed systems in Chapter 2 and the tracing frameworks commonly used to increase observability in Chapter 3. After that, it introduces the concepts and technologies used in the report. Chapter 4 introduces Qualitative Reasoning, a form of abstraction used to reduce temporal complexity. Chapter 5 describes Allen's Interval Algebra, a powerful tool to represent and reason over temporal intervals. Chapter 6 details Answer Set Programming, a form of declarative programming used in the report.

The report showcases the two novel approaches I created to assist in reducing uncertainties in deployed systems. The first is a temporal knowledge-based solution. The methods and its applications are shown in Chapter 7 and the qualitative diagnosis-based approach and its applications are in Chapter 8. Finally, the conclusions, further plans and possibilities are described in Chapter 9.

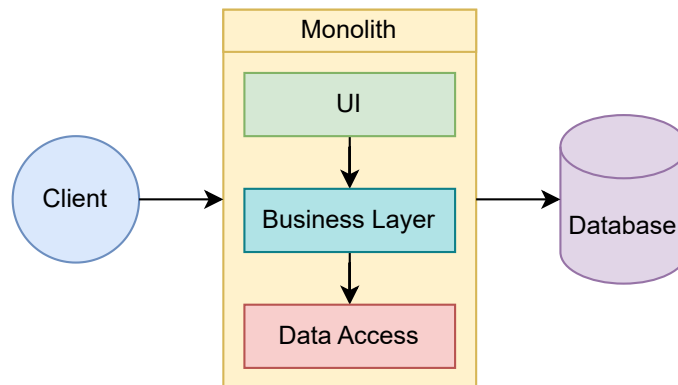


## Chapter 2

# Deployed Systems

### 2.1 Monolithic Applications

Monolithic applications and architecture represent a traditional approach to software design, where all components and functions of an application are tightly integrated into a single, unified system. The entire application, including the user interface, business logic, and data access layers, is built as a single, self-contained unit. Monolithic applications are known for their simplicity, as they are easier to develop, test, and deploy compared to more complex architectures.



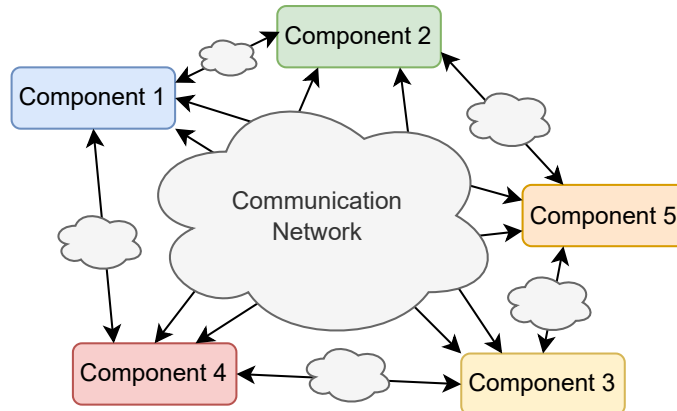
**Figure 2.1:** Monolithic Applications

However, they can become unwieldy and difficult to maintain as they grow in size and complexity. One of the key challenges with monolithic applications is that any changes or updates to one part of the system can potentially affect the entire application, making it challenging to scale and adapt to evolving business needs. With the advent of microservices architecture, many organizations are transitioning away from monolithic applications to embrace more modular and scalable approaches to software development.

### 2.2 Distributed Systems

Distributed systems, in contrast to monolithic architectures, are designed to handle complex tasks by distributing them across a network of interconnected computers. This approach enables collaboration and resource sharing, enhancing performance, fault tolerance, and scalability. In distributed systems, components can be geographically dispersed yet

work together as a single, cohesive unit. The fundamental idea behind distributed systems is to break down a large task into smaller sub-tasks that can be processed independently on different machines. These systems often rely on technologies such as remote procedure calls to facilitate communication between components.

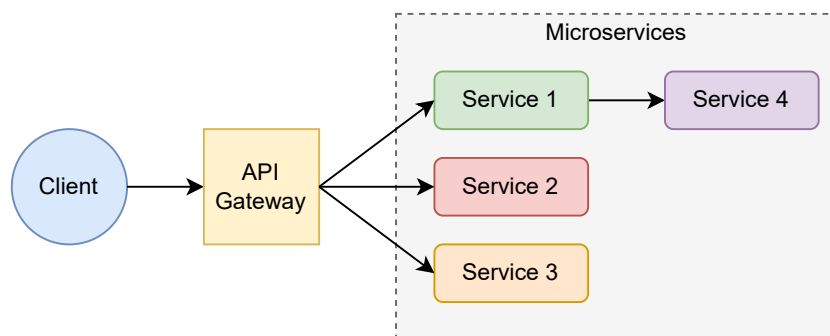


**Figure 2.2:** Distributed Architecture

Distributed systems offer several advantages, including improved reliability, as failures in one part of the system do not necessarily affect the entire system. However, building and managing distributed systems come with their challenges, such as debugging, handling network failures, and emergent properties from the interaction of a large number of subsystems. Despite the complexities involved, distributed systems play a crucial role in modern computing, powering everything from cloud computing platforms to large-scale web applications.

## 2.3 Microservice Architecture

Microservice architecture is a contemporary approach to software design where a complex application is broken down into smaller, loosely coupled services, each representing a specific business functionality. These services operate independently and communicate with each other through well-defined APIs. Unlike monolithic architectures, microservices allow for flexibility and scalability as each service can be developed, deployed, and scaled independently.



**Figure 2.3:** Microservice Architecture

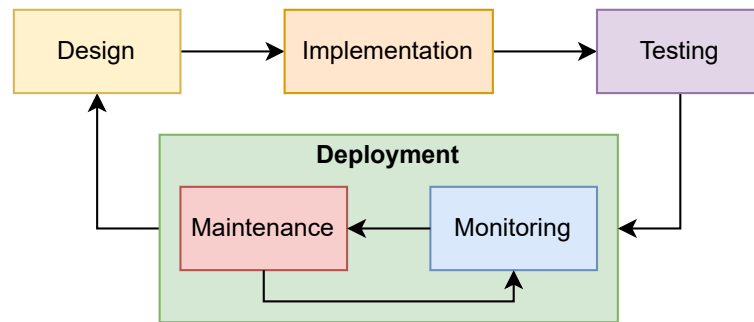
Microservices promote a distributed, decentralized approach to development, allowing teams to work on different services concurrently, using different technologies and pro-

programming languages best suited for each service’s requirements. This architectural style fosters rapid development, facilitates continuous integration and deployment, and enables organizations to adapt to changing business needs more effectively.

However, managing the communication between services, ensuring data consistency, and dealing with potential service failures are challenges that need to be carefully addressed in microservice architectures. Despite the complexities, the benefits of microservices, such as improved agility, scalability, and resilience, make them a popular choice for building modern, large-scale applications and services. [6]

## 2.4 System deployment

System deployment is a critical phase in the software development lifecycle where the developed application or system is prepared and released for users. It involves the process of installing, configuring, testing, and making the software operational for its intended users or clients.



**Figure 2.4:** Deployment Process

Deployment strategies can vary widely, from distributed cyberphysical systems to traditional on-premises installations and cloud-based deployments. Continuous monitoring and maintenance post-deployment are essential to address user concerns, implement updates, and maintain the system’s performance and security.

## Chapter 3

# Distributed Tracing

In complex, distributed systems, where requests are spread across many applications and services, understanding the flow of information and the interactions between different components is paramount. Distributed tracing provides a solution to this challenge by offering a comprehensive view of the entire system’s behavior, transcending the boundaries of individual services and servers. This approach not only facilitates rapid issue detection and troubleshooting but also empowers organizations to enhance their system performance.

### 3.1 Modern Tracing Frameworks

Distributed Tracing Frameworks were created to increase the observability of systems based on microservice architecture. These frameworks have emerged as indispensable tools, providing developers with invaluable insights into the internal workings of complex systems. [3] The following chapters describe state-of-the-art distributed tracing frameworks and tools, common standards used for tracing, and a benchmark example system used as an example later in the report.

#### 3.1.1 Dapper

Dapper [12] is a tried and proven distributed systems tracing infrastructure created and used by Google. After two years of deploying and using the system, there are a number of important achievements and lessons to take away. The aim of this section is to highlight the positive impact and characteristics of a high-quality tracing infrastructure and showcase the importance of this technology.

The aim of Dapper was to aid Google developers in understanding the behavior of their complex distributed systems. These systems were particularly important to Google, as large collections of small servers proved to be a cost-efficient platform for Internet services workloads. Tracing was selected as the optimal solution to observe these systems because the behavior is spread across a number of services and machines.

Three main requirements were made before development:

- Low overhead: For the result to be useful and normal operation to remain possible, the tracing infrastructure needed to have a negligible performance impact on the service performance.

- **Application-level transparency:** To maintain a ubiquitous tracing environment, it was important to not rely on application-level developers' collaborations. This is why Dapper opted in general, that developers should not need to be aware of the tracing system. However, it is important to note that Dapper still allows developers to include additional instrumentation.
- **Scalability:** Google's systems are increasingly large, and the system needed to handle this vast size.

Trace collection is handled by a three-stage logging and collection pipeline, shown in Figure 3.1. First of all, local log files are created from span data. These logs are pulled from the production host by Dapper collectors. Finally, it is written to one cell of a regional Dapper Bigtable repository. Each cell represents a span, and a row makes up an entire trace. The median latency in this collection process is less than 15 seconds. Allowing developers to almost immediately access and process traces from their systems.

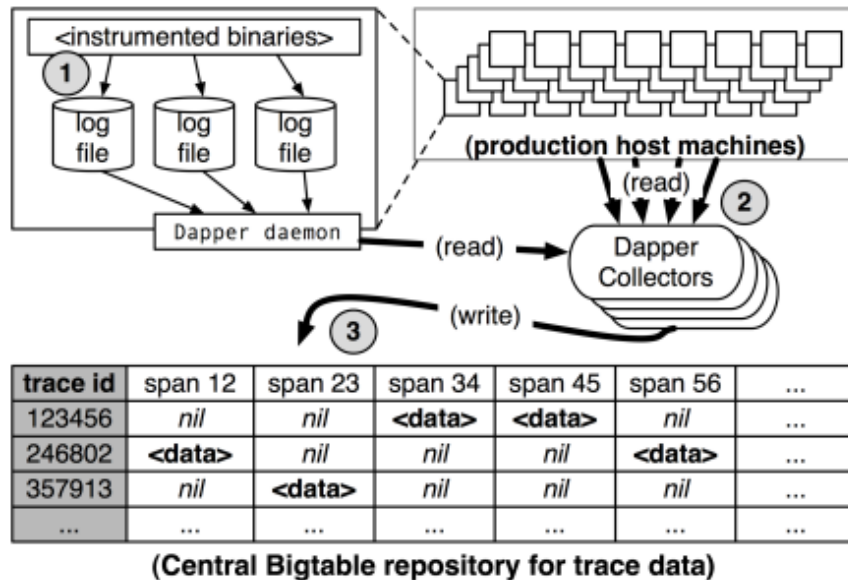


Figure 3.1: Dapper Pipeline. Source: [12], p5

After over two years of being the main production tracing system for Google, Dapper has generally met its objectives and shown previously unseen uses of tracing data. Providing extremely low overhead and paving the way for a number of developer tools.

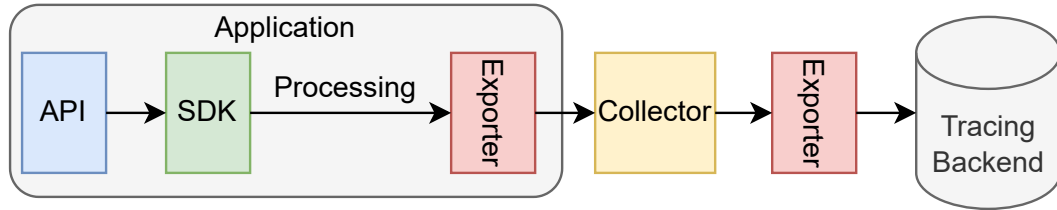
### 3.1.2 OpenTelemetry

This section aims to show that there are existing free, open-source, and modular tracing and observation frameworks and tools. These make the use of distributed tracing a significantly more favorable option for developers seeking to increase productivity and efficiency.

OpenTelemetry is a vendor-neutral open-source Observability framework under the Cloud Native Computing Foundation project for for instrumenting, generating, collecting, and exporting telemetry data such as traces, metrics, and logs. [13] This report focuses on its tracing framework, previously known as OpenTracing. This framework became the industry standard and is currently supported by over 40 observability vendors, integrated by many external libraries, tools, and services. It is important to note that OpenTelemetry

is solely focused on the generation, collection, management, and export of telemetry data. In itself, it provides no tools to store or visualize this data.

The main objective of OpenTelemetry was to fulfill the ever-increasing need for observability in modern complex systems. Observability is the ability to understand the internal state of a system by examining its outputs. In the context of software, this means being able to understand the internal state of a system by examining its telemetry data, which includes traces, metrics, and logs. [13] To fulfill this objective, it created a new common standard for code instrumentation and sending telemetry data to an Observability backend. This was achieved by combining two prior projects, OpenTracing and OpenCensus.



**Figure 3.2:** OpenTelemetry Architecture

OpenTelemetry is designed to be extensible. It allows developers to extend the framework at almost every possible level. This could allow us in the future to extend the framework with the methods and programs showcased in the report.

## 3.2 Trace and Span Structure

From now on, this report will focus on the trace and span structure of the OpenTelemetry standard, as it is considered the industry standard.

A trace represents the complete journey of a specific request as it traverses various services and components within an application. These traces provide insights into how requests are processed, which services are involved, and how much time is spent at each stage of the request. Traces in OpenTelemetry have a hierarchical structure consisting of spans.

Spans are the building blocks of traces. They represent a single operation within a trace and encapsulate information about a specific interval during which an operation occurred. Spans are organized hierarchically, allowing for the representation of parent-child relationships between different spans. The root span represents the overall request, while child spans represent sub-operations or actions that are part of the larger request processing flow. Spans in OpenTelemetry include information and attributes about the operation (Table 3.1).

Information	Description	Example
Name	The name of the operation.	/v1/ts-food-service/food-list
Parent span ID	Specifies the parent span, required for the hierarchical tree structure.	086e83747d0e381e
Start and End Timestamps	The start and end times of the span.	start_time: 2023-10-31 03:33:32.209459162 end_time: 2023-10-31 03:33:32.210512132
Span Context	Span context contains the Trace ID, the Span ID. May also contain other metadata.	"context":{ "trace_id": "0x5b8aa5a2d2c872e8321cf373", "span_id": "0x5fb397be34d26b51" },
Attributes	Attributes are key-value pairs to annotate a Span to carry information about its operation.	"attributes": { "http.route": "example_route" },
Span Events	A Span Event is a structured log message on a Span, used to denote a singular point in time during the span.	"event": { "name": "hello world!", "timestamp": "2023-10-30T14:56:59.124761Z" }
Span Links	Links associate one span with one or more spans, implying a causal relationship.	Meets 086e83747d0e381e
Span Status	A span status is set, when there is a known error in the application code, such as an exception.	Unset, Ok or Error

**Table 3.1:** Span information [13]

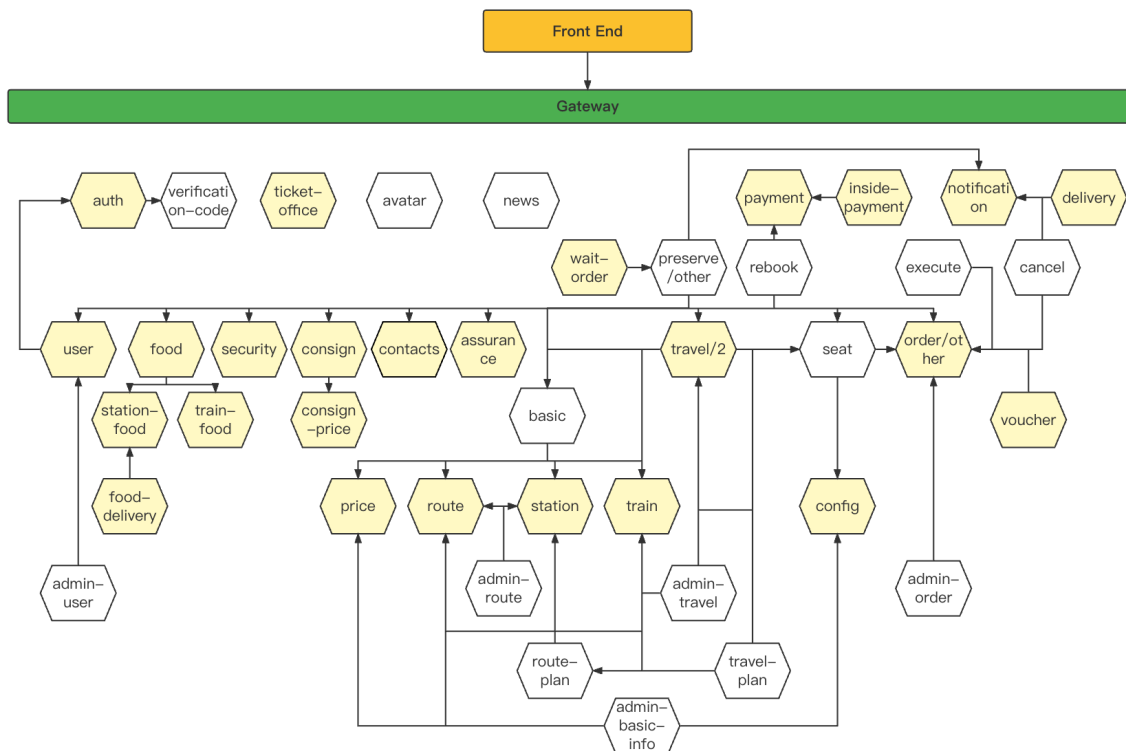
### 3.3 Benchmark System: Train Ticket

Train Ticket is an Open-Source Microservice Benchmark consisting of a train ticket booking system that contains 41 microservices. [4] The architecture of this system can be seen on Figure 3.3.

In this benchmark system, a user has the capability to inquire about available train tickets for a journey from city A to city B on a specific date. The user can make a reservation by selecting the passenger details and seat class that meet their requirements. Upon a successful booking, prompt payment is required. Subsequent to the successful payment, the user will receive an email confirming their ticket reservation. The user retains the flexibility to make ticket modifications before the scheduled departure or within a specified time window following the train's departure. [14]

The system is designed using the principles of microservice architecture. It contains and uses synchronous invocations, asynchronous invocations, and message queues. The design strategy also promotes flexibility and adaptability, the two main strengths of microservice-based systems. The design and structure of this benchmark system make it the ideal candidate to use in this report.

Train Ticket provides support for both Docker and Kubernetes deployment for benchmarking. This report does not utilize these features. However, in the future, this system is to be used in the performance benchmarking and testing of the methods shown in this report.



**Figure 3.3:** Train Ticket Benchmark - Service Architecture Graph. Source [4]



## Chapter 4

# Qualitative Reasoning

### 4.1 Overview

Qualitative reasoning is a form of abstraction, which considers the understanding and analysis of systems based on qualitative rather than quantitative information. Unlike quantitative methods that rely on numerical measurements, qualitative reasoning focuses on abstract, qualitative aspects to gain insights into complex phenomena. This approach is particularly valuable when dealing with diverse and intricate systems, where precise quantitative data might be challenging to interpret.

**Discretization** In qualitative reasoning, continuous properties are represented by discrete entities. This allows symbolic representation and reasoning. Secondly, it is a means of abstraction. A continuous variable with an infinite number of possible values may be represented only by its sign (+,0,-). This abstraction allows models to work even with minimal information, requiring only a few details. [9]

**Relevance** Discretization is based on the nature of the system and the planned reasoning about it. Qualitative value sets are defined to be relevant to the task at hand. Within a specific qualitative value, the behavior should be the same, in regards to the goal of the reasoning. Different purposes require a different approach to discretization, keeping in mind the relevance of the qualitative regions. [9]

**Ambiguity** The abstraction introduced by Qualitative Reasoning comes with its drawbacks. Often there is not enough information to come to a certain conclusion, leading to several possible predictions. Because of this, qualitative models often produce ambiguous results. However, this can also be seen as a strength. Qualitative modeling algorithms can help identify the relevant phenomena and find possible behaviors of the system. [9]

### 4.2 Modeling

Figure 4.1 shows a possible quality space in the case of water temperature. The quality space is separated by multiple landmarks, absolute nil point, freeze point, boil point, and infinite plus. The values were chosen to show the physical properties of water. If the goal of the system requires it, this quality space could be expanded or simplified as needed.

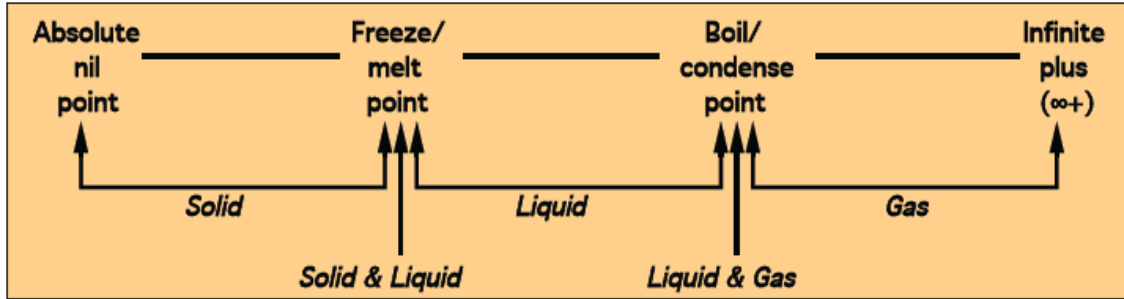


Figure 4.1: Quality Space. Source [11], p14.

For example, in the case of drinking water, more landmarks could be introduced to mark the safe temperature range for consumption.

Systems are made up of entities, which are physical objects or abstract concepts in the system. The relevant properties of these entities are represented as quantities that can change from influences within the system. Qualitative Reasoning considers two main types of relations between properties shown in Figure 4.2.

**Direct Relation** is a directed relation between two properties. The influence may be positive or negative. In the case of a positive direct influence, the increase in the source quantity will lead to an increase in the target quantity, while a decrease will lead to a decrease. In the case of negative influence, the increase in the source quantity will cause a decrease within the target quantity. An example of this relation can be seen in Figure 4.2 (left) between the source Flow In quantity and the target Change Amount (water level). As the Flow In increases, the Change Amount of the water level will increase, as more water will flow in.

**Proportional Relation** helps to propagate the effects within a modeled system. The derivative of the target quantity changes based on the derivative of the source quantity. This relation can also be positive or negative, where this change will be directly proportional in the case of positive proportional influence and inversely proportional in the case of negative proportional influence. This relation is shown in Figure 4.2 (right) between (water) Amount and (water) Height. As the water amount increases, its derivative becomes positive, leading to the derivative of the water height to be positive, causing an increase in Height.

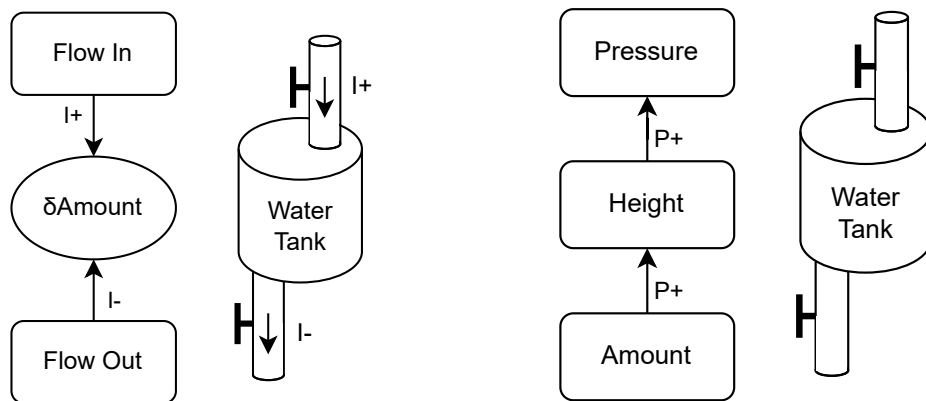


Figure 4.2: Direct Influence (left) and Proportionality (right). Source [11], p15, p16.

	+	0	-
+	+	+	?
0	+	0	-
-	?	-	-

**Table 4.1:** Two Influence Relations

**Magnitudes** of properties is important to consider, as sometimes effects can be ignored because they are negligible compared to others. The pressure decrease inside the water tank caused by the expansion of the universe would be negligible compared to the pressure increase caused by the water level increases. A common strategy to deal with magnitude representation is partitioning the magnitude into distinct equivalence classes, such as small, medium, and large. In the previous example, the first property would be a small decrease, and the pressure increase caused by the water level increasing would be a large increase. With this additional information about the magnitude, it can easily be deduced that the sum of these influences will lead to an increase. However introducing magnitude representation into qualitative models will lead to an increase in complexity, and the system requirements must be considered when deciding the ideal system model.

**Uncertainty** is caused due to influences from multiple sources. Deducing the sum of multiple contradictory influences is not possible for a single solution. To handle this the model needs to have a defined way to deal with multiple influences. Table 4.1 shows the sum of two influences can be seen. In two scenarios the sum of the influences may be uncertain. In this case, all possible outcomes have to be examined (increasing, steady, decreasing).

In case of more than two influences, if there are both increasing and decreasing influences the outcome can not be evaluated with complete certainty. This is because the abstraction of the qualitative model hides the relation and magnitude between changes. Using some form of magnitude representation helps to decrease unknowns caused by this delta calculus but due to the abstraction nature of qualitative reasoning, uncertainties can never be completely avoided, only mitigated.

# Chapter 5

## Allen's Interval Algebra

Representing and reasoning with temporal knowledge is needed in many areas, including computer science, from program verification to process modeling and even artificial intelligence. Interval-based temporal logic strives to effectively deduce and express temporal hierarchy. Allen's Interval Algebra [1] takes intervals as primitives and describes the relationships between these intervals using constraint propagation.

### 5.1 Requirements for Time Representation

Timing in microservice-based applications is ever-changing. This requires a great deal of flexibility in time representation. To be able to create assumptions and reason across traces recorded from these systems the following characteristics are needed:

**Uncertainties** about the architecture, deployment, and execution of requests must be considered and allowed. Due to the ever-changing and adapting architecture designers and operators do not have a concrete rule set about the relation of spans in traces. Ensuring that spans and their behavior can be analyzed while allowing unknowns in the model way is imperative.

**Relative knowledge** makes up most of the constraints and conditions when overseeing traces. Viewing spans relative to one another allows the required freedom needed to view and reason in changing conditions. The framework used to model spans and traces must be able to accommodate this.

**Variable precision** is a fundamental requirement when modeling complex systems. It is essential to recognize that different situations demand varying degrees of detail and accuracy in capturing temporal aspects. In some scenarios, a broad overview of time-related events might suffice, while in others, a highly granular and precise representation is necessary to capture subtle nuances. Acknowledging these disparities in timing magnitudes is crucial for developing effective models that reflect the intricate dynamics of the overall system.

Time point and state space-based approaches prove insufficient in modeling microservice-based systems due to their limitations in capturing complex temporal relationships, concurrency, and dynamic nature of microservices. Time points lack precision in representing


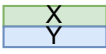





interactions, leading to inadequate modeling of overlaps and durations, while state space methods struggle to handle concurrent activities and dependencies effectively.

Allen’s interval algebra offers a richer and more expressive framework for modeling the complex temporal aspects of microservice-based systems. It provides a versatile way to capture temporal relationships, concurrency, dynamic behaviors, temporal constraints, and dependencies, and supports formal reasoning, making it a suitable choice for modeling and analyzing the temporal aspects of microservice architectures.

## 5.2 Temporal Intervals

The calculus characterizes the possible interactions and their inverse between intervals in 13 distinct relations, including "before," "after," "meets," "met by," "overlaps," "overlapped by," "during," "contains," "starts," "started by," "finishes," "finished by," and "equals," each describing a specific temporal relationship between two intervals. These can be seen on Figure 5.1. These relations provide a standardized and intuitive way to describe the temporal order and overlap between intervals, allowing for precise modeling of complex temporal scenarios.

Understanding these relations is crucial for analyzing the temporal aspects of complex processes. The formalism provided by Allen’s Interval Algebra and its associated toolset serves as a base for applications requiring temporal reasoning, ensuring accurate modeling and analysis of temporal relationships.

Relation	Symbol	Inverse Symbol	Example
X precedes Y	p	pi	
X equal Y	eq		
X meets Y	m	mi	
X overlaps Y	o	oi	
X during Y	d	di	
X starts Y	s	si	
X finishes Y	f	fi	

**Figure 5.1:** Interval relations

## 5.3 Example

The following example, taken from "Maintaining Knowledge about Temporal Intervals" [1] by James F. Allen, showcases the uses and methods of Allen’s Interval Algebra.

"John was not in the room when I touched the switch to turn on the light."

Let S be the time of touching the switch, and L be the time interval where the light was on. R shall be the time that John was in the Room. We can describe the possible relations as follows:

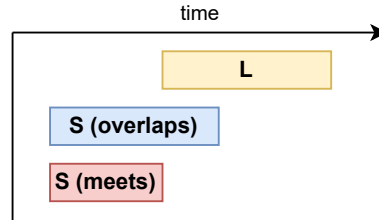


Figure 5.2: S overlaps (o) or meets (m) L.

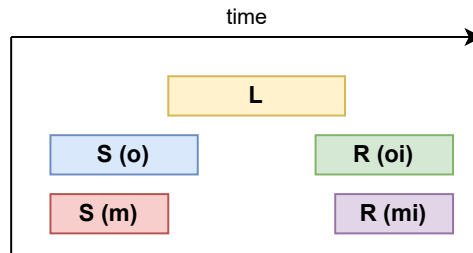


Figure 5.3: S precedes (p), meets (m), is met by (mi) or after (pi) R.

- S overlaps (o) or meets (m) L.
- S precedes (p), meets (m), is met by (mi) or after (pi) R.

The possible arrangement of intervals is visualized on Figure 5.2 and Figure 5.3.

Continuing the story with the statement: "But John was in the room later while the light went out" allows us to expand the knowledge base of relations.

The possible relations deduced from this statement are as follows:

- L overlaps (o) or starts (s) R.

Combining this with all the temporal knowledge deduced previously we decrease the number of uncertainties about the intervals. Figure 5.4 shows the complete reconstruction of events. It can be seen, that some relations proved to be impossible in the complete story.

From this information and from the previous knowledge about the relations between R and S, the fact, that S precedes or meets R can be deduced.

Allen's Interval Algebra introduces its own algorithm to reason across relations. This report uses a different approach based on difference logic, introduced in Section 6.3.

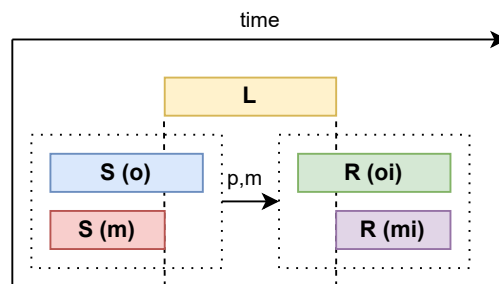


Figure 5.4: Reconstruction of events

## Chapter 6

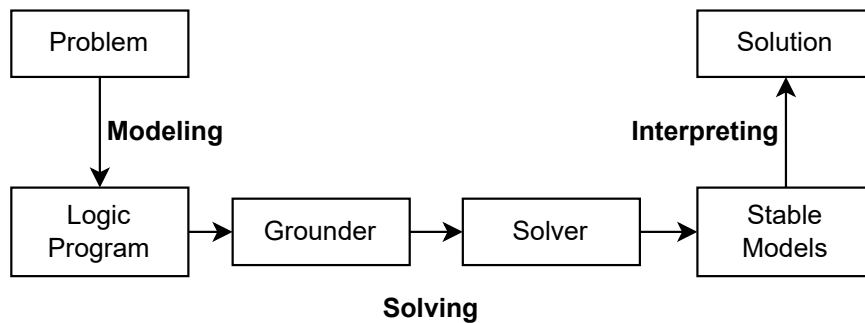
# Answer Set Programming

Answer Set Programming (ASP) [2] is a declarative problem-solving approach, mainly focused on knowledge-intensive combinatorial problems. The major advantages of ASP are its simplicity and its ability to model incomplete specifications.

In this report, ASP will be used as the reasoning framework to model and evaluate the relationships in and between traces.

### 6.1 Stable Solution

The ASP solving process consists of three main steps, as shown in Figure 6.1. Modeling transforms the problem into a logic program. During the solving, the grounder and solver determine the answer sets (stable models) that fulfill the rules of the logic program. The solution is created by interpreting the stable models.



**Figure 6.1:** Solving Process

### 6.2 Reasoning

The building blocks for ASP programs are atoms, literals, and rules. Atoms are considered elementary propositions that may be true or false. Literals are atoms or their negations. Rules can be created from literals, as seen here:

$$r : - a1, a2, \text{not } a3.$$

Where  $a_1$ ,  $a_2$ ,  $a_3$ , and  $r$  is also an atom.

The head of this rule is  $r$ , the body consists of the literals  $a_1$ ,  $a_2$ , and  $\text{not } a_3$ . This rule provides justification to "establish" or "derive" that the head is true if all literals in the body are true. A non-negated literal ( $a_1$ ) is true if the atom ( $a_1$ ) has a derivation. A negated literal ( $\text{not } a_3$ ) is true if the atom ( $a_3$ ) has no derivation.

Facts represent the background knowledge of the system. Rules are used to deduct results from facts. Facts are special rules which contain no body.

$$\begin{array}{c} \textit{service\_ok}. \\ \textit{behavior\_steady} : \textit{-service\_ok}, \textit{not fault\_outage}. \\ \hline \{\textit{network\_ok}, \textit{behavior\_steady}\} \end{array}$$

Disjunctive rules are used to describe the nondeterministic behavior of the system. A simple example may be when, out of two facts, only one is true, but determining which exactly is impossible.

$$\begin{array}{c} \textit{service\_ok}. \\ \textit{behavior\_steady} : \textit{-service\_ok}, \textit{not fault\_outage}. \\ 1\{\textit{response\_time\_low}, \textit{response\_time\_medium}\}1 : \textit{-behavior\_steady}. \\ \hline \{\textit{service\_ok}, \textit{behavior\_steady}; \textit{response\_time\_low}\} \\ \{\textit{service\_ok}, \textit{behavior\_steady}; \textit{response\_time\_medium}\} \end{array}$$

Integrity constraints determine the boundaries of the system to eliminate undesirable results and inconsistencies.

$$\begin{array}{c} 1\{\textit{fault\_overload}, \textit{fault\_ratelimit}, \textit{fault\_outage}\}1. \\ \hline \{\textit{fault\_overload}\}, \{\textit{fault\_ratelimit}\}, \{\textit{fault\_outage}\} \end{array}$$

### 6.2.1 Incomplete Information

ASP considers two types of negation. Negation as failure means that deriving " $\text{not } a_3$ " comes from a failure to derive " $a_3$ ". This differs from the classical logic negation of " $\neg a_3$ ". " $\text{not fault.}$ " means that we have no knowledge of a fault in the system. While " $\neg \text{fault.}$ " means that there is no fault in the system. With this tool, incomplete information can be accurately modeled.

### 6.2.2 Additional Building Blocks

Atoms can be extended by arguments and variables. Arguments begin with a small first letter, while variables begin with a capital first letter. The solver replaces variables with all possible arguments that could be in their places.

$$\begin{array}{c} \textit{min\_time}(\textit{foodlist}, \textit{steady}). \\ \textit{min\_time}(\textit{ticketinfo}, \textit{steady}). \end{array}$$



The above code snippet contains the `min_time` atom, which represents the minimum execution time of a span with its span name, and the qualitative representation of the execution time (decreasing, steady, increasing). The two span names, `foodlist`, and `ticketinfo`, are considered arguments. This is evidenced by them starting with a small first letter.

$$\text{span\_name}(Name) : - \text{min\_time}(Name, \_).$$


---


$$\{ \text{span\_name}(\text{foodlist}), \text{span\_name}(\text{ticketinfo}) \}$$

This code snippet collects the names of all spans. In the body of this rule "Name" is a variable, which will be replaced with all possible variables that exist in this atom. The underscore character indicates that an attribute is irrelevant during the evaluation of this rule.

Aggregates are functions that evaluate over a set of atoms and return a number. In this report, the Count aggregate will be used most commonly. This aggregate returns the cardinality of the atom set it is applied to.

$$\text{numberOfPosInfluence}(N) : - N = \#count\{B : \text{influence}(B, pos)\}.$$

Directives instruct the solver how to process the rules. Most commonly the `#show` directive is used, which determines what kind of atoms the output should include. The following number shows the number of arguments for the atom. In (2.2) `start_point` atoms with two arguments will be shown. In (2.3) the relation atoms with 3 arguments will be shown.

$$\%Directives \tag{6.1}$$

$$\#show \text{start\_point}/2. \tag{6.2}$$

$$\#show \text{relation}/3. \tag{6.3}$$

Single-line comments in ASP are marked with a `%` character at the start of the line, as seen in the previous code snippet (2.1).

### 6.3 Difference Logic Extension

Difference logic handles problems in the description of differences. These formulas are called difference constraints and must take the following form:

$$x - y \leq c$$

Where  $x$  and  $y$  are variables, and  $c$  is a constant. Meaning, the difference of  $x$  and  $y$  shall be less than or equal to  $c$ .

Usually, the inequalities in difference problems are not in the normal form by default. To create a difference constraint from an inequality, it must be brought to its equivalent normal form. The following are common transformations:

$x \leq y$  must be written as  $x - y \leq 0$ .

$x - y = c$  must be written as  $x - y \leq c$  and  $y - x \leq c$

$x - y \geq c$  must be written as  $y - x \leq -c$

1	M(A)	20
2	M(B)	10
3	M(C)	5

**Table 6.1:** Job 1

1	M(B)	30
2	M(A)	15

**Table 6.2:** Job 2

$x \leq c$  must be written as  $x - z_0 \leq c$ , where the special zero variable,  $z_0$ , must be 0.

The following example consists of finding a feasible schedule to get two jobs executed over three machines in the allotted time. The jobs consist of different operations, which may have to be executed on a different machine. Table 6.1 and Table 6.2 show the execution order of these operations, the machine that can execute the given operation, and the execution time.

From this, the difference constraints can be created as follows.

All operations must start after 0. In other words, the start time of each operation is greater than or equal to 0.

$$z_0 - s_{11} \leq 0$$

$$z_0 - s_{12} \leq 0$$

$$z_0 - s_{13} \leq 0$$

$$z_0 - s_{21} \leq 0$$

$$z_0 - s_{22} \leq 0$$

All operations within one job are executed sequentially. This means that the start time of the next operation is greater than the sum of the start time of the current operation and the current operations execution time.

$$s_{11} - s_{12} \leq -20$$

$$s_{12} - s_{13} \leq -10$$

$$s_{21} - s_{22} \leq -30$$

The operations on a machine are also performed sequentially.

$$M(A) : s_{11} - s_{22} \leq -20 \text{ or } s_{22} - s_{11} \leq -15$$

$$M(B) : s_{12} - s_{21} \leq -10 \text{ or } s_{21} - s_{12} \leq -30$$

From these difference constraints, the solution can be calculated using multiple methods. This report employs Clingo-DL to solve these difference logic problems.

### 6.3.1 Clingo-DL

**Clingo** is an ASP system to ground and solve logic programs. Created by the University of Potsdam as part of the Potassco project for Answer Set Programming. It combines the capabilities of their grounder, gringo, and their solver, clasp, into a complete system. Clingo also includes powerful integration tools with common scripting languages. [5]

**Clingo-DL** extends Clingo with constraints over difference logic. Allowing for modeling and solving naturally encoded timing-related problems, solvable in polynomial time. [8] Difference constraints can be implemented very efficiently since they enable a linear-time check for unsatisfiability. [7]

Formally, Clingo-DL creates assignments mapping variables to integers. A difference constraint  $x - y \leq c$  is satisfied by assignment  $\tau$  if  $\tau(x) - \tau(y) \leq c$ , this is denoted by  $\tau \models x - y \leq c$ .  $\tau \models S$  for a set of difference constraints  $S$  means that for every difference constraint  $\tau \models x - y \leq c$ . In this case,  $S$  is satisfiable and  $\tau$  is a solution to  $S$ . [7]

The returned value assignment (solution) is the one with the lowest sum of the values for all variables. For example, in case scheduling, this leads to scheduling each job as soon as possible.

A simple example shows the basic structure of difference constraints: The two variables inside the `&diff` block are calculated based on the constraints imposed by the model. In this example, it is known that  $x - y$  is smaller or equal to  $-3$ . The solver attempts to find the lowest possible values; thus, the solution is  $x = 0, y = 3$ .

$$\& \text{diff } \{x - y\} \leq -3.$$

---

$$\{dl(x, 0)dl(y, 3)\}$$

The example described in Section 6.3 can be implemented and solved in Clingo-DL. The inequalities created from the problem statement have already been modified to the standard form required by Difference Logic. All that is left to do is create the Clingo-DL program.

The inequalities ensuring that operations start after 0 can be written as shown in Listing 6.1. The atom "s" contains the information about the operation start time, the job identifier, and the operations index in the job.

$$z0 - s11 \leq 0$$

```
&diff {0 - s(1,1)} <= 0.  
&diff {0 - s(1,2)} <= 0.  
&diff {0 - s(1,3)} <= 0.  
&diff {0 - s(2,1)} <= 0.  
&diff {0 - s(2,1)} <= 0.
```

**Listing 6.1:** Ensure Start After 0

The code in Listing 6.2 implements the sequential operation within jobs.

```
&diff {s(1,1) - s(1,2)} <= -20.  
&diff {s(1,2) - s(1,3)} <= -10.  
&diff {s(2,1) - s(2,2)} <= -30.
```

**Listing 6.2:** Jobs

Sequential operation on the machines is constrained by the code snippet shown in Listing 6.3. Difference constraints can not be placed inside disjunctive rules, so another atom is used to determine which start time is earlier. A more suitable implementation of this behavior is possible, but that goes beyond the scope of this example.

```
O{s11_before_s22}1.
&diff {s(1,1) - s(2,2)} <= -20 :- s11_before_s22.
&diff {s(2,2) - s(1,1)} <= -15 :- not s11_before_s22.

O{s12_before_s21}1.
&diff {s(1,2) - s(2,1)} <= -10 :- s12_before_s21.
&diff {s(2,1) - s(1,2)} <= -30 :- not s12_before_s21.
```

**Listing 6.3:** Machines

Running the program yields the following solutions to the problem.

```
Answer: 1
d1(s(1,1),45) d1(s(1,2),65) d1(s(1,3),75) d1(s(2,1),0) d1(s(2,2),30)
Answer: 2
d1(s(1,1),0) d1(s(1,2),30) d1(s(1,3),40) d1(s(2,1),0) d1(s(2,2),30) s11_before_s22
Answer: 3
d1(s(1,1),0) d1(s(1,2),20) d1(s(1,3),30) d1(s(2,1),30) d1(s(2,2),60) s12_before_s21 s11_before_s22
SATISFIABLE
```

**Listing 6.4:** Solution

## 6.4 Interval Logic in ASP

"Allen's Interval Algebra Makes the Difference" [7] presents an encoding method for temporal networks using ASP extended by difference constraints. For each interval, a start and end point integer variable is introduced. Their approach introduces constants for the 13 relations (eq, p, pi, m, mi, o, oi, s, si, d, di, f) in Allen's Interval Algebra. An example of the overlap relation can be seen in Listing 6.5. The relations are described via a predicate, which contains three arguments. The first two arguments describe the intervals, and the third argument defines the relation between the two temporal intervals. This report uses this encoding to reason across spans in distributed traces, further elaborated in Chapter 7.

```
% X overlaps Y
& diff { sp(Y) - ep(X) } <= -1 :- relation(X,Y,o).
& diff { sp(X) - sp(Y) } <= -1 :- relation(X,Y,o).
& diff { ep(X) - ep(Y) } <= -1 :- relation(X,Y,o).
% X is overlapped by Y
relation(X,Y,o) :- relation(Y,X,oi).
```

**Listing 6.5:** Overlap relation

## Chapter 7

# Proposed Approach: Temporal Reasoning over Distributed Traces

Temporal Reasoning over Distributed traces is a novel approach I created for this report to better explore the use of temporal interval relations in the scope of distributed tracing. Viewing spans in traces as intervals I was able to use the tool set of Allen's Interval Algebra to reduce the uncertainties common in microservice based systems.

### 7.1 Distributed Traces in Allen's Interval Algebra

#### 7.1.1 Spans as Intervals

I considered spans as temporal intervals. This allows the use of Allen's Interval Algebra for logic reasoning. This is an abstraction over normal tracing, as exact timing and other attributes are dismissed, keeping only the temporal relations between spans.

A span is described using Clingo-DL to be an at least 1 time unit long temporal interval. The atom contains the Span ID and the Trace ID as arguments as shown in Listing ???. This allows traceability between the results of temporal reasoning and the real-world systems tracing framework.

The following examples use the Example Trace shown in Figure 7.1. The left side of the figure shows the relationship graph of the trace, while the right side shows the temporal structure of the trace.

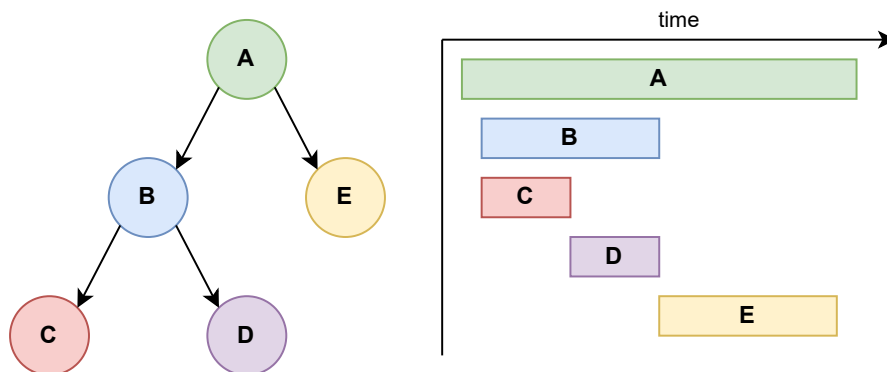


Figure 7.1: Example trace. Source [14], p5

### 7.1.2 Modeling from Timing data

Start and End time data from spans can be used to create a complete relation tree using Allen's Interval Algebras relations. The timing data must be sampled to allow time coincidence with other spans, as in real-world systems timing is continuous and when one span ends another may be beginning in logic, but won't start immediately but with a slight delay. Without this, only the overlap and during relations could be evaluated.

Timing data is always logged by tracing frameworks. Sometimes, links or relations may be absent or difficult to deduct. Creating relation trees from timing data allows for further analysis such as trace comparison, consistency checking, and other methods driven by relations. The ASP mapping of spans, using their start and endpoint, can be seen on Listing 7.1.

```
&diff {sp(SPAN)-ep(SPAN)} <= -1 :- span(SPAN,PARENT,TRACE,SP,EP) .
&diff {SP - sp(SPAN)} <= 0 :- span(SPAN,PARENT,TRACE,SP,EP) .
&diff {sp(SPAN) - SP} <= 0 :- span(SPAN,PARENT,TRACE,SP,EP) .
&diff {EP - ep(SPAN)} <= 0 :- span(SPAN,PARENT,TRACE,SP,EP) .
&diff {ep(SPAN) - EP} <= 0 :- span(SPAN,PARENT,TRACE,SP,EP) .
```

**Listing 7.1:** Span Description

From the created intervals, relations can be deduced using. As only one relation can be true between two intervals I chose to use a disjunctive rule, which declares, that exactly one relation shall be true. To introduce the relation between two intervals to the knowledge base the "relates" atom is used with its two arguments being the two intervals present in the relation. This can be seen on Listing ??.

```
1{relation(A, B, eq);relation(A, B, d);relation(A, B, p);relation(A, B, m);\\relation(A, B, o);
  relation(A, B, s);relation(A, B, f); relation(A, B, di);relation(A, B, pi);relation(A, B, mi);
  relation(A, B, oi);relation(A, B, si);relation(A, B, fi)}1 :- relates(A,B).
```

**Listing 7.2:** Deducing Relation

The following example shows the example span recreated using simplified timing data. I defined all spans using their start and endpoints. Clingo-DL then creates the intervals satisfying the description. Using the relates atom, I can find the relation between any two spans, in this case, span A and span B. This can be seen on Listing 7.3.

```
span(a, none, t, 0, 5).
  span(b, a, t, 1, 3).
    span(c, b, t, 1, 2).
      span(d, b, t, 2, 3).
        span(e, a, t, 3, 4).

relates(b, c).
```

**Listing 7.3:** Timing Data

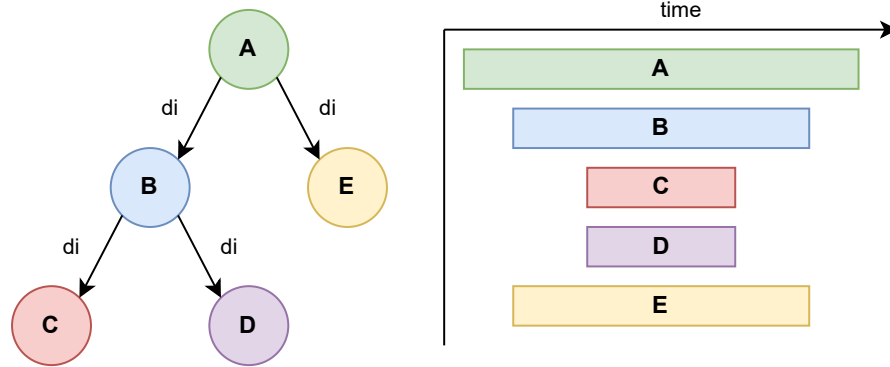
The solution contains all intervals created correctly, and the difference constraints are fulfilled as expected. The relation between span B and span C is solved as such:

$$\{relation(b, c, si)\}$$

### 7.1.3 Modeling from Relation Tree

Accurately reasoning over spans and traces requires a solid knowledge base about the relations between all spans. However, there may only be incomplete information about a trace. Tracing data must contain the parent-child hierarchy building a directed acyclic graph (DAG), where the first vertex is the root span. The edges are directed from the

parent spans to their child spans. In this DAG, all vertices can only have one edge directed to them but may have multiple or no outgoing edges. All edges represent the inverse of the During relation, also known as Contains. The graph and the span timeline recreated only from this information, based on the example trace shown on Figure 7.1 can be seen on Figure 7.2.



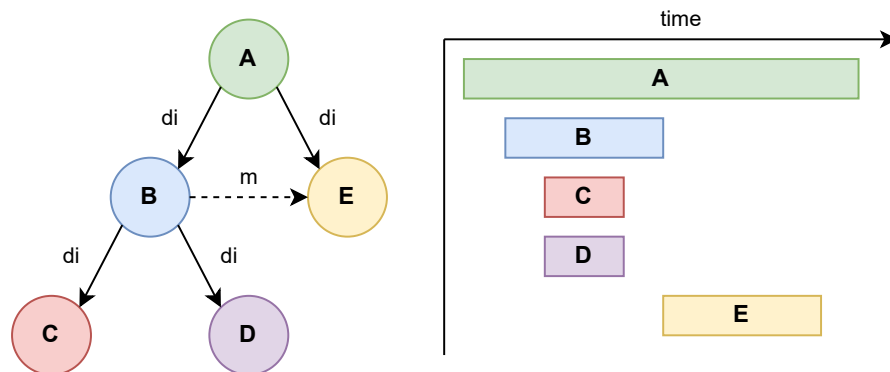
**Figure 7.2:** Trace from Parent-Child relation tree

This information by itself is not enough to reproduce an accurate representation of the temporal structure of this trace. The ASP implementation of this state of knowledge can be seen in Listing 7.4.

```
span(a, none, t).
relation(a, b, di).
relation(a, e, di).
  span(b, a, t).
  relation(c, b, di).
  relation(d, b, di).
    span(c, b, t).
    span(d, b, t).
span(e, a, di).
```

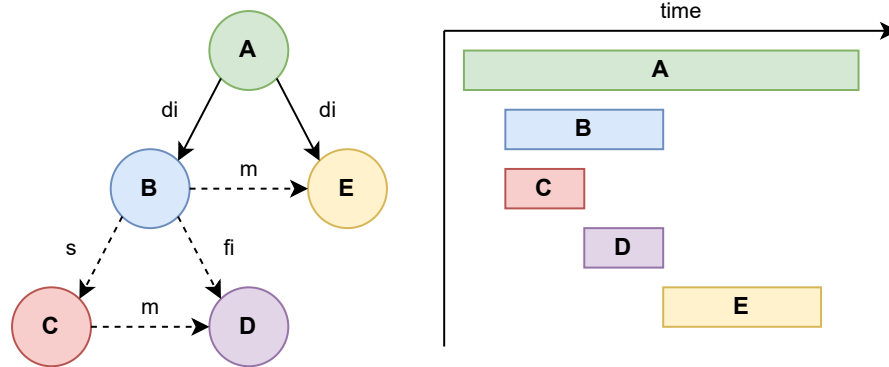
**Listing 7.4:** Relation Tree

Expanding this graph with other relations allows for more accurate temporal knowledge. This information can come from span links in the tracing data. The result can be seen on Figure 7.3 In this step, I introduce the fact, that B meets E, shown by the dashed line between vertices B and E. This can be added by system knowledge about the fact that E will begin when B ends. With this information, the knowledge base is closer to reality, however, it is still not completely accurate.



**Figure 7.3:** Trace from relation tree expanded with span link

Using more knowledge from span links, I can introduce more edges, thus including more edges in the relation tree, creating an accurate temporal structure of this example trace. For this, however, the system has to be instrumented with span links. The final result can be seen on Figure 7.4. The expanded ASP representation can be seen on Listing 7.5.



**Figure 7.4:** Complete Trace recreated from expanded relation tree

```

span(a, none, t).
relation(a, b, di).
relation(a, e, di).
  span(b, a, t).
    relation(c, b, s).
    relation(d, b, fi).
      span(c, b, t).
      span(d, b, t).
span(e, a, t).

relation(b,e,m).
relation(c,d,m).

```

**Listing 7.5:** Complete Knowledge

## 7.2 Applications

Various factors contribute to uncertainties in the operation of deployed systems:

- **Emergent properties:** The intricate details of how these services interact and function are not fully understood, leading to uncertainties in their behavior. This lack of comprehensive understanding gives rise to a plethora of uncertainties in their behavior. Developers may struggle to predict how these services will respond to various inputs or under different conditions due to the obscurity surrounding their internal mechanisms.
- **Actual and expected behavior:** Discrepancies between the expected or documented behavior and the real-time execution and calls within the environment have the potential to introduce a significant degree of uncertainty. These disparities can emerge due to a variety of factors, such as unforeseen bugs, and irregularities in the underlying hardware or software configurations.
- **Flexible and adaptive behavior:** Even a single type of request can yield different outcomes based on a multitude of conditions. These conditions encompass a wide array of variables, including user input, system load, network latency, and the current state of the underlying hardware and software components. The intricate interplay of these variables introduces unpredictability into system operations.



- **Separation of sub-systems:** The separation of sub-systems within a larger system architecture results in unknowns in different parts of the system and their connections. This separation can lead to complexities where the interactions and dependencies between these subsystems are not entirely clear, adding to the overall uncertainties in the system's operation.

### 7.2.1 Missing Span Substitution

Issues may arise from incomplete instrumentation or the use of external services, which cause spans to be missing from the tracing data. However, there may be system knowledge about this span and the relations compared to other, known spans. Using this information, I can extend an existing trace with a span that was not recorded during runtime.

In this example, the D span was not recorded, but it is known from system knowledge that it must be somewhere in this trace. The initial knowledge can be seen on Figure 7.5.

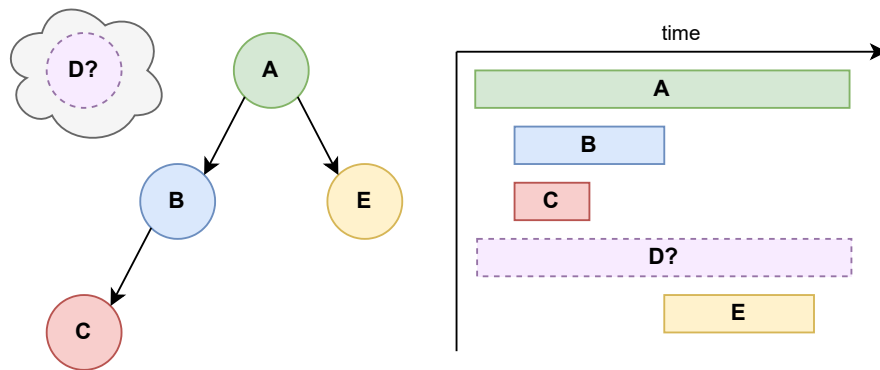


Figure 7.5: Missing Knowledge

Two temporal relations are known about span D. D finishes B ( $B \text{ fi } D$ ), and C meets D. Using these two relations, the program can find a solution, where D fits in the trace. The result can be seen on Figure 7.6. The ASP model of this can be seen on Listing 7.6.

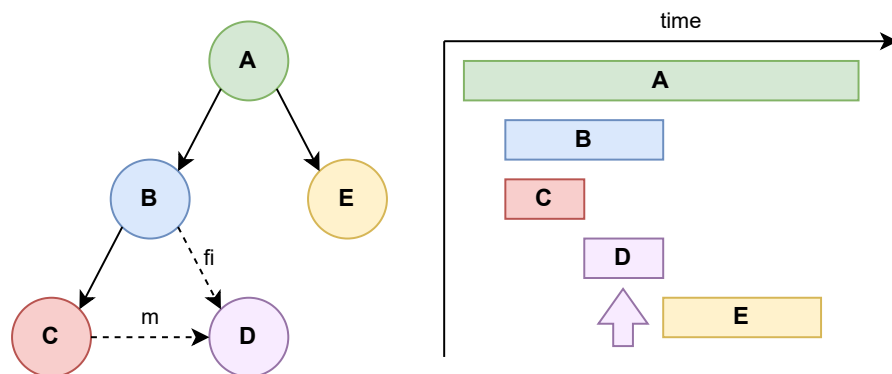


Figure 7.6: Finding the Missing Span

As the span and trace IDs are both also contained in the ASP model. These properties are enough to identify a span with complete certainty. This allows for traceability between the ASP model and the tracing framework. The missing span can be reintroduced to complete the trace.

```

span(a, none, t, 0, 5).
  span(b, a, t, 1, 3).
    span(c, b, t, 1, 2).
      % d missing
    span(e, a, t, 3, 4).

relation(d, b, f).
relation(c, d, m).

```

**Listing 7.6:** Missing Knowledge

## 7.2.2 Behavior Consistency Check

Defining constraints over span relations can be used to find abnormal traces. This is a valuable tool when analyzing a large data set. The temporal relations are able to uncover unexpected behavior from the system. I implemented this method using two atoms, which constrain relations between two traces as shall be or shall not be. This allows the creation of a set of relations to which two spans are constrained. If these constraints are broken, the trace contains an error. These atoms are shown in Listing 7.8.

```

relates(A,B) :- relation_shall_be(A,B,REF).
error(A,B,REF,REL) :- relation_shall_be(A,B,REF), relation(A,B,REL), REF != REL.

relates(A,B) :- relation_shall_not_be(A,B,REF).
error(A,B,REF,REL) :- relation_shall_not_be(A,B,REF), relation(A,B,REL), REF = REL.

```

**Listing 7.7:** Relation Constraints

```

span(a, none, t, 0, 5).
  span(b, a, t, 1, 3).
    span(c, b, t, 2, 3).
      span(d, b, t, 2, 3).
    span(e, a, t, 3, 4).

% Only allow C precedes D and C meets D.
% Example System Requirement:
% C must be executed by the time D begins.
relation_shall_not_be(c,d,pi).
relation_shall_not_be(c,d,mi).
relation_shall_not_be(c,d,o).
relation_shall_not_be(c,d,oi).
relation_shall_not_be(c,d,d).
relation_shall_not_be(c,d,di).
relation_shall_not_be(c,d,f).
relation_shall_not_be(c,d,fi).
relation_shall_not_be(c,d,e).

```

**Listing 7.8:** Relation Constraints

Solving this program returns the following solution:  $\{relation(c,d,eq) error(c,d,eq,eq)\}$  As the C and D spans interval relation is equal, breaking the imposed constraints.

### 7.2.3 Trace Comparison

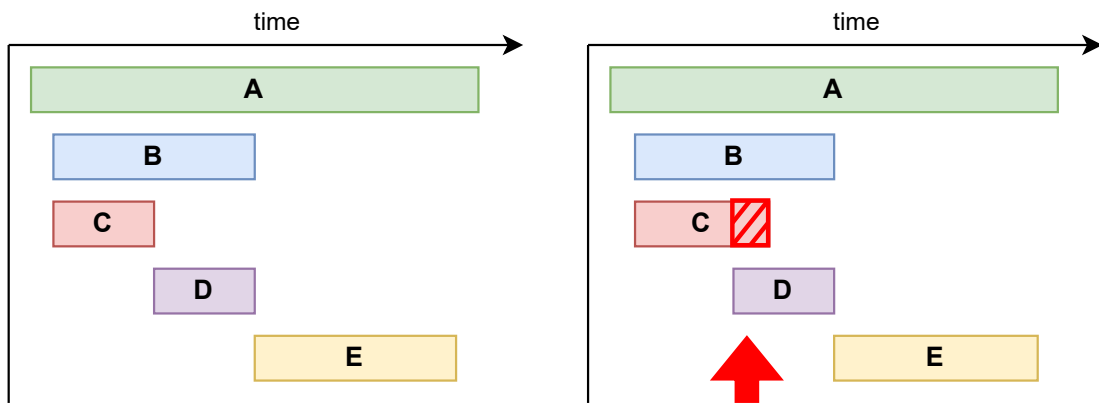
Trace comparison is an invaluable tool when analyzing anomalous behavior in the system. Span relations are handled with a relationship matrix. Comparing the relationship matrices of the compared traces, the differences can be evaluated. With this method, the uncertainties within the execution of a type of request can be evaluated.

Differences in the two relationship matrices can be seen on Table 7.1 marked in red. The elements inside the matrix show the relationship between the trace on the left side and the trace on the top side. For example, in Table 7.1 of the Left Traces matrix, the intersection of D on the left header and A on the top header is d, meaning D is during A in the Left Trace. After evaluating the temporal relations in a trace, a matrix can be created. Comparing these two matrices shows the comparison of temporal relations between the two examined traces. This difference, marked with a red arrow, is visualized in Figure 7.7.

\	A	B	C	D	E
A	e	di	di	di	di
B	d	e	si	fi	m
C	d	s	e	<b>m</b>	p
D	d	f	<b>mi</b>	e	m
E	d	mi	pi	mi	e

\	A	B	C	D	E
A	e	di	di	di	di
B	d	e	si	fi	m
C	d	s	e	<b>o</b>	p
D	d	f	<b>oi</b>	e	m
E	d	mi	pi	mi	e

**Table 7.1:** Relationship Matrices of the Left and Right Trace



**Figure 7.7:** Comparison of the traces

## 7.2.4 Merge

Due to inadequate configuration, multiple subsystems may provide two separate traces from a single main request. This situation can create a fragmented view of the request's journey, making it challenging to comprehensively analyze its flow and pinpoint potential issues. However, merging these multiple traces, which collectively constitute a single request, can prove to be immensely beneficial during operational activities.

In this example, the root span of the main request is span A. Span A has a child span, span B, which is a request to a separated sub-system. Due to this, the path that the request takes inside the B sub-system is obscured from the original span. As distributed tracing is also implemented in the sub-system this path is known in a separate trace, recorded by the sub-system. Using the knowledge about the execution of this request, I can add the fact that span B and span X are equal. With this fact, the complete trace can be reconstructed.

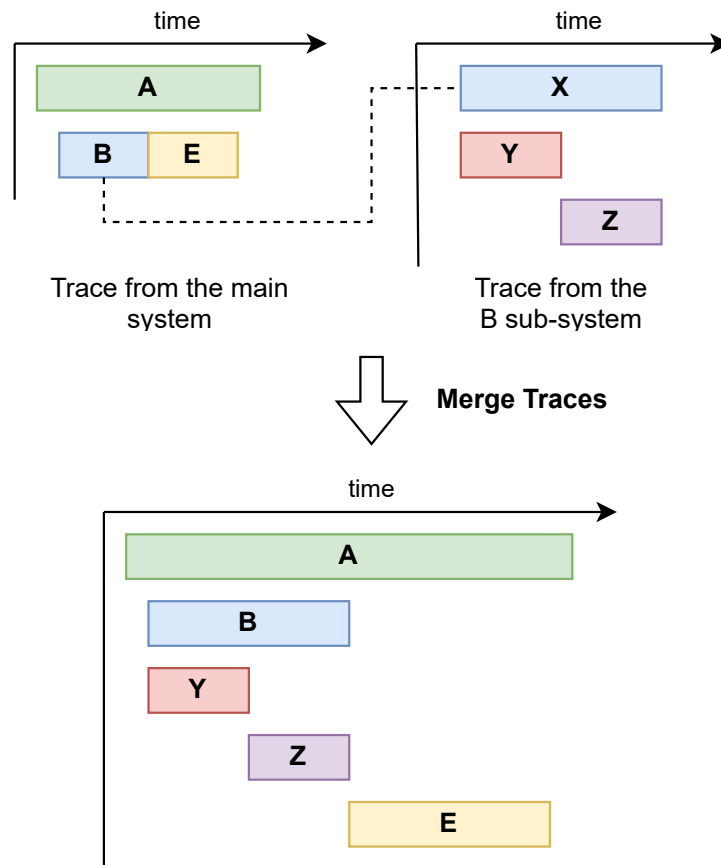


Figure 7.8: Trace Merge.

## Chapter 8

# Proposed Approach: Qualitative Diagnosis over Distributed Traces

The interconnected nature of services, spans, and both physical and virtual nodes form a complex network. Studying how information, faults, and other metadata propagate within this intricate system is crucial for effective planning, modeling, and operation. Despite the significance of this analysis, there is frequently a scarcity of relevant data available. This report addresses this challenge by introducing methodologies that rely on qualitative factors like the change in the minimum and maximum execution time of spans. Additionally, these methodologies consider other well-known properties of the system, such as service structure and the architecture of both physical and virtual components. By integrating these diverse elements, I present a qualitative diagnosis approach to understanding the behavior of large-scale microservice-based systems.

This approach can be used to extend the Temporal Methods described in the previous chapter. As they view the system from a different perspective, their concurrent use can yield a greater reduction of uncertainties.

### 8.1 Qualitative Approach

The methods described in this chapter use a subset from the area of Qualitative Reasoning. For this use, I considered the change (increase, steady, decrease) of significant timing characteristics in aggregates of a type of span, minimum execution time, and maximum execution time. These quantities are derived from the real-world timing data of traces collected in a certain time frame. The two quantities are assigned to each span. Because of this, the model only contains Proportional Relations, and a simplified form of delta calculus for calculating the effect of multiple influences.

### 8.2 Propagation Analysis

Error propagation analysis (EPA) is a systematic model-based approach to assess the impact of incidental or malicious faults in the dependability and security analysis of complex systems. [10]

EPA is able to discover the effects and path of an initial error using the qualitative model of the system. This is used in this report to discover paths that lead to accidents.

In our case propagation may also be used in non-fault conditions, to assess the effect of a change in the system.

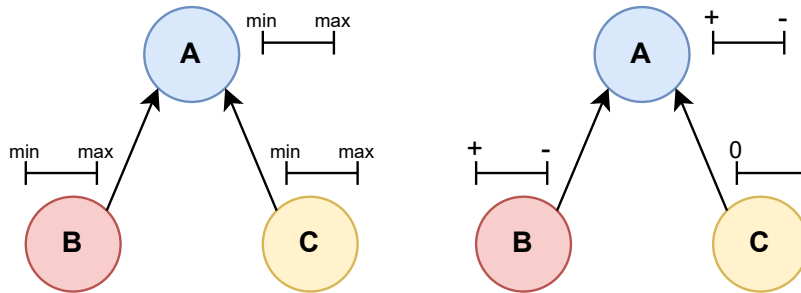
### 8.3 Mapping Traces and Spans

Traces and their spans by themselves contain fairly little information by default. Instrumenting the code base with status messages, error codes, and other metadata requires a common standard, which may prove difficult in large-scale systems or where commercial off-the-shelf software is being used. A common denominator between all tracing systems and standards is the start and end time of a span. Using only this data allows usage across all possible systems and frameworks.

Sampling traces in a given time frame allows us to calculate the minimum and maximum execution times of the spans. Common fault modes of the components can often be connected to a specific type of minimum and maximum time frame change. For example, in the case of a component outage, the minimum and maximum duration both decrease, as all requests immediately fail after being sent. If a component is rate-limited, it may cause the minimum duration to decrease, as the component denies some requests immediately. However, some requests will be successful, but still slower than during normal operation due to the high load.

The spans of a common trace type (request) are vertices in a directed acyclic graph (DAG). The edges are directed from lower-level spans to their parent span. They could also point at other higher-level spans if this influence is known from other knowledge.

In this DAG, microservices and physical computers or other elements can also be included, which influence the system behavior as vertices. Edges from these vertices shall be directed to other vertices which are influenced by them. In this case, the minimum and maximum duration do not mean the execution time of the vertices, but the way they influence the connected spans.



**Figure 8.1:** Propagation

```
n_inc(A, lp, N) :- span(A), POS_INC = #count{B: influence(A, B, pos), lp(B, increasing)},
  NEG_DEC = #count{B: influence(A, B, neg), lp(B, decreasing)}, N = POS_INC + NEG_DEC.

n_dec(A, lp, N) :- span(A), POS_DEC = #count{B: influence(A, B, pos), lp(B, decreasing)},
  NEG_INC = #count{B: influence(A, B, neg), lp(B, increasing)}, N = POS_DEC + NEG_INC.
```

**Listing 8.1:** Collecting influences

```

% n_inc > 0 && n_dec = 0 => +
lp(A, increasing) :- n_inc(A, lp, X), X > 0, n_dec(A, lp, 0).
% n_inc = 0 && n_dec = 0 => 0
lp(A, steady) :- n_inc(A, lp, Y), Y = 0, n_dec(A, lp, Y), Y = 0, not lp(A, increasing), not lp(A,
    decreasing).
% n_inc = 0 && n_dec > 0 => -
lp(A, decreasing) :- n_inc(A, lp, 0), n_dec(A, lp, X), X > 0.
% n_inc > 0 && n_dec > 0 => ?
1{lp(A, increasing); lp(A, steady); lp(A, decreasing)}1 :- n_inc(A, lp, X), X > 0, n_dec(A, lp, Y), Y
    > 0.

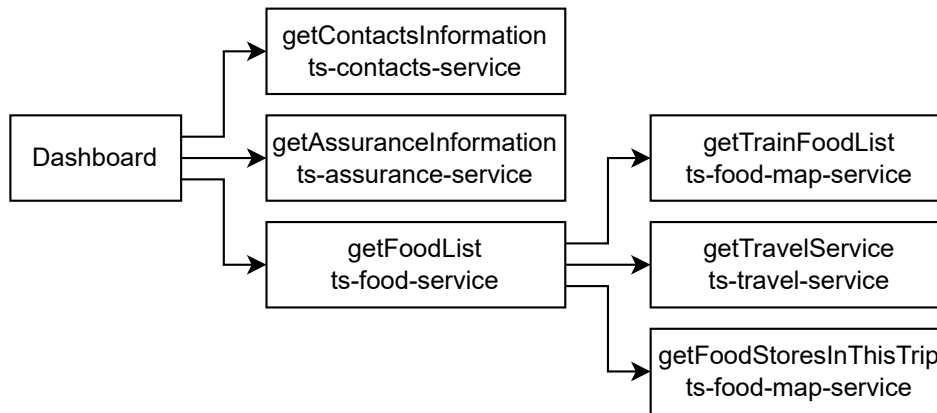
```

**Listing 8.2:** Calculating influences

## 8.4 Applications

### 8.4.1 Example Trace

In the following sections, the use cases will be showcased with an example trace, called Query Ticket Information, the service graph can be seen on Figure 8.2. This activity originates from a UI request based on the Dashboard. In the blocks, the upper line shows the span name, and the lower line shows the provided service. It should be noted that the `getTrainFoodList` and the `getFoodStoresInThisTrip` request are both based on the `ts-food-map-service`. The rest of the spans are all handled by their own separate service, to simplify the diagram and the ASP code, these services are not considered or shown separately. The ASP model of this trace can be seen in Listing 8.3.



**Figure 8.2:** Query Ticket Information Service Graph

```

span(dashboard).
  span(getContactsInformation).
  span(getAssuranceInformation).
  span(getFoodList).
    span(getTrainFoodList).
    span(getStations).
    span(getFoodStores).

influence(dashboard,getContactsInformation,pos).
influence(dashboard,getAssuranceInformation,pos).
influence(dashboard,getFoodList,pos).
  influence(getFoodList,getTrainFoodList,pos).
  influence(getFoodList,getStations,pos).
  influence(getFoodList,getFoodStores,pos).

```

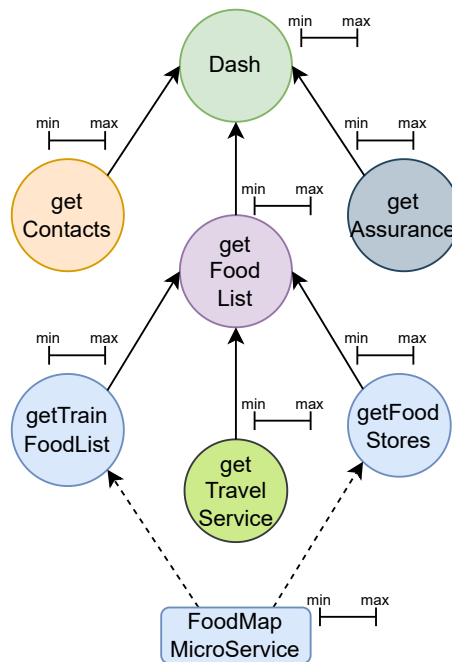
**Listing 8.3:** Span description

## 8.4.2 Diagnostics

To create diagnostics from observations, abductive reasoning must be used. Creating plausible conclusions from the known duration changes in traces. In this mode of operation, there is no knowledge of the properties of the non-span vertices in the DAG. There may also be missing data from spans.

Abductive reasoning, often referred to as inference to the best explanation, is a vital form of logical inference where uncertainties are reduced based on the most plausible explanation for a set of observations and knowledge. When faced with incomplete or ambiguous information, abductive reasoning enables to exploration of various possibilities and proposes explanations that can account for the available evidence.

Figure 8.3 shows the base propagation graph containing all spans and the relevant microservices of the trace. In normal usage all microservices would be included, however, in this example, all spans except `getTrainFoodList` and `getFoodStores` are contained in a separate microservice. For ease of understanding these microservices are not included in this model.



**Figure 8.3:** Trace Structure in Propagation

I will showcase the use of the diagnostics using a simple example, shown in Figure 8.4. We have the relevant timing data from the distributed tracing framework, showing, that the minimum and maximum execution time of spans `Dash`, `getFoodList`, `getTrainFoodList`, and `getFoodStores` both increased. The timing statistics remained steady for spans `getContacts`, `getAssurance`, and `getTravelService`. The ASP representation of this information is shown in Listing 8.4.

```
%symptoms
lp(getFoodList, increasing).
hp(getFoodList, increasing).
lp(getContacts, steady).
hp(getContacts, steady).
```

**Listing 8.4:** Part of the ASP propagation input



In this example, three types of faults are considered for the foodMap microservice, outage, highload, and rate limit. These and their possible effect in the propagation model can be seen in detail on Listing 8.5.

```

%fault(foodMapSERVICE, outage).
lp(foodMapSERVICE, decreasing) :- fault(foodMapSERVICE, outage).
hp(foodMapSERVICE, decreasing) :- fault(foodMapSERVICE, outage).

%fault(foodMapSERVICE, highload).
lp(foodMapSERVICE, increasing) :- fault(foodMapSERVICE, highload).
hp(foodMapSERVICE, increasing) :- fault(foodMapSERVICE, highload).

%fault(foodMapSERVICE, ratelimit).
lp(foodMapSERVICE, decreasing) :- fault(foodMapSERVICE, ratelimit).
ep(foodMapSERVICE, increasing) :- fault(foodMapSERVICE, ratelimit).

```

**Listing 8.5:** Fault Modes

To find the source of the quantity changes in the propagation graph, one of the faults must be in effect. This is defined in ASP using disjunctive rules, shown in Listing 8.6.

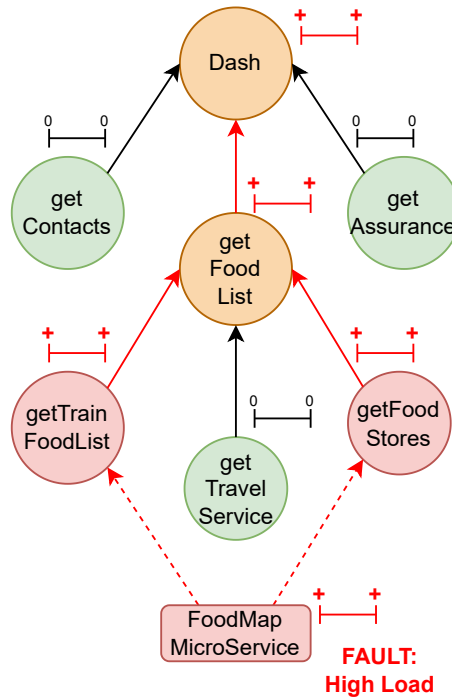
```

1{fault(foodMapSERVICE, highload); fault(foodMapSERVICE, outage); fault(foodMapSERVICE, ratelimit)}1.

```

**Listing 8.6:** Disjunctive Fault rule

After running the program, abductive reasoning gives us the fault that caused the discrepancies. The root cause was a fault of the High Load fault mode in the FoodMap microservice. The path of the error can be seen on 8.4.



**Figure 8.4:** Running Propagation - High Load fault mode

### 8.4.3 What-ifs

Experimenting on deployed systems is costly and may cause a worse user experience. This propagation-based model allows cheap and quick evaluation of possible scenarios. Due to its abstraction, exact results can not be made, however, this could be further refined in the future.

This approach allows for a clear understanding of the relationships between different variables and how errors in one variable can propagate through the system, influencing the system’s behavior. Through deductive reasoning, engineers can identify vulnerable points in the system, enabling them to focus on minimizing faults in these critical areas, thus enhancing the overall reliability of the system.

In the example shown on Figure 8.5, the scenario defines a Rate Limit fault in the FoodMap microservice, and the increase of the minimal execution time in the getAssurance span. From this using deductive reasoning the solution can be found. These scenarios will lead to the overall request maximum execution time to increase. Due to the abstraction of Qualitative Reasoning the change of the minimum execution time can not be determined.

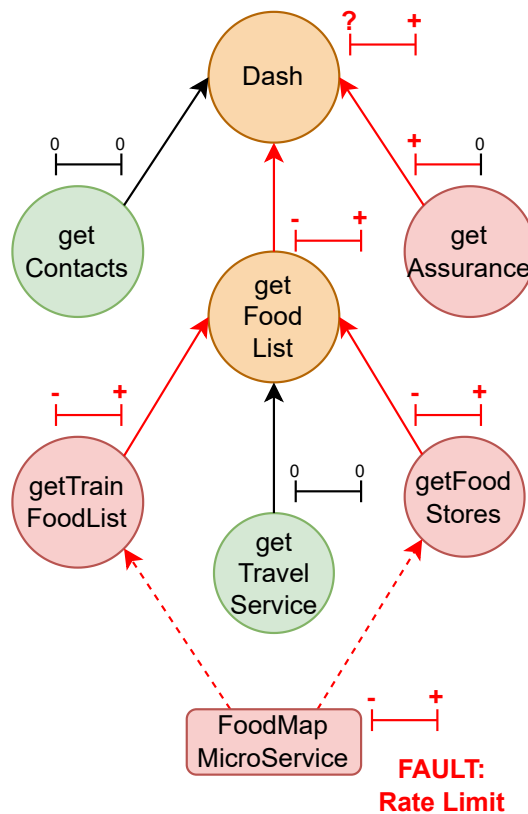


Figure 8.5: What if? example

#### 8.4.4 Co-deployment Design and Evaluation

Deploying microservices on the same node offers several advantages in terms of resource optimization and efficient utilization of computing resources. Sharing the same node allows for streamlined resource allocation, making it easier to scale services horizontally and manage workloads effectively. Furthermore, deploying microservices on a single node simplifies deployment and monitoring processes. However, careful consideration must be given to resource allocation, load balancing, and fault tolerance to ensure optimal performance and reliability when co-locating microservices on a single node.

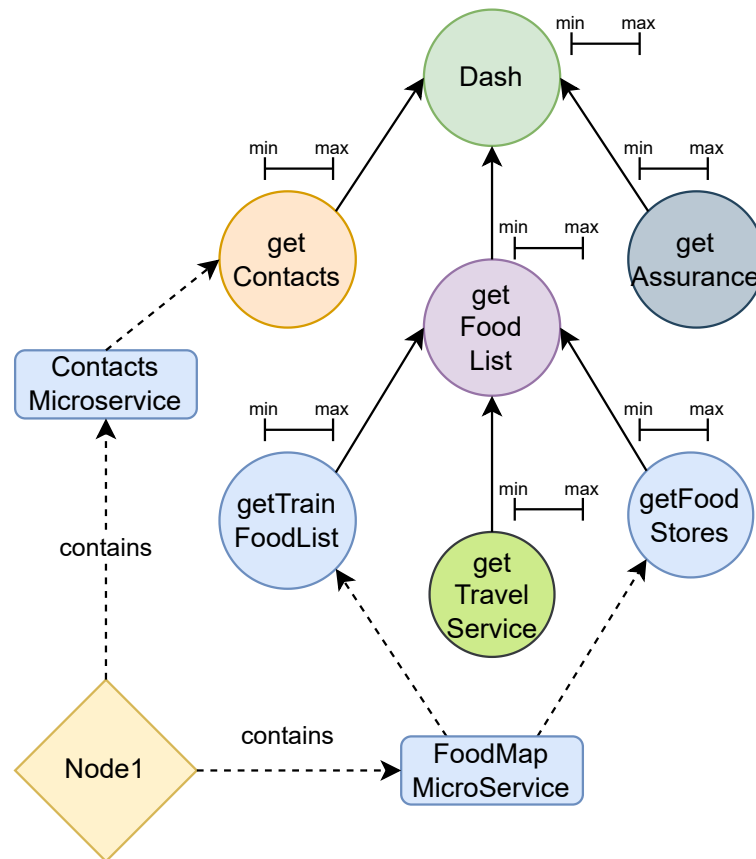
During deployment, it is crucial to model and assess the outcomes of co-deployment to guarantee the system operates correctly. Understanding how different services interact and influence one another when deployed together is essential for achieving optimal system behavior. The framework introduced in this report, encompassing propagation analysis

and the Qualitative Diagnostics methodology, serves as a powerful tool in comprehensively studying the effects of co-deployment.

Modeling and evaluating the result of this co-deployment is paramount to ensure optimal system behavior. Using propagation and the Qualitative Diagnostics framework developed in this report the effects of co-deployment can be studied and also taken into consideration during operations. My approach can also be used to ensure that the co-deployed services not only function as anticipated but also verifying that they are indeed hosted on the same node.

An example of co-deployment in the propagation graph can be seen on Figure 8.6. Where the Contacts microservice and the FoodMap microservice are both hosted on the same node, marked Node1.

The boundaries of this report did not allow to completely explore the possibilities of this methodology.



**Figure 8.6:** Co-deployment of the Contacts and FoodMap microservice

# Chapter 9

## Summary

### 9.1 Conclusion

In this report I showcased two novel approaches to reasoning across traces from distributed tracing. The first approach is a temporal method, where I considered spans as intervals in Allen's Interval Algebra to reduce uncertainties about system behavior using temporal knowledge. In the second approach I used Qualitative Reasoning and Error Propagation across a knowledge base created from aggregate timing statistics of traces. This allows root cause diagnosis, impact analysis, and codeployment evaluation without the need for instrumentation, as fault modes can be defined using the statistical anomalies of recorded span timing data. I explored the benefits of both of these novel approaches on several possible applications for microservice based system designers and operators.

### 9.2 Further Work

The possibilities in using logic reasoning across distributed traces are extensive. This report scratches the surface of possibilities. To further enhance these novel approaches and evaluate their effectiveness in real-world systems, I have the following plans:

- **Magnitudes in Qualitative Reasoning** would benefit the more accurate modeling and reasoning of my Qualitative Approach. As spans differ greatly in terms of timing magnitude, some may be only nanoseconds long, while others can take as long as multiple seconds.
- **Testing with real-world trace data** is the next step to evaluating the usefulness of these novel approaches. Creating an efficient parser to place the reasoning engine inside the distributed tracing workflow is imperative to further testing and evaluation.
- **Deployment** using the TrainTicket benchmarking system could bring light the new strengths and weaknesses. Further testing using fault injection could help evaluate the need for this approach and the efficiency of it as well. This step is also needed to study the use of the codeployment evaluation methodology.
- **Benchmarking** is important for such approaches. As distributed tracing systems create incredibly vast amounts of data, it is important for these methods to work efficiently and with low overhead.

# Acknowledgements

I want to express my sincere appreciation to my advisors, András Földvári and Imre Kocsis, for their enduring support, encouragement, and invaluable guidance. Their expertise, patience, and commitment have played a pivotal role in the completion of this report.

# Bibliography

- [1] James F Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, 1983.
- [2] Christian Anger, Kathrin Konczak, Thomas Linke, and Torsten Schaub. A glimpse of answer set programming. *Künstliche Intell.*, 19(1):12, 2005.
- [3] Andre Bento, Jaime Correia, Ricardo Filipe, Filipe Araujo, and Jorge Cardoso. Automated analysis of distributed tracing: Challenges and research directions. *Journal of Grid Computing*, 19:1–15, 2021.
- [4] CodeWisdom Fudan University. Train Ticket: A Benchmark Microservice System (2021.). <https://github.com/FudanSELab/train-ticket/>.
- [5] Martin Gebser, Benjamin Kaufmann, Roland Kaminski, Max Ostrowski, Torsten Schaub, and Marius Schneider. Potassco: The potsdam answer set solving collection. *Ai Communications*, 24(2):107–124, 2011.
- [6] Javad Ghofrani and Daniel Lübke. Challenges of microservices architecture: A survey on the state of the practice. *ZEUS*, 2018:1–8, 2018.
- [7] Tomi Janhunen and Michael Sioutis. Allen’s interval algebra makes the difference. In *International Conference on Applications of Declarative Programming and Knowledge Management*, pages 89–98. Springer, 2019.
- [8] Tomi Janhunen et al. Clingo goes linear constraints over reals and integers. *Theory and Practice of Logic Programming*, 17(5-6):872–888, 2017. DOI: 10.1017/S1471068417000242.
- [9] Kenneth D. Forbus. Handbook of Knowledge Representation, Chapter 9 Qualitative Modeling (2008). <https://web.stanford.edu/class/cs227/Lectures/lec13.pdf>.
- [10] András Pataricza. Model-Based Dependability Analysis. DSc thesis. 2008.
- [11] Prof. Vinay K. Chaudhri, Stanford. CS 227: Knowledge Representation and Reasoning, Lecture 13. Qualitative Reasoning (spring 2011). <https://web.stanford.edu/class/cs227/Lectures/lec13.pdf>.
- [12] Benjamin H Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspán, and Chandan Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. 2010.
- [13] The OpenTelemetry Authors. OpenTelemetry Documentation (2023.). <https://opentelemetry.io/docs/>.

- [14] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chenjie Xu, Chao Ji, and Wenyun Zhao. Benchmarking microservice systems for software engineering research. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings*, ICSE '18, page 323–324, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356633. DOI: 10.1145/3183440.3194991. URL <https://doi.org/10.1145/3183440.3194991>.