UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA

Detection of transaction consistency problems in microservices

André Filipe dos Santos Silva

Mestrado em Engenharia Informática

Dissertação orientada por:
Prof. Doutora Maria Antónia Bacelar da Costa Lopes
Prof. Doutor Alcides Miguel Cachulo Aguiar Fonseca

2024

Acknowledgments

First and foremost, I would like to express my sincere gratitude to my two teachers, Antónia Lopes and Alcides Fonseca, for your expertise, guidance and patience throughout this journey.

I also want to thank my friends and colleagues at Premium Minds, whose persistent reminders and occasional annoyance kept me on track to complete this thesis.

Lastly, I want to thank my closest family for your support and belief in me throughout my entire academic journey.

This work is financed by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia within project DACOMICO (PTDC/CCI-COM/2156/2021).

Resumo

A arquitetura de microserviços destaca-se como uma solução para aumentar a flexibilidade e a escalabilidade dos sistemas de software. Contudo, apesar dos seus benefícios, a adoção desta arquitetura traz consigo desafios significativos, especialmente no que diz respeito à consistência dos dados distribuídos entre os vários serviços. Uma das principais dificuldades que emergem neste contexto é garantir a consistência das transações, uma vez que, ao contrário das arquiteturas monolíticas, os microserviços frequentemente perdem a propriedade de isolamento que é crítica para manter a integridade dos dados.

Esta tese aborda o problema da deteção de anomalias de consistência em transações dentro de sistemas baseados em microserviços. As anomalias de consistência, tais como as relacionadas com a serialização incorreta de operações, podem levar a estados de dados inconsistentes, comprometendo a fiabilidade dos sistemas. O objetivo principal deste trabalho é desenvolver uma abordagem capaz de identificar antecipadamente estas anomalias, permitindo que os desenvolvedores possam corrigi-las antes que o sistema seja implementado.

No contexto da pesquisa sobre consistência de dados em sistemas distribuídos, esta tese fundamenta-se em conceitos estabelecidos tanto nas arquiteturas monolíticas quanto nas de microserviços. A literatura existente destaca a dificuldade de manter a consistência em sistemas que utilizam microserviços devido à perda de isolamento e às transações distribuídas. Diversas abordagens têm sido propostas para enfrentar esses desafios, incluindo ferramentas de análise estática e dinâmica, como CLOTHO e ACIDRain, que se focam na deteção de anomalias de serialização em diferentes cenários. No entanto, essas ferramentas apresentam limitações quando aplicadas diretamente a arquiteturas de microserviços, onde as operações distribuídas e as complexas interações entre serviços requerem soluções mais especializadas. Esta tese contribui para o tema ao propor uma abordagem específica para microserviços, complementando e estendendo as técnicas existentes.

Para enfrentar este tema, o STRIVE foi desenvolvido, um algoritmo que utiliza técnicas de análise estática para detetar potenciais anomalias de serialização em microserviços. A abordagem baseia-se na modelação das operações do sistema e nas suas interações através de um grafo. Neste grafo, as operações de base de dados são representadas como nós, e as dependências entre elas são representadas como arestas. O algoritmo foca-se na deteção de ciclos dentro deste grafo, uma vez que os ciclos indicam potenciais violações de consistência. Se um ciclo for detetado, isso sugere que há uma sequência de operações que poderia levar o sistema a um estado inconsistente.

Além do desenvolvimento do STRIVE, esta tese inclui também o design de uma Linguagem Específica de Domínio (DSL), que facilita a descrição dos sistemas de microserviços. Esta DSL permite que os desenvolvedores modelam os seus sistemas de forma abstrata, sem a necessidade de escrever código de baixo nível. A DSL foi projetada para ser intuitiva, permitindo a definição de operações,

transações e fluxos de dados, o que é essencial para a análise de anomalias de consistência. A modelação em alto nível proporcionada pela DSL é transformada pelo STRIVE num modelo interno que pode ser analisado para a deteção de anomalias.

Os resultados obtidos com o STRIVE são demonstrados através de estudos de caso, que ilustram a aplicação prática da abordagem proposta. Os estudos de caso incluem a migração de sistemas monolíticos para arquiteturas de microserviços, onde os desafios de consistência são mais pronunciados. A comparação com outras ferramentas existentes mostra que o STRIVE oferece uma solução igualmente eficaz para a deteção de anomalias, destacando-se pela sua capacidade de realizar esta análise de forma antecipada, durante a fase de desenvolvimento, num sistema estruturado com microserviços. No entanto, apesar das contribuições apresentadas por este trabalho, há várias áreas que permanecem abertas para pesquisa futura. Uma delas é a extensão do algoritmo para suportar níveis adicionais de isolamento, o que permitiria uma análise ainda mais abrangente das possíveis anomalias. Além disso, há potencial para melhorar a DSL, tornando-a mais fácil de usar e mais expressiva, de modo a capturar um maior número de detalhes do sistema. Um passo crucial para o futuro é o desenvolvimento de um parser que automatize a tradução das descrições feitas em DSL para a representação interna utilizada pelo STRIVE, simplificando ainda mais o processo de modelação e análise.

A presente tese constitui um passo no sentido de melhorar a fiabilidade das arquiteturas de microserviços. Ao permitir a deteção proativa de problemas de consistência nas transações, o STRIVE oferece uma ferramenta prática e eficaz para os engenheiros de software, contribuindo para a robustez dos sistemas distribuídos. No entanto, a complexidade crescente dos sistemas modernos exige uma evolução contínua das ferramentas e abordagens aqui apresentadas, para que possam ser plenamente aplicáveis a um leque mais vasto de cenários e desafios.

Em resumo, o trabalho aqui desenvolvido não só contribui para a compreensão dos desafios inerentes à consistência dos dados em arquiteturas de microserviços, mas também propõe uma solução prática para a sua deteção e correção. Esta solução, embora promissora, requer melhorias contínuas e adaptações para enfrentar as novas exigências impostas pela evolução das tecnologias e das arquiteturas de software. A continuidade deste trabalho deverá focar-se na expansão das capacidades do STRIVE e da DSL.

Palavras-chave: Microserviços, Consistência de Dados, Transações Distribuídas, Análise Estática, Linguagem Específica de Domínio (DSL)

Abstract

The widespread adoption of microservices architecture in modern software development has introduced significant challenges in maintaining data consistency across distributed services. This thesis addresses the problem of detecting transaction consistency anomalies within microservices-based systems, which are prone to issues due to the loss of isolation in distributed transactions. To tackle this problem, we propose STRIVE; a novel approach that leverages static analysis techniques to identify potential serialization anomalies in microservices. By modeling system operations and their interactions as a graph, the algorithm detects cycles that indicate possible consistency violations. This method enables the early detection of anomalies during the development phase, thereby preventing data inconsistencies that could arise after deployment. The work includes the design of a Domain-Specific Language (DSL) to facilitate the description of microservices systems, allowing developers to model their systems at a high level of abstraction without needing low-level code. The effectiveness of STRIVE is demonstrated through case studies, showing its ability to identify potential issues in various microservices architectures. Furthermore, we also identify several areas for future exploration, including the extension of the algorithm to support additional features and the refinement of the DSL. Towards improving the reliability of microservices architectures, this thesis provides a tool that enables the proactive detection of transaction consistency issues, contributing to the broader effort of making distributed systems more robust and reliable.

Keywords: Microservices, Data Consistency, Distributed Transactions, Static Analysis, Domain-specific language (DSL)

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

Many existing enterprise applications that are structured according to well-known and popular architectural patterns [10] have a monolith architecture. In these systems, the components responsible for the different business functionalities are all bundled into a single deployable element and typically share a single database.

Applications with a monolith architecture are generally easier to develop than distributed applications. However, these benefits come at the cost of scalability, performance and maintainability. Microservices architecture, often shortened to microservices, is a recently proposed architectural style to counter these disadvantages. In this style, applications are structured as a set of loosely coupled services that can be developed and deployed independently. Each service encapsulates a small and well-defined set of business functionalities and runs in its own process and database. Systems with microservice architecture are much easier to scale and evolve, making this architecture well-suited for the modernization of many long-lived monolithic applications.

The migration of monolithic applications to microservices is not trivial[1]. One of the main problems is to ensure data consistency, namely maintaining data consistency across services and obtaining a consistent view of the data. This is because the database, which was once centralized, becomes partitioned and distributed between different services. Hence, system operations might need to update data in multiple databases. In the monolith, system operations could rely on the properties of ACID transactions to guarantee isolation and a consistent view of the database [12]. In microservices, it is also possible to implement distributed transactions with similar guarantees, for instance, using the two-phase commit protocol [11]. However, this requires that the system's different microservices put a lock around their information to create a consistent view of the data for the active global distributed transaction, which creates a bottleneck on the entire system, defeating the purpose of improving scalability (one of the major advantages of the architectural style). Another approach is to coordinate the overall distributed state change using local transactions together with compensating mechanisms, like Sagas [1], that do not rely on locking and do not create such a bottleneck. Sagas are more complex than traditional ACID transactions to implement and are error-prone. Moreover, they only guarantee eventual consistency.

Once a monolithic application is split into´ several microservices unless distributed transactions are used, isolation is lost and concurrent system operations can see the effects of each other, and data

[1] martinfowler.com/articles/break-monolith-into-microservices.html

consistency anomalies can occur. Business logic can end up making decisions based on inconsistent information.

Consider, for instance, a bank system that allows users to perform bank operations such as deposit and withdraw as well as consult and update information about the customers (for example, name, address and credit rating) and the accounts (for example, balance). In this system, there is also a need for dynamic credit rating adjustments based on user behavior and transaction history. This dynamic adjustment ensures that customers with responsible financial behavior are rewarded with credit rating increases, while those with risky behavior have their ratings reduced. The withdrawal operation is particularly interesting as the maximum amount a user can withdraw is influenced by the credit rating, and after the withdraw, a dynamic credit rating adjustment is conducted, meaning that this operation needs both account and customer data. Apart from the withdraw, the credit rating can also be adjusted directly.

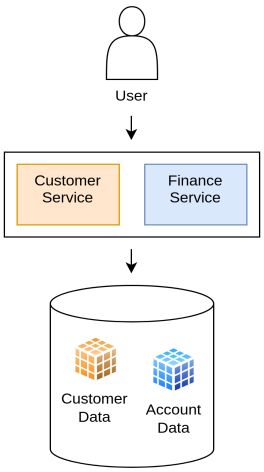

Figure 1.1: Bank system with a Monolith Architecture

Figure 1.1 presents a version of this system with a monolithic architecture. The business logic is implemented in Finance Service and Customer Service, two modules of the same codebase that are deployed together. The Finance Service is responsible for the bank operations and Customer Service for managing customers' information. The business logic is fulfilled by components that exist, side by side, in the same runtime instance, and the customer and account data is stored in the same database. As mentioned before, with this architecture, an operation such as a withdrawal, which needs customer and account data, can be performed in a single ACID transaction, isolated from other concurrent operations. Even if a direct credit rating adjustment is made, since it is also an ACID transaction, it either is executed before the withdraw was commited, or after the withdraw commited. A consistent view of the data is preserved until every operation is finished.

Let us now consider that the monolith was split into microservices, resulting in the architecture presented in Figure 1.2. In this system, there are two microservices, Customer Microservice and Finance Microservice, and each has its own database. The two microservices correspond to the modules described earlier but are now in different codebases and can be developed and deployed independently.

In this architecture, without any mechanism in place that provides data consistency, the system operations lose isolation. For example, while a withdrawal is being executed, the Finance Microservice

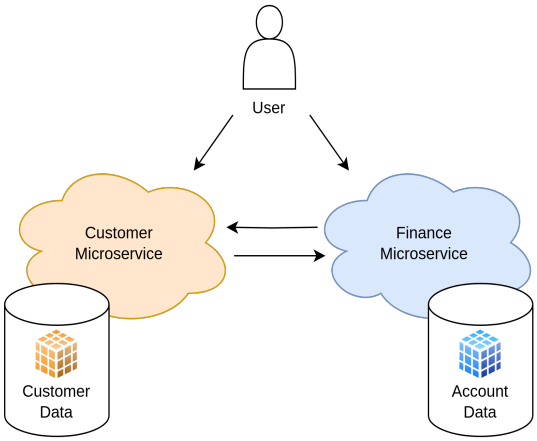

Figure 1.2: Bank system with a Microservice Architecture

communicates with the Customer Microservice to retrieve the credit rating of the respective customer's account and subsequently initiates the withdrawal. However, simultaneously, that same customer's credit rating can be modified by a concurrent operation, such as a reduction. After the withdrawal is complete, a dynamic credit rating adjustment is executed, for example, an increase of the credit rating. Since there are no guarantees that a consistent view of the data is preserved until the operation is finished, the direct credit rating reduction can occur between the withdrawal operation reading the initial credit rating, and the subsequent increase. Due to the interleaving of these two operations, the credit rating increase can overwrite the reduction, potentially creating a database state that does not correspond to either of the results of the two sequential executions of both operations, as if they were isolated.

These data inconsistencies problems, or anomalies, are especially important for applications in fields such as health or finance, where business-critical decisions rely on data consistency between executions of business operations. This problem presents significant challenges for software developers during the implementation of these systems, and they need to take these phenomena into account and create coordination mechanisms across the systems' services to avoid or compensate for these anomalies.

1.2 Goals

The goal of this thesis is to explore the challenges posed by serialization anomalies in microservice-based systems and design an algorithm that utilizes static analysis techniques to detect such anomalies. This includes understanding the impact of concurrent database operations, identifying the potential risks to data consistency and integrity, and analyzing existing approaches for detecting and mitigating serialization anomalies, identifying their strengths and limitations, and proposing an alternative with a different approach.

Another key objective of this research is the implementation of the previously designed algorithm, allowing for comprehensive testing and evaluation. By developing a practical solution, the thesis aims to showcase the algorithm's effectiveness and motivate its real-world usage. This implementation will serve as a tangible example of the algorithm's application and facilitate the understanding of its functionality and benefits. Furthermore, by demonstrating the practicality of the algorithm, developers can be guided

in taking appropriate countermeasures and preventive actions against serialization anomalies in their microservice-based systems.

1.3 Contributions

This thesis presents a novel algorithm for detecting potential serialization anomalies in database operations within microservice-based systems in a early stage of development. By leveraging static analysis techniques, the algorithm operates on a high-level description of system endpoints and database operations, offering a unique approach to identifying potential issues, without the need for a runtime environment. This allows for early detection of possible serialization anomalies across complex and distributed systems, addressing scalability challenges associated with large-scale microservice architectures.

Alongside the algorithm, an actual implementation was also developed, providing a real-world usage that complements the algorithm. The implementation showcases a concrete input and output, demonstrating the practical application and effectiveness of the algorithm in a real-world scenario. Additionally, the implementation provides an idea for a usable solution for software engineers to integrate into their development processes, aiding in the prevention of concurrency-related problems that could affect data integrity and consistency.

1.4 Outline

The following chapters of this thesis are structured as follows:

- **Chapter 2: Background** - This chapter presents the foundational concepts necessary for understanding the work, including monolithic versus microservices architectures, transactions, data consistency, and the types of anomalies that can arise in distributed systems.

- **Chapter 3: Related Work** - This chapter reviews the literature on related topics, focusing on migration tools for transitioning from monolithic to microservices architectures and existing techniques for detecting consistency problems.

- **Chapter 4: Approach** - This chapter details the methodology developed in this work, including the design and implementation of the STRIVE algorithm. It explains how the system is modeled using the DSL and describes the graph-based approach to detecting serialization anomalies.

- **Chapter 5: Modeling Systems for Static Analysis** - This chapter discusses the design of the Domain-Specific Language (DSL) used to model systems for static analysis. It covers the grammar and structure of the DSL and addresses the challenges of modeling systems at an appropriate level of abstraction.

- **Chapter 6: Anomaly Detection** - This chapter explains the core algorithm for detecting anomalies, including the graph construction process and cycle detection techniques. It also discusses how detected cycles are analyzed to identify potential serialization anomalies.

- **Chapter 7: Case Studies and Evaluation** - This chapter presents case studies that demonstrate the practical application of the proposed approach. It includes a comparison with other tools and techniques.
- **Chapter 8: Future Work** - This chapter outlines potential directions for future research.
- **Chapter 9: Conclusion** - The final chapter summarizes the contributions of this work. It also highlights the limitations of the current approach and suggests areas for further exploration.

Chapter 2

Background

This section introduces concepts of the monolith and microservices architectures, transactions, and data consistency. It also provides insights into the anomalies that can be present in distributed transaction and presents a brief characterization of two types of program analysis techniques for the detection of errors. The terminology used throughout the rest of the document is also presented.

2.1 Monoliths vs Microservices

When it comes to enterprise applications, two widely-adopted architectural styles are monoliths and microservices [10]. Even if built in a modular way, applications with a monolithic architecture are structured as a single deployable artifact and typically use a single relational database. This style of organization quickly provides a simple and safe way to execute concurrent business transactions since the code is called locally, and database accesses can benefit from ACID properties, which will be described in more detail in the next section. This architectural style simplifies development and maintenance but makes it very difficult to update the system when the business needs to change, creating a need to shift to a different architecture.

Microservices architecture, on the other hand, is a style for developing flexible software applications as it enables a more independent continuous delivery and deployment. Applications with this style are structured as a collection of loosely coupled services that can be independently developed, updated, deployed and scaled. Each microservice is an independent piece of software that performs a specific function, typically aligned with a business capability and its own data. A microservice can communicate with another synchronously, calling an API that the other service exposes, using lightweight protocols such as HTTP and gRPC, or asynchronously with events or message passing[1]. This approach simplifies system scalability by allowing new services to be added or existing ones to be expanded with minimal disruption to the overall architecture.

By breaking a large application up into a collection of small services, the result is a system with a more complex design. Services must use interprocess communication mechanisms, instead of just doing method calls, and be designed to deal with the unavailability of the other services. As discussed before, implementing business transactions that span multiple services requires creating coordination mechanisms across the systems' services to avoid or compensate for data inconsistency anomalies.

[1] learn.microsoft.com/en-us/azure/architecture/microservices/design/interservice-communication

Migrating from monolithic to microservices requires careful planning and consideration of the advantages and disadvantages of each approach. Migration strategies should consider the costs and risks involved in migrating existing applications, the feasibility of retaining and maintaining existing functionality, as well as taking transaction problems that can appear after the migration into account.

2.2 Transactions

A database's contents can be accessed and modified as part of a single logical unit of work known as a transaction. Read and write operations are used by transactions to access and manipulate data. In order to preserve data consistency, transactions are required to meet specific properties. These are known as ACID properties, with ACID standing for atomicity, consistency, isolation and durability [24].

- Atomicity, also known as the "all or nothing" rule, says that a transaction either fully executes, meaning all steps are executed, or fails, meaning none of the steps are.
- Consistency refers to the characteristic that transactions cannot violate integrity constraints created in the database. For instance, a defined constraint might be that all bank accounts have a positive balance, and due to this property, any transaction that would update the balance to a negative value would be canceled.
- Isolation specifies that transactions cannot see the effects of other concurrent transactions that did not finish yet.
- Durability states that all committed changes from transactions will remain saved in the system even in the case of a crash or failure.

The DBMS (Database Management System) is responsible for ensuring that transactions meet these properties, so a developer does not need to worry about implementing them. In a microservice system, however, each individual service can benefit from ACID transactions locally, but the problem arises in operations that manipulate data from other services' databases. For instance, if a business transaction spans multiple microservices and one of them fails, then all other microservices have to act accordingly and also fail or compensate for that failure from their point of view, otherwise the Atomicity of the global operation is lost. The same problem can occur with the Isolation property, if a microservice starts an operation and asks another microservice for information, that information can already be outdated since the other microservice is always available for requests, resulting in the first microservice seeing the effects of another concurrent transaction after its own transaction started.

As mentioned before, microservice-based systems can also be "ACIDified", with solutions that support distributed transactions like the two-phase commit protocol [2]. As discussed by Nitin et al. [19], these solutions are not a viable option in general, unless we can rely on expensive data centers and meticulously synchronized atomic clocks.

If a business transaction spans multiple services, a more common and efficient solution is to use Sagas to implement the operation and the steps required to maintain data consistency across the services. The

[2] developers.redhat.com/blog/2018/10/01/patterns-for-distributed-transactions-within-a-microservices-architecture

saga pattern only grants the ACD properties to global business transactions, lacking isolation, typically resulting in the countermeasures needed to be implemented at the application level.

2.3 Data Consistency

In monoliths with a single database, data consistency reduces to database consistency, i.e., the degree to which data stored in the database is accurate and reliable. In the microservice architecture, it is the degree to which data held by a given microservice is consistent with other data maintained by the other microservices. There are several different consistency models, contracts that define the degree of consistency. For the purpose of this work, the more relevant models are those prevalent in monoliths and microservices: strong consistency and eventual (weak) consistency.

Data can become inconsistent due to poor design choices, or the propagation of inaccurate data. In an enterprise environment, it may be perfectly normal for inaccurate data to exist in the system due to the nature of that data being non-critical, or the performance benefits outweigh the cost of maintaining consistency. On the other hand, some data is critical for the business and must be maintained consistently.

Different mechanisms can be used to ensure that data kept by a system remains consistent over time, each offering advantages and disadvantages depending on the type of data to be stored and the application requirements [23].

2.3.1 Strong consistency

This model means that the user never sees an inconsistent state of the system. A monolith supports this easily with ACID transactions since all the data related to a specific transaction is colocated, and concurrent transactions do not see changes of each other, resulting in coherent database states. For microservices, strong consistency means that every service should only expose consistent data to the user and every other service. To implement this model in this architecture, it is necessary to use mechanisms that prevent any data modification while a business transaction is happening globally across the system. While this is possible with coordination and global locks, it drastically reduces the availability and performance of the system.

An alternative approach to achieve strong consistency, while mitigating the performance impact, is the use of optimistic locks. In this approach, transactions proceed optimistically without acquiring locks on data. Before committing, a transaction checks whether the data it has modified has been concurrently modified by other transactions. If conflicts are detected, the transaction can be rolled back or merged with the conflicting changes, ensuring that data remains consistent. Optimistic locking minimizes the contention for locks, allowing transactions to execute in parallel more frequently, but it requires careful conflict resolution strategies, and consumes additional resources with possible useless computations.

2.3.2 Eventual consistency

This model offers an alternative to the strong consistency model and is often favored in microservices and other scenarios where strong consistency might lead to poor system performance, or when high system availability is a priority. Eventual consistency relaxes the requirement of immediate consistency across all parts of the system.

In an eventual consistency model, the primary guarantee is that, given enough time and absence of new updates, all replicas of data within the system will eventually converge to a consistent state. This means that, for a given transaction committed by one service, it may take some time for all other services to see and reflect that change. This delay in achieving consistency is known as ”eventual.”

Eventual consistency allows individual transactions to complete even if other transactions have modified the same data items concurrently. However, this introduces the possibility of temporary inconsistencies or anomalies in the system. If one service relies on data from another service but reads it before the eventual propagation of the latest updates, it might operate on outdated or inaccurate information, potentially leading to data consistency anomalies.

These anomalies are discussed in more detail in Section 2.4. Eventual consistency is favored when high system availability and performance are paramount, and the system can gracefully handle temporary inconsistencies in exchange for these benefits. Different strategies like conflict resolution mechanisms and techniques to handle these anomalies are often employed to mitigate their impact.

2.4 Anomalies

Weakening the isolation property of the ACID principle implies that database inconsistencies might arise when multiple transactions work on the same data concurrently. If an inconsistent state is achieved, a serializability anomaly has occurred. Serializability is a property that ensures that the execution of multiple transactions is equivalent to one of the possible executions of multiple transactions sequentially, one at a time. It is vital to guarantee and maintain the consistency and integrity of the data. Various types of serializability anomalies can occur depending on the type of operations and the order in which they are executed [22]:

- **Dirty read**: A microservice reads data that is being updated by another microservice, and the data is not yet in a consistent state.
- **Dirty write**: A microservice writes data that is being read by another microservice, and the data is not yet in a consistent state.
- **Non-repeatable read**: A microservice reads the same row of data multiple times and gets different values because another microservice updated that row.
- **Phantom read**: A microservice reads a set of rows, but the set changes when another microservice inserts or deletes rows, causing the first microservice to see different results on subsequent reads.
- **Lost update**: Two microservices incrementally update the same data simultaneously and one update is lost or overwritten.
- **Write skew**: Two microservices update different pieces of related data simultaneously, leading to an inconsistency between the two pieces of data.
- **Short/Long fork**: A change is made to a service that requires a few/large (short or long) numbers of other services to also be changed in order to maintain compatibility. This problem is more related to service coupling and interdependencies rather than immediate data inconsistencies, which can occur if services are tightly coupled.

All these anomalies can cause the microservices to make decisions based on incomplete or inconsistent data. In summary, serializability anomalies originate from concurrency side effects when the isolation property is lost. Business operations that only rely on one microservice to be fulfilled are less likely to suffer from anomalies, i.e., if each microservice has its own database and tables, it is less likely that a dirty read or lost update anomaly will occur because each microservice is interacting with its own data, plus each individual component can also support ACID properties. In cases where services need to access or update each other's data, such as the examples described above, the anomalies have a higher probability of occurring due to isolation not being present in distributed transactions. Each system is unique, with its specific requirements and characteristics, meaning that they can have their own instances of these serializability anomalies. Fortunately, there are known techniques to prevent them, once we are aware of them. To conclude, more efforts in identifying these anomalies, especially during a migration process, will result in safer and more correct software.

2.5 Software Analysis for Detection of Defects

Since the serializability of business transactions is no longer ensured, there are some execution scenarios in which a given decomposition of the monolithic application might not behave as intended.

Static and dynamic analysis are two different but complementary ways of inspecting software to verify the absence of defects. Both provide valuable information that can be used to improve the quality and reliability of software applications, and they complement each other by revealing different aspects of the software's functionality [2]. The differences between the two types of analysis are discussed in more detail below.

2.5.1 Static Analysis

Static analysis uses formal mathematical and logical rules to search for errors in software without executing it. A popular static analysis technique is to simulate the program execution with abstract values (instead of the concrete runtime values in a real execution). This execution spans all possible program states and allows for the variables to be controlled and decoupled from runtime values, thus not limiting the analysis only to predictable and common program states. It can be used to identify any errors or bugs in the software in the development stages. Another advantage of this kind of analysis is that it can be used on a simpler model of the software instead of the actual code, which is typically done by having an abstract representation of the artifact being analyzed, reducing the complexity of the analysis, but in turn, the bugs detected in these models are not directly mapped to the original artifact.

2.5.2 Dynamic Analysis

Dynamic analysis is an execution technique that may be used to verify, diagnose, optimize or control the behavior of software and computer systems. It involves analyzing how the code is executing during run time, as opposed to how the code was written, to identify problems and inefficiencies. Using this kind of analysis, the software code can be tested against unexpected situations, and can help pinpoint bugs with genuine and precise counter-examples. However, dynamic analysis cannot be used to find all bugs due

to the time-consuming process of executing all possible combinations of runs and should be used as a complementary tool to static analysis.

This chapter discussed foundational concepts critical to understanding the complexities of microservices architectures. It contrasts monolithic and microservices architectures, explains transaction management, and details data consistency models like strong and eventual consistency. The chapter also introduces anomalies that can arise in distributed systems, such as dirty reads and lost updates, and briefly touches on software analysis techniques for detecting these issues. This sets the stage for the upcoming sections, where these concepts will be further explored and applied, particularly in the context of detecting and mitigating transaction consistency problems in microservices systems.

Chapter 3

Related Work

In this section, I describe work related to the problem of assisting developers in migrating to microservices, including both industrial applications and theoretical techniques to detect consistency problems.

3.1 Migration Assisting Tools

The necessity of migrating monoliths to microservices lead the industry to refine and streamline this process. The end objective is to have a decomposition with loosely connected and specialized microservices. However, tackling this process manually is difficult, time-consuming and error-prone. The migration requires complex planning, engineering time and resources, making complete rebuilds rarely feasible [19]. This has motivated the development of several approaches and tools to help developers find a good partitioning of the monolith into several microservices, and establish the business functionalities boundaries [3, 6, 13, 15, 19]. An example of one of these tools is IBM's Mono2Micro [15], which targets JavaEE monoliths and suggests a set of partitions of application classes that are functionally cohesive.

Automatic approaches to monolith decomposition typically rely on the computation of module dependencies and on the application of clustering or evolutionary algorithms over these dependencies to create module partitions that have certain "good" properties (e.g., high functional cohesion and low coupling). As mentioned by Nitin et al. [19], one of the limitations of current approaches is that the recommended partitions typically do not take into account the loss of isolation and the lack of consistency in the data that can arise after the migration.

3.2 Detection of consistency problems

Previous work on the detection of consistency problems has focused on developing static analyses to check for serializability of programs under weaker isolation levels in centralized databases [9, 14]. More recently, there has also been some work on static analysis to check for serializability of client applications of weakly consistent replicated distributed databases [4, 18, 20].

For instance, Rahmani et al. [20] addressed this problem by developing CLOTHO, a parametric tool based on the specification of the consistency model offered by the database. CLOTHO generates abstract executions of database transactions programmed by the user, and detects serialization anomalies by checking for cyclic dependencies between read and write events in those transactions, reporting them back with test inputs to reproduce them. The tool does this by constructing a conjunction in First-order

logic (FOL), as shown in Figure 3.1, composed of different clauses, such as the context axioms, the consistency model guarantees, the dependencies between the read and writes events, and the anomalies themselves.

$$\varphi \equiv \varphi_{\text{CONTEXT}} \wedge \varphi_{\text{DB}} \wedge \varphi_{\text{DEP}\rightarrow} \wedge \varphi_{\rightarrow\text{DEP}} \wedge \varphi_{\text{ANOMALY}}$$

Figure 3.1: CLOTHO conjunction, adapted from Rahmani et al. [20]

The satisfiability of the formula is then checked with Z3 [17]. This way, the problem of detecting a serializability anomaly is reduced to a SMT problem, with the goal of finding a satisfying assignment to the logic formula that results in a cycle being present.

Considering its benchmarks, the analysis technique implemented by CLOTHO seems promising for detecting transaction consistency problems in microservices decompositions of monolithic applications. However, CLOTHO is not directly applicable in these scenarios because microservices architectures are subject to different constraints and do not reduce to sets of operations executed on a single replica of a distributed database. Specifically, when CLOTHO analyzes an application's operations to detect cycles, it simulates concurrent executions of the same operation across different replicas. This approach does not align with microservice-based systems, where each operation is typically handled by the service responsible for that operation's specific business context.

There are also anomaly detection techniques that use dynamic analysis instead, for instance, they rely on the execution of the system to build dependency graphs to check for cycles or analyze logs after execution [5]. The tools mentioned in the study [5] approach a different but somehow related problem, that is, the detection of constraint consistency errors in distributed enterprise systems (for instance, different constraints are imposed on the same type of input in different components). These tools focus mainly on security issues and the detection of security anomalies, and allow anomaly detection on a system that is already deployed.

Another relevant dynamic analysis algorithm is presented in ACIDRain [25]. The paper introduces a novel method called Abstract Anomaly Detection (2AD) for detecting isolation anomalies in web applications. The 2AD approach utilizes dynamic traces of database accesses, real SQL logs, to analyze the space of possible concurrent interleavings and identify potential anomalies. The methodology applies weak isolation to reason about both API calls and concurrent executions, enabling the construction of an abstract history that represents the infinite space of concurrent schedules in a finite data structure. The method involves several steps (Figure 3.2): First, it generates traces of normal application behavior from the database, either from typical usage or explicitly for anomaly detection. Then, it lifts this trace to extract API call-generated transactions and their associated variables. Next, it detects potential anomalies in the extracted trace, and if an anomaly is found, the method refines the trace to pinpoint the specific sequence of API calls causing the problem. While ACIDRain detects real anomalies due to using real SQL logs as input, the system needs to be deployed, meaning it relies on an existing operational environment, which can limit its ability to identify issues during the early stages of development or before deployment. This dependence on live traces can make it less effective for proactive testing and may miss potential anomalies that could arise in different or unforeseen operational scenarios.

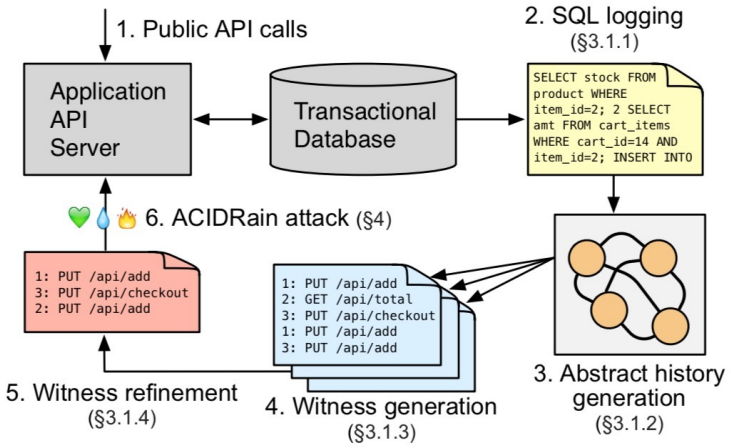

Figure 3.2: 2AD workflow to discover ACIDRain attacks (Figure 2 from [25])

This chapter provides an overview of existing tools and methodologies for detecting consistency problems in microservices architectures. CLOTHO and ACIDRain are highlighted as significant contributions in this domain. However, while CLOTHO is effective in detecting serializability anomalies in distributed databases, it is not directly applicable to microservices, where each service often has its own database and specific operations. CLOTHO's focus on managing transactions across distributed replicas does not address the complex, inter-service interactions and independent data management that are characteristic of microservices architectures. On the other hand, ACIDRain's dynamic analysis approach offers valuable insights by analyzing real execution traces but falls short in early detection of anomalies, as it requires the system to be already deployed. These gaps justify the need for this thesis, which aims to develop an early anomaly detection algorithm tailored for microservices.

Chapter 4

Approach

Given the susceptibility of microservice architectures to serialization anomalies in comparison to their monolithic counterparts, STRIVE[1] (**S**tatic **T**ransaction **R**ead-Write **I**nterleaving **V**erification **E**ngine) was developed to identify these anomalies within microservices systems. This chapter introduces STRIVE's pipeline, designed to assist system developers in addressing these challenges.

4.1 Overview

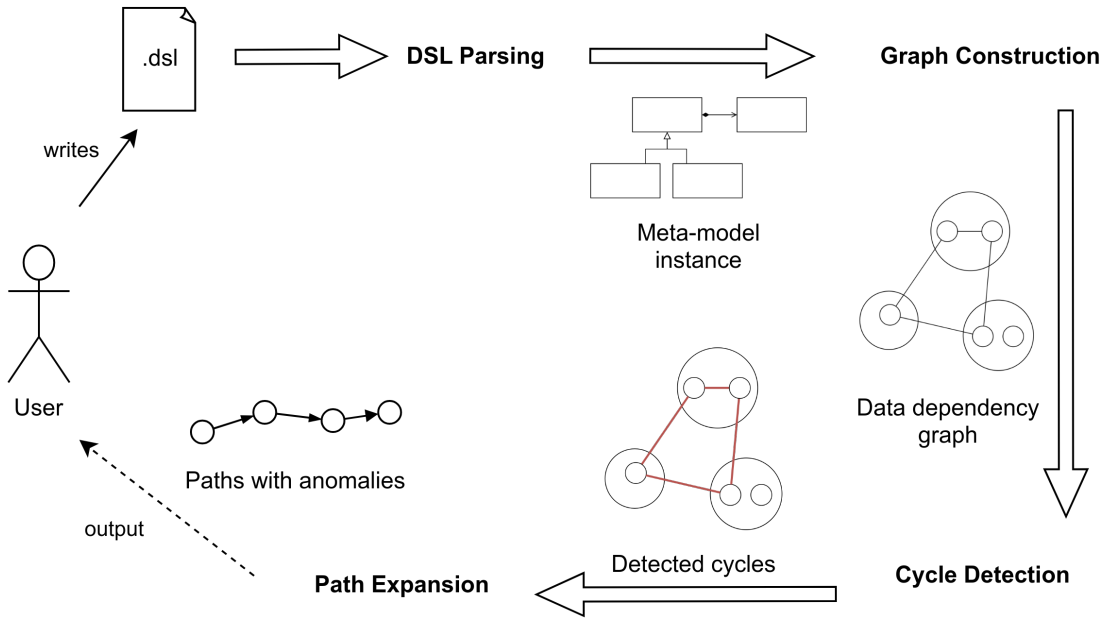

Figure 4.1: STRIVE's pipeline

In STRIVE, the identification of serialization anomalies within a microservice-based system requires that some information is extracted from the system. Concretely, users are tasked with describing the microservices within the system. This description should encompass details like available endpoints, the composition of endpoints in terms of database operations, the database tables these operations interact with, and which operations can be executed atomically. This description does not require a low-level

[1] https://github.com/andre-f-silva/strive

code representation of the system, a high-level textual description suffices. The approach is, hence, applicable if the system has not yet been implemented.

This textual description serves as the initial step in STRIVE's pipeline. A text-based Domain Specific Language (DSL) is proposed, which allows users to describe microservices, focusing primarily on the information needed to identify anomalies, bypassing low-level decisions such as specific programming languages and technologies.

Once all the necessary system characteristics are modeled, the next stage involves organizing this information for the anomaly detection algorithms. As detailed in subsequent sections, this approach combines ideas from CLOTHO and ACIDRain, leveraging graph theory. All described endpoints, database operations, and atomicity structures are transformed into a graph where database operations become nodes, grouped by their atomicity and endpoints, while edges represent data dependencies among these operations. This results in a data dependency graph representing the microservices of the system, forming the foundation for anomaly detection. The graph is then explored using specific rules, to identify anomalies, effectively reducing the anomaly detection problem to a cycle detection problem.

To provide valuable feedback to the end-user, the detected cycles need to be translated to anomalies. This will be done by converting each cycle into a reproducible sequence of database operations, to effectively present to the end-user scenarios in which they need to implement mechanisms to avoid the detected anomalies. This will be achieved through dependency DAG (Directed Acyclic Graph) creation and topological sort algorithms.

The following sections will provide an in-depth explanation of methodology, algorithms, and implementation details of this approach, with the aim of providing a comprehensive understanding of this approach's effectiveness and potential applications in real-world scenarios.

4.2 Running Example

In this section, we revisit the bank example used in Chapter 1. An anomaly was detected in the microservice version of the system, where the concurrent executions of a withdraw operation and a credit rating modification operation created an incorrect system state. Figure 4.2 describes the two operations in more detail. The figure shows 2 microservices, Finance and Customer. The Finance Microservice exposes a withdraw endpoint. An execution of withdraw has 3 steps: (1) the read customer's credit rating, that is a remote call to the Customer Microservice, (2) update the customer's balance, which is a database transaction to the microservice's respective database table, and finally, (3) a remote call to update the customer's credit rating. The Customer Microservice's exposed endpoints are described in a similar way, providing entry points to read and modify a customer's credit rating.

This example will be constructed and verified with STRIVE to explore the different stages of the pipeline and reach the detected anomaly.

4.3 Describing the System

To feed STRIVE with the necessary information to detect anomalies, the user needs to provide a description of the system under study, much like what Figure 4.2 portraits. To create a standardized way

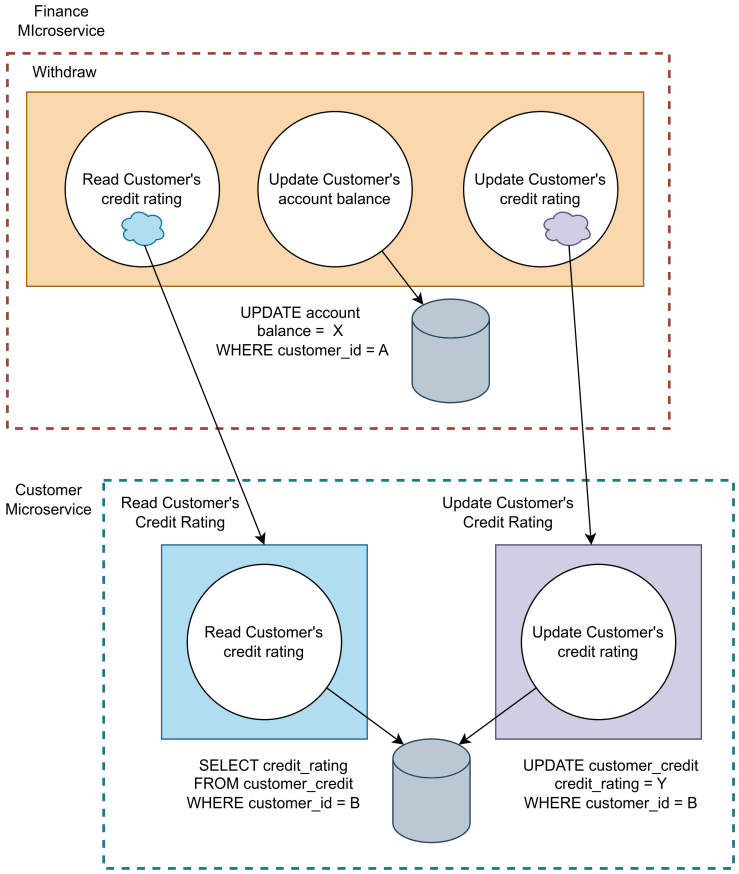

Figure 4.2: Running Example: Bank operations

of describing a system, a DSL was designed. The DSL's grammar guides the user to delineate the microservices, indicating the available endpoints, detailing database operations executed in each endpoint, and data flow. STRIVE will then convert the text description into an internal model in order to proceed to the next stages.

4.3.1 DSL

The Finance Microservice represented in Figure 4.2 can be described as follows in the designed DSL:

```
System {
    Microservice Finance {
        Table account {
            columns: [id, customer_id, balance]
        }
        Operation updateBalance {
            type: write
            table: account
            input: [customer_id, updated_balance]
            output: []
            affected_columns:[balance]
            predicate: account.id == customer_id
        }
        Endpoint withdraw {
            input : [customer_id, amount]
            output : [updatedbalance]
```

```
            transactions : {
                [remote getCustomerCreditRating(customer_id)]
                [updateBalance(customer_id, amount)]
                [remote updateCustomerCreditRating(customer_id, credit_rating)]
            }
        }
    }
}
```

The description starts by describing the Table account with the columns id, customer_id, balance.

Then it describes a database operation, updateBalance that is a write operation targeting the account table. It receives as input an customer_id and updated_balance and no output is desired. It's a write specifically affecting the balance column and it uses account.customer_id == customer_id as a filter, which is similar to a SQL query WHERE clause, saying that this operation affects lines in the account table that have a customer_id equal to the input customer_id.

Lastly, there is the definition of the endpoint withdraw. This endpoint receives as input a customer_id and amount and outputs the updatedBalance. Then the transaction sequence of the endpoint is defined: the endpoint first does a remote call, to the getCustomerCreditRating endpoint passing the customer_id as input, performs the withdraw, with updateBalance, and calls another endpoint, updateCustomerCreditRating, with the respective input also. The remaining parts of the system are described in a similar manner, namely the Customer Microservice, that owns the two endpoints that are remotely called by the withdraw endpoint.

The square brackets surrounding the different transactions mean an atomic block, indicating that all items within each square bracket block, including database operations or remote calls, are executed atomically, preventing the exposure of intermediate states. STRIVE is designed to identify anomalies in systems with distributed business transactions that lack protective mechanisms like the two-phase commit. As a result, remote transactions should be individually encapsulated within square brackets. This signifies that, during an endpoint's execution, a remote call reads the visible state of the target microservice and the obtained information can be used in the later operations of the endpoint without concerns. If there were multiple local database operations, they could be included in a single atomic transaction by enclosing them in square brackets, since STRIVE assumes that the local databases are under ACID conditions. This means that the visible states of the corresponding microservice's database are limited to the states before and after the operations enclosed with square brackets.

4.3.2 Internal Representation

The first step in the pipeline is the parsing of the DSL input into an internal representation. Although the parser was not developed in this thesis, its purpose is to convert the text into a meta-model instance to be manipulated for the next algorithms.

The meta-model instance corresponding to the DSL is presented in Figure 4.3 and It represents an organized structure ready for manipulation. The example illustrates that an Endpoint (or a BusinessTransaction), is composed of one or more TransactionUnit. A TransactionUnit can either be a RemoteTransaction, or a LocalTransaction. A RemoteTransaction represents a call to another Endpoint, and a LocalTransaction is a group of database operations that are executed atomically in the microservice's local database. So the three transactions items in the DSL:

```
transactions : {
    [remote getCustomerCreditRating(customer_id)]
    [updateBalance(customer_id, amount)]
    [remote updateCustomerCreditRating(customer_id)]
}
```

represent in sequence, a RemoteTransaction to the getCustomerCreditRating endpoint (which would be detailed in the DSL and the meta-model), a LocalTransaction composed of only one atomic database operation, updateBalance, and another RemoteTransaction, to the UpdateCustomerCreditRating endpoint.

In Figure 4.3 it is also possible to see the data flow through the variables. The withdraw endpoint takes in two input parameters, customer_id and amount. These parameters are then passed along through various TransactionUnits. These parameters create connections between the data in different TransactionUnits, establishing data dependencies.

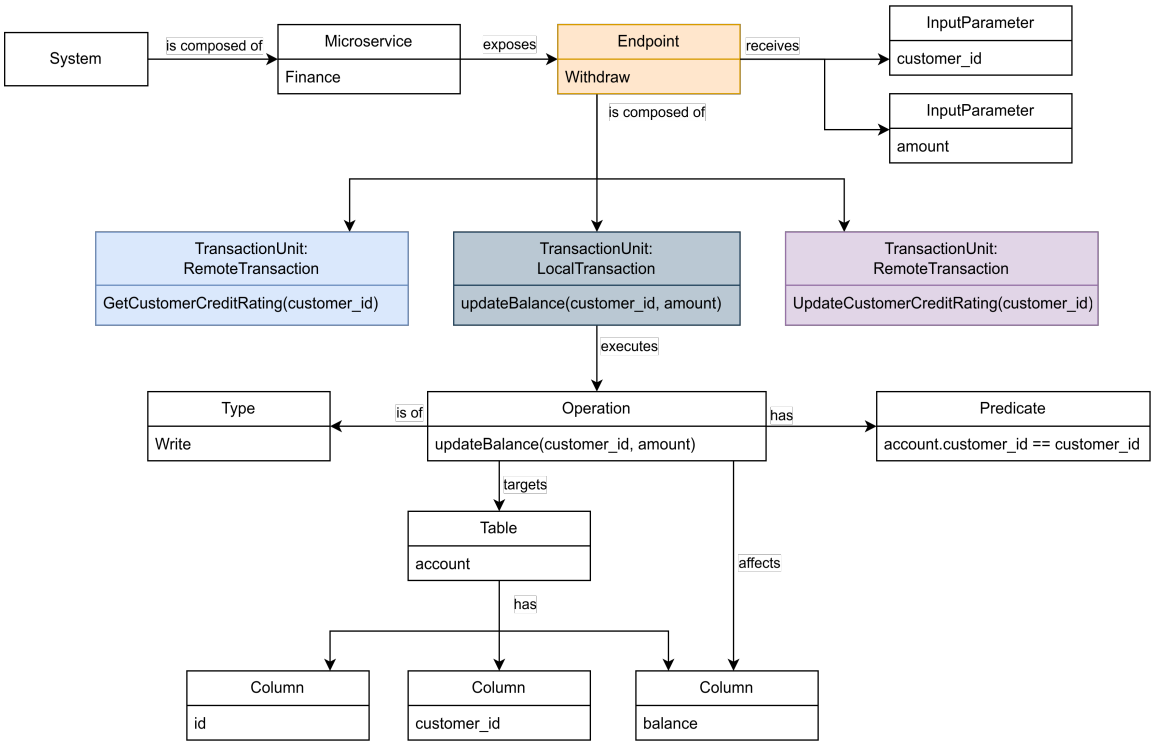

Figure 4.3: Meta-Model Instance for Finance Microservice

4.4 Anomaly Detection

After treating the user's input, it can now be leveraged to start the anomaly detection stages. As discussed in Chapter 3 ACIDRain's anomaly detection core algorithm is based on creating a graph to capture the dependencies and conflicts between different endpoint calls and database transactions. STRIVE builds upon ACIDRain's foundational concepts, with modifications tailored to fit the microservices architecture. STRIVE also draws upon inspiration from CLOTHO's SMT solver usage to detect anomalies, to verify

problematic dependencies between operations. The following sections will dive into STRIVE's anomaly detection algorithm.

4.4.1 Data dependency graph construction

The first step of creating the data dependency graph is defining what will be the nodes. In this case, as noted by the importance of the detailed specification of the database operations, such as the target table columns, input, output, and the atomic groups, the database operations are important building blocks of the anomaly detection, thus being the base nodes of the graph. Figure 4.4 shows the base nodes of the data dependency graph of the example presented in Figure 4.2.

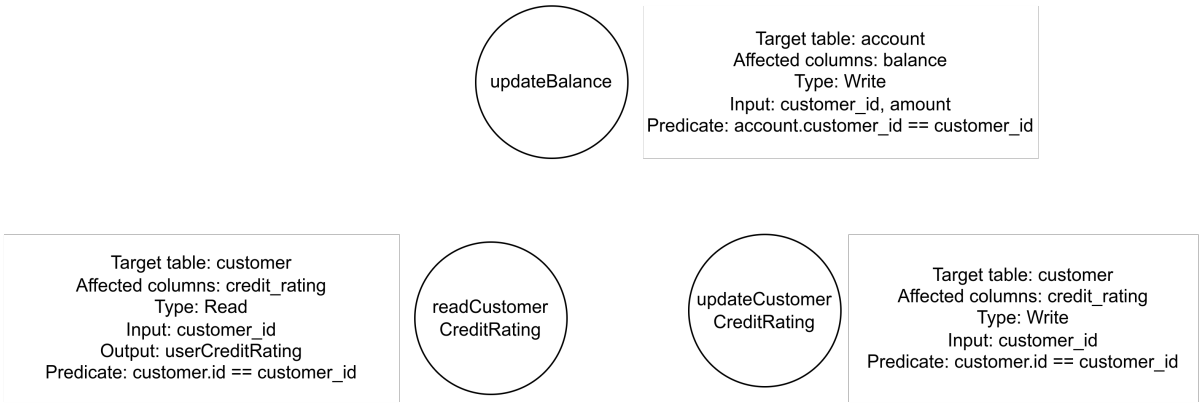

Figure 4.4: The base nodes of the data dependency graph for the running example

Edges between these nodes are then established when there are read/write conflicts or data variables dependencies. There is a read/write conflict when at least one of the operations is a write (the anomalies referenced in Section 2.4 do not occur without a write), and there are database table columns in common between the pairs of the nodes' operations. Another way to establish edges are the dependencies between data variables. This is done by checking if there are intersections between the input and output variables of the operations. In either case, the predicates of the operations also need be both verified. The algorithm that establishes the edges between the nodes is explained in greater detail in Section 6.1.4.

In the case of the running example, a single edge is established between the 3 base nodes, as illustrated in Figure 4.5. The edge represents a read/write conflict since the two operations interact with the same table (customer) and column (credit_rating). Then a logical conjunction is created between both of the operactions' predicates: customer_credit.customer_id == customer_id_read $\wedge$ customer_credit.customer_id == customer_id_update. This logical clause signifies that an edge is established between the two operations under the circumstance that the customer_id used as input is the same in both operations. If the readCustomerCreditRating were to affect user A, and updateCustomerCreditRating were to affect user B, there would be no conflict, thus no edge.

As hinted before, the interleavings between commited database states of the different microservices are the root problem of anomalies. These committed states are the result of atomic blocks of database operations corresponding to the LocalTransactions (see Section 4.3.2). To reflect this in the graph, the database operations nodes in the same atomic block are grouped into LocalTransaction nodes. In the

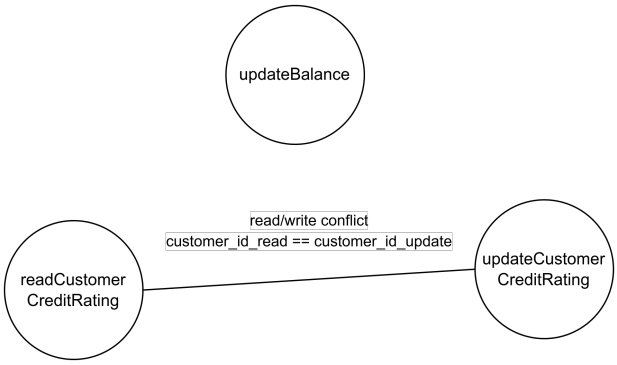

Figure 4.5: Edges of the data dependency graph for the running example

running example, this is represented in Figure 4.6.

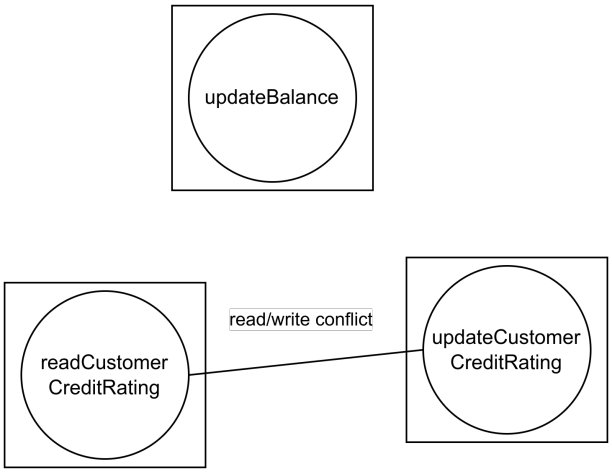

Figure 4.6: LocalTransaction nodes around Database Operations nodes, in the data dependency graph for the running example

Furthermore, the problematic interleavings happen when conflicting operations of other endpoints happen in between operations of one endpoint. So, if the interleavings are going to be detected through the LocalTransactions nodes, these need to be grouped inside the corresponding Endpoint node to maintain the endpoints contexts present. The result of doing this in the running example is illustrated in Figure 4.7.

At this point, the database operation and LocalTransaction nodes and edges are established, but the remote endpoint calls in the Withdraw endpoint are still not modeled has hinted by the cloud symbols in Figure 4.7. The way this is done, is that every LocalTransaction node in the endpoint that is being called remotely, is copied into the endpoint caller. This is represented in Figure 4.8.

Details about the remote call inlining and other ideas used in graph construction stage are explained in Section 6.1.

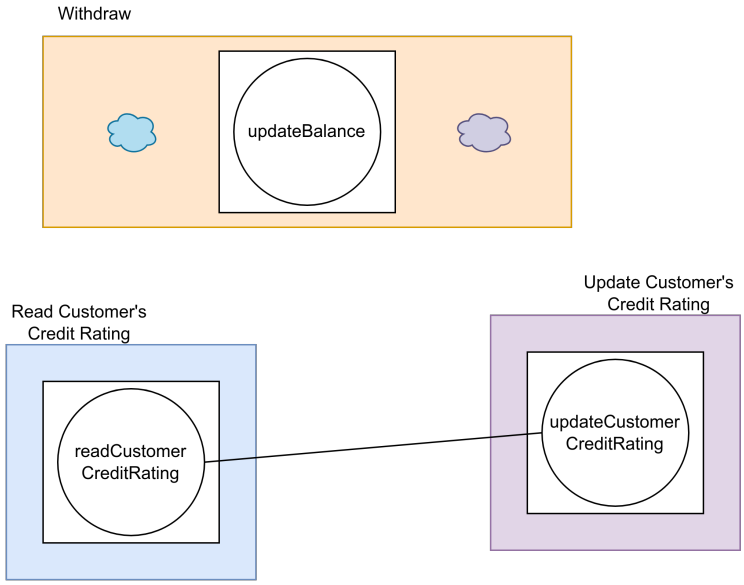

Figure 4.7: Endpoint nodes around LocalTransaction nodes, in the data dependency graph for the running example

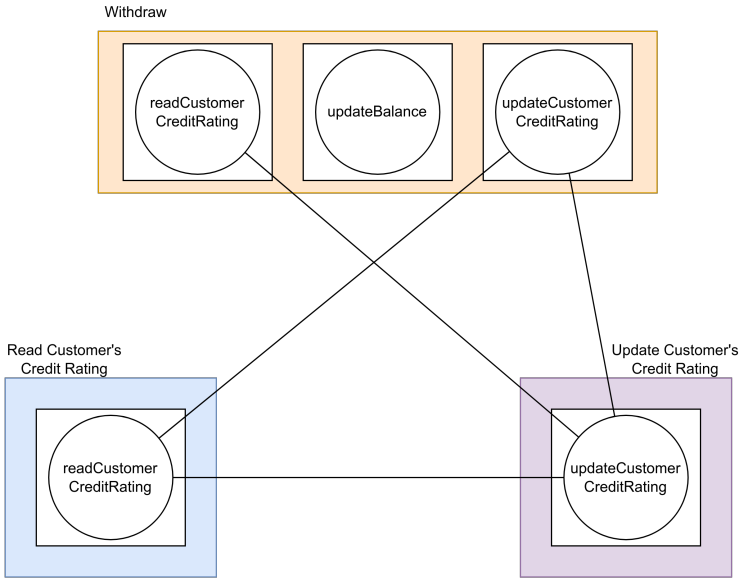

Figure 4.8: Remote Call Inlining of the example in Figure 4.7

4.4.2 Cycle Detection

Once the data dependency graph is built, STRIVE can now proceed to the actual anomaly detection. The objective is to identify problematic interleavings in an endpoint's execution. The approach involves traversing the graph from an endpoint node, navigating through conflict edges, and determining if it's possible to return to the same endpoint node. This effectively reduces the anomaly detection problem to a cycle detection problem.

STRIVE runs a modified depth-first search to discover these problematic cycles, which is based on

the navigation through LocalTransaction nodes, each corresponding to visible system states. Additionally, for a cycle to represent a problematic interleaving, it must contain at least two different visible system states of the same Endpoint node.

An example of a cycle in the graph of Figure 4.8 is readCustomerCreditRating → updateCustomerCreditRating → updateCustomerCreditRating (see Figure 4.9). This cycle represents the anomaly discussed in Chapter 1: A call to updateCreditRating between the Withdraw read and update credit rating.

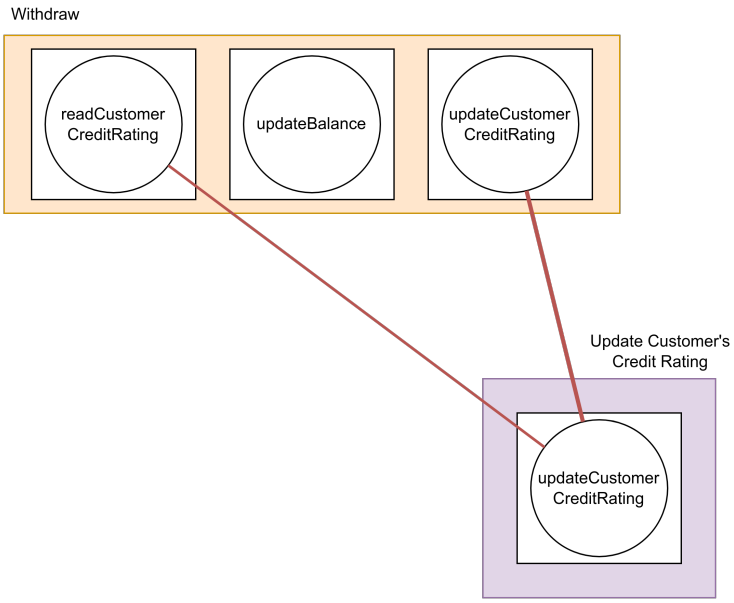

Figure 4.9: Withdraw and UpdateCreditRating cycle

Note that the monolith version of the same system originates zero cycles due to database operations all being atomic, note the LocalTransaction spanning all database operations in Figure 4.10. Since in this case, both endpoints interact with the same monolithic database and no remote calls are needed, so no interleaving opportunities are created, thus no cycle can occur.

4.4.3 Cycle Walks

While the detected cycles point out potential problematic interleavings, they do not necessarily capture all database operations associated with the involved endpoints. For example, consider again the cycle in Figure 4.9: readCustomerCreditRating → updateCustomerCreditRating → updateCustomerCreditRating, In this sequence, the updateBalance operation is excluded as it played no role in triggering the detected anomaly and was deemed non-problematic. For a more detailed understanding, especially in endpoints with multiple operations, it is beneficial to present a step-by-step sequence of every database operation. This aids in pinpointing how the anomaly within the cycle can be reproduced, for instance, the direct call to the UpdateCustomerCreditRating endpoint can occur either before or after the updateBalance, resulting in the anomaly in both cases. Extracting such information from the cycle provides valuable insights for further analysis.

To compute all sequences of database operations that trigger the anomaly within the identified cycle, the dependencies between the nodes of the cycle need to be modeled. This modeling is achieved

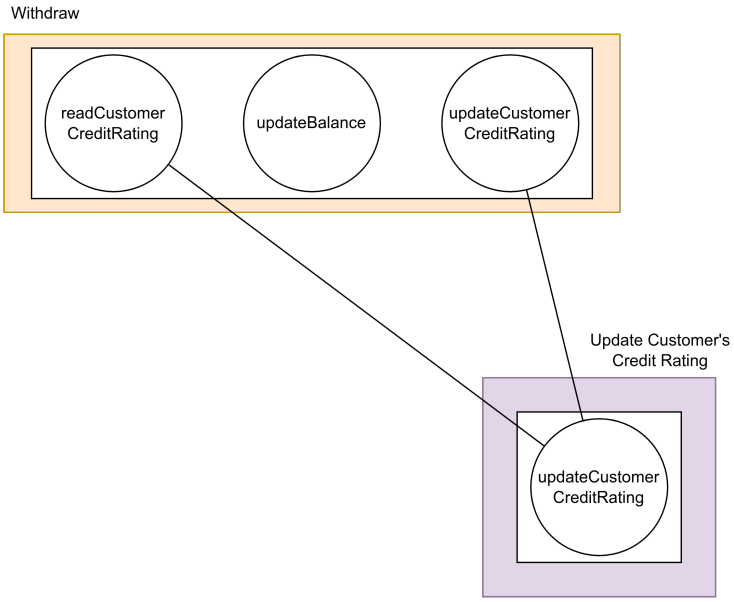

Figure 4.10: Monolith Withdraw and UpdateCreditRating graph

by generating a dependency DAG based on the cycle. The idea behind this DAG is that its edges represent dependencies of precedence between operations. These dependencies encompass both the order of execution dictated by the endpoint flow and the relationships with nodes from other endpoints that contribute to triggering the anomaly. Figure 4.11 shows the DAG produced for the cycle in Figure 4.9. The black edges denote the sequential order in which the LocalTransaction nodes are executed within the Withdraw endpoint, reflecting the inherent flow of the endpoint. On the other hand, the red edges represent the order that must exist for the anomaly to manifest.

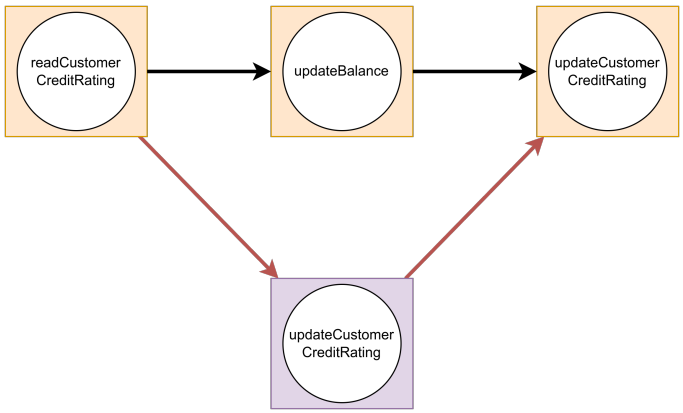

Figure 4.11: Cycle dependency DAG

Valid sequences of LocalTransactions that trigger the anomaly are those that adhere to the established DAG dependencies. To extract these sequences, the DAG is topologically sorted. In the context of the DAG in Figure 4.11, two topological paths are generated, as depicted in Figure 4.12. These paths serve as valuable guides for users seeking to reproduce the anomaly, offering a clear understanding of the

sequential orderings necessary for the anomaly to occur.

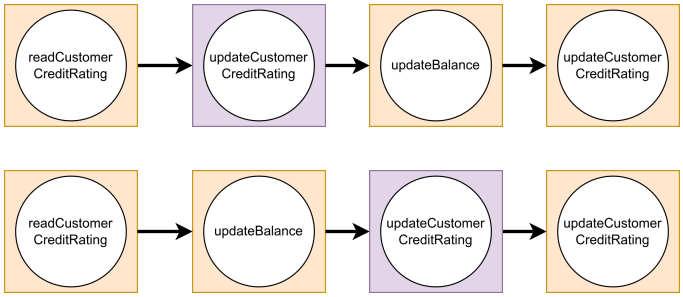

Figure 4.12: Topological sort paths of the DAG

4.5 Output Analysis

With the example paths generated by STRIVE, users gain insights into identified cycles and paths that trigger anomalies within the system. Equipped with this information and their own understanding of the system's intricacies, users can make informed decisions to proactively avoid potential anomalies. An example of an interpretation of the anomaly represented in the cycle in Figure 4.11 could be: "If the Withdraw endpoint and the UpdateCustomerCreditRating endpoints are called at the same time, both targeting the same customer, a consistency problem can occur and an erroneous behavior will be provoked. A potential solution could be to guarantee that in the system's environment, these endpoints can never be called in parallel when referencing the same customer."

As the anomaly detection problem was simplified into a cycle detection problem, the corresponding anomaly prevention strategy revolves around eliminating these cycles. This elimination is achieved by removing edges, which translate to introducing concurrency protection mechanisms into the system, like the previous solution discussed.

Nevertheless, users must carefully analyze the cycles, verifying whether they genuinely correspond to anomalies within the visualized system. As the detection relies on a simplified text-based description of the system through the DSL, certain safeguards against concurrency issues, such as conditional executions or domain invariants, may not be fully captured, potentially leading to false positives. For instance, the previous solution can already be considered or implemented in the system, but was not captured in STRIVE's input.

Acknowledging the possibility of false positives, STRIVE empowers users with the flexibility to make manual adjustments. This includes options like pruning specific edges to refine the system model, followed by re-running the analysis. This iterative process serves to pinpoint and enhance the accuracy of detected anomalies, aligning the results more closely with the nuanced realities of the system.

Chapter 5

Modeling systems for static analysis

Following the introduction of a DSL in the previous chapter as an interface for users in STRIVE's anomaly detection pipeline, this chapter dives deeper into the DSL's objectives, design philosophy, and grammar, as it aims to bridge the gap between system designs and practical and effective anomaly detection. This chapter will also focus on the broader topic of modeling systems for static analysis and the challenges that come with it.

5.1 DSL

5.1.1 Objectives and Design

The DSL, as an illustrative example of system modelling, is designed with two main objectives in mind: to abstract the complexity inherent in microservices architectures and to capture what is needed to detect anomalies. It does this by allowing users to provide descriptions of the system's architecture, which are subsequently refined into detailed models for anomaly analysis. The DSL's syntax and grammar, detailed below, offer a structured method to capture such descriptions, including service composition, data flow, and transactional behavior, without requiring in-depth technical detail.

5.1.2 Grammar

The DSL's grammar is designed to encapsulate the nuances of microservices systems effectively:

```
System ::= 'System' '{' Microservice+ '}'

Microservice ::= 'Microservice' Identifier '{'
       (Table | Operation | Endpoint)+ '}'

Table ::= 'Table' Identifier '{' 'columns:' '[' [Parameters] ']' '}'

Operation ::= 'Operation' Identifier '{'
              'type:' ('read' | 'write')
              'table:' Identifier
              'input:' '[' [Parameters] ']'
              'output:' '[' [Parameters] ']'
              'affected_columns:' '[' [Parameters] ']'
              'predicate:' BooleanExpression
              '}'
```

```
BooleanExpression ::= 'Any valid boolean first-order logic expression
    with parameters'
// Omitted for simplicity

Endpoint ::= ['Internal'] 'Endpoint' Identifier '{'
             'input:' '[' [Parameters] ']'
             'output:' '[' [Parameters] ']'
             'transactions:' '{' TransactionAtomicBlock+ '}'
             '}'

TransactionUnit ::= '[' FunctionCallList ']'

FunctionCallList ::= FunctionCall (',' FunctionCall)*

FunctionCall ::= LocalCall | RemoteCall

LocalCall ::= Identifier '(' [Parameters] ')'

RemoteCall ::= 'remote' Identifier '(' [Parameters] ')'

Parameters ::= Identifier (',' Identifier)*

Identifier ::= [a-zA-Z_][a-zA-Z0-9_]*
```

Description of the main components:

- **System:** The system is defined as a collection of one or more Microservices, each identified by a unique name, reflecting the hierarchical nature of microservices architectures.
- **Microservice:** A microservice is a core component of the system, representing an independently deployable service within the architecture. Each microservice can define one or more Tables, Operations and Endpoints.
- **Table:** Represents a database table, specifying the columns that are relevant for the operations and endpoints within the microservice. This component allows the DSL to model the data aspects of microservices.
- **Operation:** Represents a database operation (read or write) performed by the microservice. It includes references to the target table, input and output parameters, affected columns, and a predicate for conditional operations. This component is crucial for understanding the data flow and potential points of data contention.
- **Endpoint:** Defines an API endpoint of a microservice, including its input and output parameters and a set of TransactionUnits. The optional keyword ”Internal” can designate an endpoint as inaccessible for direct calls.
- **BooleanExpression:** A condition used in operations to specify the data selection criteria, aiding in the precise modeling of operations' behavior. This constructor will be explored in greater detail in the next chapter.

- **TransactionUnit:** A sequence of function calls that are executed atomically as part of an endpoint. Function calls can be local database operations (LocalCall) or remote calls to other microservice endpoints (RemoteCall), highlighting the distributed nature of microservices systems.

This DSL allows the user to construct diverse system models using textual descriptions that closely mirror the descriptions that would be made during the initial planning and development phases of the system's architecture. This DSL also aids in building a meta-model that accurately represents the system's architecture and behavior, such as the one illustrated in Figure 4.3. The meta-model serves as an abstract blueprint of the system, capturing key elements such as microservices, operations, and data flows, that provides a foundation for the analysis performed by STRIVE.

While the DSL above represents a possible solution towards abstracting and modeling systems for STRIVE's pipeline, it's important to note that a formal evaluation of the DSL has not been conducted within the scope of this thesis. The practical effectiveness of the DSL, particularly in terms of its usability, expressiveness, and impact on the anomaly detection process, remains to be empirically validated.

5.2 Modeling Detail and Scope

The selection of an appropriate level of abstraction and detail is critical not only in the design of the DSL but also across the broader spectrum of system modeling techniques employed for static analysis. A high-level abstraction, such as that utilized in the DSL design, offers significant advantages. It allows the anomaly detection algorithm to work at the design phase, when not a single line of production level code exists, while also working on existing systems, where only a text-based description is needed. The former being one of the main use cases, such as migrating an existing monolith to microservices, where the potential anomalies STRIVE aims to find start to arise. Moreover, it encourages rapid prototyping and architectural iteration, allowing for preemptive adjustments that can conserve time and resources during software development.

While this high abstraction helps the user conceptualize different system compositions, it could lead to lack of important details and over-generalizations. This can cause the anomaly detection algorithm to report false positives, which are identified anomalies that, in reality, would not affect the actual system. This contrasts with dynamic analysis tools, which, by analyzing running systems, tend to report fewer false positives.

Despite the potential for false positives, false negatives will not exist as long as all anomalies can be reached by the system's description. This happens because STRIVE works under a worst case scenario premise, adopting a conservative approach in its analysis, assuming the most challenging conditions under which serialization anomalies might occur. This emphasizes the necessity for careful evaluation of anomaly reports and precise and thoughtful system descriptions by the users.

5.3 Addressing false positives

False positives are a well-known consequence of performing static analysis over system abstractions. Here are some scenarios where false positives might arise due specific aspects not being captured in the proposed DSL:

- **Conditional Execution Paths:** The DSL lacks the ability to express conditional logic within transactions. This could lead to the detection of potential anomalies that, in reality, would never occur because the conditions for their occurrence are not met in actual system operation.

- **Domain-Specific Invariants:** Systems often have domain-specific invariants or business rules that ensure consistency and correctness. The anomaly analysis might flag transactions as potentially problematic when, in fact, their executions are constrained by these unrepresented rules to always maintain consistency.

- **Concurrency Control Mechanisms:** As STRIVE uses a pessimistic approach, it does not eliminate any sort of read or write anomaly from its analysis that would be prevented by concurrency control mechanisms. STRIVE wants to show the user the necessity of having these mechanisms in place.

  For instance, in Chapter 4, the anomaly being showcased occurs during the concurrent execution of withdraw and credit rating modification transactions targeting the same customer. A proposed solution was to implement a custom concurrency control mechanism in the system that prevents multiple operations from occurring simultaneously on the same customer. Mentioning this mechanism in the system's abstract model is currently not possible, thus assuming the mechanism exists, a false positive like the anomaly above is reported, as the analysis in the previous chapter showed.

- **Error Handling and Recovery Procedures:** Many systems implement sophisticated error handling and recovery procedures that ensure consistency even in the face of operations that might initially seem problematic.

While considering some of these aspects are a compromise of static analysis, other aspects can still be covered using a more primitive approach. Conditional executions paths and error handling procedures can be modeled, by unfolding these behaviors to a linear transaction. For instance, the Withdraw endpoint that was explored in the last chapter, was modeled after a very simple happy path execution, without error treatment. However, an error path can also be modeled as its own endpoint, for example:

```
Endpoint withdrawError {
        input: [customerid, amount]
        output: []
        transactions: {
                [remote getCustomerCreditRating(customerid)]
                [writeErrorLog(customerid, amount), cancelOperation()]
        }
}
```

This new endpoint for the Finance microservice captures situations where an error occurs after the getCustomerCreditRating endpoint call and the error is handled by an operation that writes to a log table, and the executions of other recovery procedures. This updates the Finance microservice's model for the analysis, as shown in 5.1.

By considering this updated model of the system, the anomaly detection will consider the error path being executed concurrently with all other transactions in the system, and could uncover anomalies during the error and recovery procedure itself, which could prove valuable. The same modeling strategy

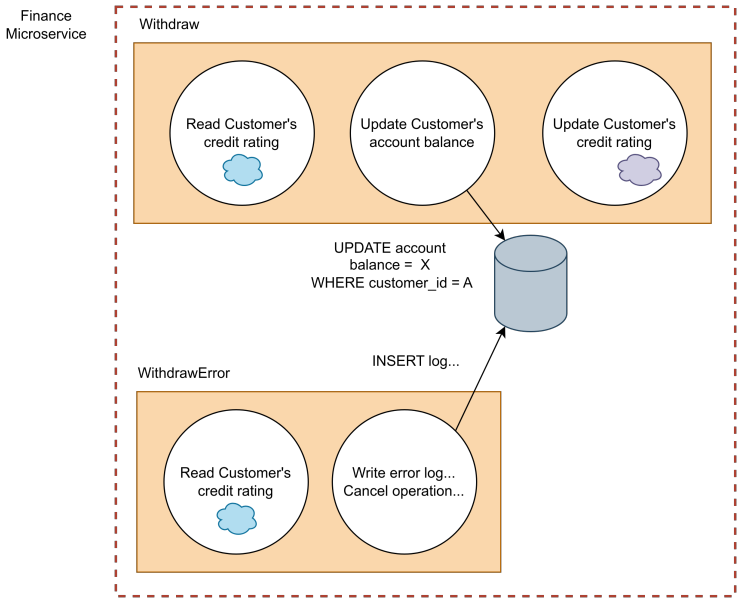

Figure 5.1: Finance microservice with Withdraw error endpoint

can be applied for transactions with conditional logic. An if-else statement and the different branches of a transaction can all be unfolded and linearized into a transaction or endpoint of their own.

Aside from the scenarios above, features like the predicates and the atomic transaction blocks were implemented to add some flexibility and mimic real mechanisms, in attempt to also reduce the number of false positives and increase the fidelity of reported anomalies. Predicates allows STRIVE to consider the logical dependencies and constraints inherent in the system's transactions by defining specific conditions under which database operations execute. This precision ensures that the framework's analysis is grounded in the actual logic of the system, rather than broad assumptions that could lead to identifying non-existent anomalies. On the other hand, atomic transaction blocks encapsulate groups of operations that the system intends to execute as a single unit, reflecting the ACID properties of transactional databases. This encapsulation helps STRIVE establish data dependencies correctly, focusing on more realistic group of database transactions, rather than examining each individual database interaction for potential dependencies. The next chapter will cover how these features are applied in practice.

Chapter 6

Anomaly Detection

This chapter will discuss in detail the anomaly detection algorithm, starting with the extraction of information from the DSL and following to the analysis of detected anomalies.

6.1 Graph Construction

As explained in the previous chapter, the objectives of the Domain-Specific Language (DSL) were to abstract the implementation details and focus on the microservice architecture and the distributed transactions. This abstraction requires only the essential information for anomaly detection without the need to describe all the implementation steps, or framework and infrastructure-specific code. From a microservice architecture described by the user in the DSL, the first step is to create a graph that represents the interactions between the different operations that make up a BusinessTransaction.

In particular, the graph models system operations as nodes and their interactions as edges. Later, the graph will be used to detect anomalies, as each cycle in the graph will correspond to one or more sequence of instructions that possibly cause anomalies.

6.1.1 Node Creation

Because we need to model several aspects such as LocalTransactions, BusinessTransactions and Microservices, the graph operates across three layers. At the lower-level layer are the database operations, acting as nodes where edge establishment takes place. These nodes are derived from the user's system description provided through the DSL and carry essential properties such as the associated database table, operation type (read or write), variables involved, and predicates. These database operation nodes are then grouped within LocalTransaction nodes. These nodes represent instances where database operations are executed atomically, thereby marking visible states of the system's data. This layer serves as the focal point for cycle detection. Furthermore, all LocalTransaction nodes linked to a common endpoint are grouped under the umbrella of BusinessTransaction nodes. These nodes are required to guarantee that each cycle starts in one LocalTransaction and concludes in another, within the same BusinessTransaction.

Considering this endpoint block of the DSL:

```
Endpoint E {
  input : [...]
  output : [...]
  transactions : {
```

```
    [operation_1(...), operation_2(...)]
    [operation_3(...)]
  }
}
```

three database operation nodes are created, operation_1, operation_2 and operation_3. The square brackets around operation_1 and operation_2 denote that they are executed atomically, so a LocalTransaction node is created around these two nodes. A separated LocalTransaction node is created for operation_3 since it is isolated in a different square bracket block. Then a BusinessTransaction node is created that gathers all these sub-nodes, since they are all part of the execution of endpoint E. This is depicted Figure 6.1.

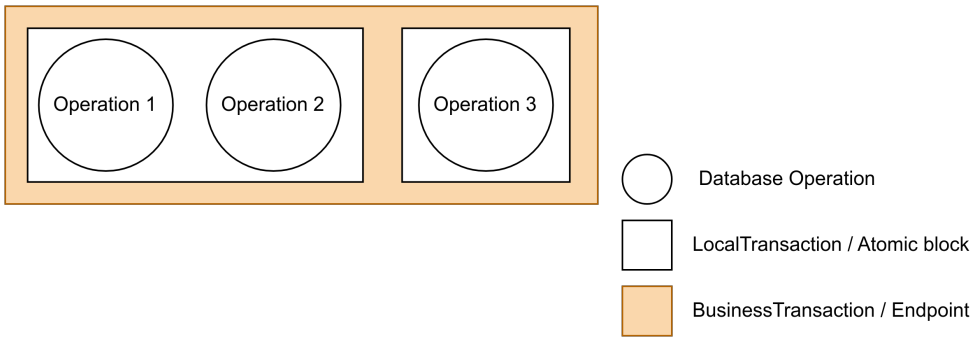

Figure 6.1: Basic endpoint node, with LocalTransactions and database operations

In the DSL, an Endpoint is encapsulated in a Microservice block, but there is no need to represent them explicitly in the graph the microservice itself plays no role in the anomaly detection. They are however needed to represent remote calls to another microservice. Processing these remote calls will be explained later, in Section 6.1.2, after the edge establishment algorithm. Two concurrent client calls to the same endpoint may also cause anomalies. As such, Section 6.1.3 will cover how we support two or more concurrent calls to the same endpoint in the graph construction.

6.1.2 Remote Call In-lining

In a microservice based system, it is very common for an endpoint to call other endpoints from other microservices during its execution, and this has to be modelable by STRIVE.

In practice, a remote call means that the logic being called remotely is a part of the context of the execution of the original call, and the algorithm does this by in-lining all LocalTransaction nodes being called, into the caller BusinessTransaction. An endpoint with a remote call can be described in the DSL with the ”remote” keyword, in the transactions list. The following is the DSL block that would reflect in Figure 6.2.

```
Endpoint A {
  transactions : {
    [operation_X(...)],
    [remote B(...)],
    [operation_Z(...)]
  }
}
```

```
Endpoint B {
  transactions : {
    [operation_1(...), operation_2(...)]
    [operation_3(...)]
  }
}
```

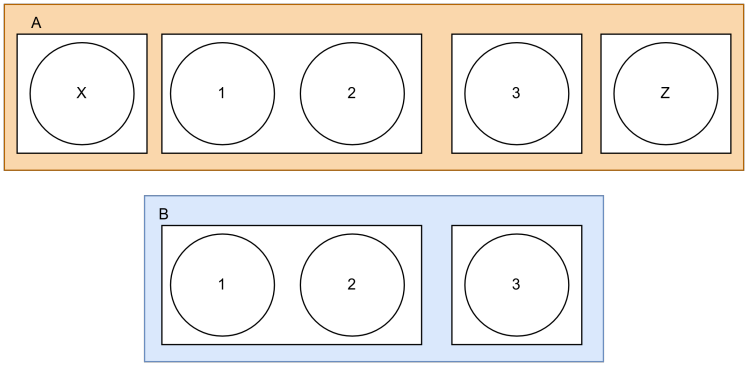

Figure 6.2: Example of a remote call in-lining

A remote call also has to be in an isolated atomic block, as the algorithm treats a remote call as an explicit loss of isolation from the previous transactions, implying that the database's state up to remote call is committed and thus visible. We do not consider scenarios where reads and writes remain uncommitted until the outcome of the remote call is determined, and we leave it for future work.

This method of processing remote calls respects how the graph is constructed, where the BusinessTransaction node is meant to encompass all database operations that are executed while traversing it. The BusinessTransaction node being remotely called is also kept without any additional edge, as it can be accessed directly.

6.1.3 Concurrent Transactions Cloning

The same endpoint can be called concurrently by multiple users (or even the same user). The probability of conflicts is higher in this scenario since concurrent calls to the same endpoint share the same logic and resources.

To model this concurrency, duplicate versions of the nodes for each endpoint are generated before any edges are established. If there are data dependencies between these duplicates, the graph will contain edges between them (or between them and other nodes). This approach intuitively reflects the fact that a write operation affects the same data as its concurrent counterpart, and a read operation dependent on a write is also dependent on the corresponding cloned operation. Therefore, conflicts between two concurrent calls of the same endpoint (e.g., A and A') are represented in the graph by edges between them. Similarly, edges established between two distinct endpoints (e.g., A and B) will also be established between A' and B. This is visualized in Figure 6.3. This interconnection ensures that interactions and dependencies are accurately depicted across concurrent executions, and anomalies of this nature will be found as the cycle detection algorithm traverse cloned nodes.

Each endpoint is cloned only once. If an anomaly traverses an endpoint and its clone, it indicates that

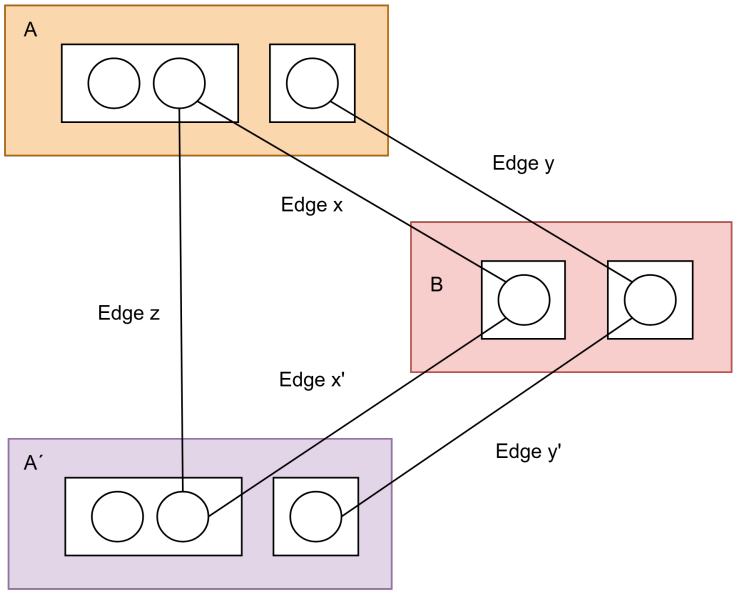

Figure 6.3: Example of Concurrent Endpoint Cloning

an anomaly would also be present among more than two calls of that endpoint; it would just be a longer path of that anomaly. Similarly, if an anomaly traverses the endpoint and two clones, there must be a shorter problematic path, as each clone has the same edges with all other nodes, and more than one clone just extends the path redundantly. Thus, one clone is sufficient to identify if an endpoint is problematic when called concurrently.

ACIDRain's algorithm addressed this by considering it a special property of a write operation, where a write node always interacts with itself, establishing a self-edge, and the cycle detection algorithm was adjusted for this special case. In STRIVE's case, this is simply reflected by creating copies, and the rest of the algorithm remains agnostic to this case, keeping the logic simplified.

Let us consider another scenario where an edge is established between a read and a write operation. Here, the write operation uses a value extracted by the read, which, in turn, relies on an endpoint input. The edge linking the read and its concurrent write counterpart assumes that the passed variable remains the same. However, in concurrent executions, the endpoint input may differ, leading to different IDs. An edge based on this ID could yield false positives. This highlights the importance of saving edge assumptions (i.e, the assertions) for later verification, rather than merely for establishing the edges.

This feature is embedded within the algorithm implementation and can be toggled on or off to facilitate analysis, allowing the inclusion or exclusion of concurrent versions of the endpoints, as this increases the number of nodes in the graph and the possible anomalies. Additionally, the algorithm enables limiting the number of parallel transactions traversed by a cycle. This capability allows for a faster, but limited, analysis.

6.1.4 Edge Establishment

In order to detect cycles, meaningful edges have to be established between the graph's nodes. There are three possible combinations of database operation pairs, depending on the operation types involved:

read-read, read-write, and write-write. A problematic serialization anomaly has at least one write, since executions of transactions with just read operations will never operate on wrong or outdated data, even with the lack of isolation.

While the nature of potential anomalies generated by the interactions of a read-write and write-write can be different, for example, read-write operations can induce a dirty read, while write-write operations can induce a lost update anomaly, the algorithm does not classify or distinguish the types of anomalies found, thus, the conditions for establishing edges between both types of operation pairs are the same.

Edges represent dependencies of two possible types. Table dependencies occur when the two operations access the same database table or columns. Variable dependencies exist when a subsequent operation uses values obtained by a previous operation. This is where the attributes declared in the DSL during an operation declaration come into play. The type to specify if the operation is a read or write. The table and affected_columns are used to find the table dependencies, and the input and output are used to find the variable dependencies.

Table dependency

This DSL description has two operations that will be connected due to a table dependency:

```
Operation read_age {
        type: read
        table: user
        input: [user_id]
        output: [user_age]
        affected_columns:[id, age]
        predicate: (...)
}

Operation update_age {
        type: write
        table: user
        input: [user_id, new_age]
        output: []
        affected_columns:[id, age]
}

// Endpoint with operation read_age
Endpoint {...}

// Endpoint with operation update_age
Endpoint {...}
```

Both read_age and update_age interact with the same table, user, and the same columns, id and age, despite existing in their own endpoint of the same microservice. These operations can lead to dirty reads or dirty writes of the age of users so, due to this table relation, an edge is established in order to form a cycle that will later be detected as an anomaly.

Variable dependency

In addition to table dependencies, operations can also be connected due to variable dependencies. Variable dependencies occur when the value read or written, by one operation is used by another operation, regardless of whether these operations interact with the same table or not. Consider the following DSL description where two operations are connected due to a variable dependency:

```
        Operation place_order {
                type: write
                table: order
                input: [user_id]
                output: [order_id]
                affected_columns:[id, user_id]
        }

        Operation notify_new_order {
                type: write
                table: notification
                input: [order_id]
                output: []
                affected_columns:[id, order_id]
        }

        // Endpoint with sequence of operations
        // place_order and notify_new_order
        Endpoint {...}
}
```

In this case, each operation targets a different table, but the notify_new_order operation creates a new notification record for the order_id of the order that was created in the previous place_order operation. This is modeled by the order_id variable in the place_order's output field matching the one in notify_new_order's input field, reflecting a variable dependency, thus establishing an edge.

The algorithm does not prioritize one dependency type over the other, considering them equal, particularly since both types can overlap when operations interact with the same table columns and share variables.

Both of the previous examples established edges under a weak assumption that any operation affecting the same table columns or involving the same variables inherently creates a dependency, regardless of specific conditions or constraints that might limit such interactions. For instance, the read_age and update_age operations can coexist without any data conflict at all, if they target different entries in the user table. Thus, establishing an edge in this example, and further detecting cycles using this edge, can lead to false positives in the anomaly analysis results. To mitigate this issue, the DSL supports user-defined WHERE predicates, that restrict the rows that are affected by the operation.

Predicates

Predicates play a crucial role in refining the conditions under which database operations interact. Each operation in the DSL is associated with a predicate that specifies a condition that must be true for the operation to execute. For instance, in the context of table dependencies, predicates ensure that edges are established only when the specified conditions are met, such as having matching IDs or filtering values in the relevant columns. In the case of the previous table dependency example, a predicate could be defined as:

```
Operation read_age {
        (...)
        input: [user_id]
        (...)
        predicate: user.id == user_id
```

```
}

Operation update_age {
        (...)
        input: [user_id, new_age]
        (...)
        predicate: user.id == user_id
}
```

The predicate user.id == user_id specifies that the column id in the user table must equal to the input variable user_id. This specification defines which records of the table are affected by the operation, and not limiting the edge establishment to column intersection. Now to establish an edge, a logical conjunction between the predicates of the two nodes is made, user.id == user_id $\wedge$ user.id == user_id. It is important to note that while both operations use a variable called user_id, it is effectively a different variable, due to variables being unique by name, in each endpoint. For clarity, let's rename the user_id variable name to match the respective operation, such as user_id_read_age and user_id_update_age, and now the predicate becomes user.id == user_id_read_age $\wedge$ user.id == user_id_update_age. The predicate translates to: "The operations have a dependency if and only if both endpoints are called with the same user ID", which is much more powerful and precise than simply establishing an edge because both operations seem to interact with the same table columns.

In a similar manner, the variable dependency example would also have predicates to increase the edge establishment's precision:

```
Operation place_order {
        (...)
        input: [user_id]
        (...)
        predicate: order.user_id == user_id
}

Operation notify_new_order {
        (...)
        input: [order_id]
        (...)
        predicate: notification.order_id == order_id
}
```

Figure 6.4 reflects the edges established in this running example.

A predicate can be any boolean first-order logic expression, with the common operators (==, !=, and, or, $<$, $>$), that can reference variables in the input and table columns, in order to establish conditions where the operation is executed. These predicates are evaluated with z3, an SMT solver, and in order to the edge to be established, the logical conjunction between predicates has to be satisfiable.

These predicates could just be evaluated for satisfiability where a cycle is found, but performance wise it is much better to check the satisfiability at each edge, to prevent establishing impossible edges right away, speeding up the cycle detection process. As it will be demonstrated in the cycle detection algorithm section, a cycle can be unsatisfiable even if all individual edges are satisfiable.

When evaluating predicates that reference variables it is also important to consider that the variables might have been constrained by prior operations. Figure 6.5 shows an example, where an edge between the "update user" operation and the "update non admin user" operation can be established when looking only at the two operations, but actually cannot when considering previous operations. The predicate

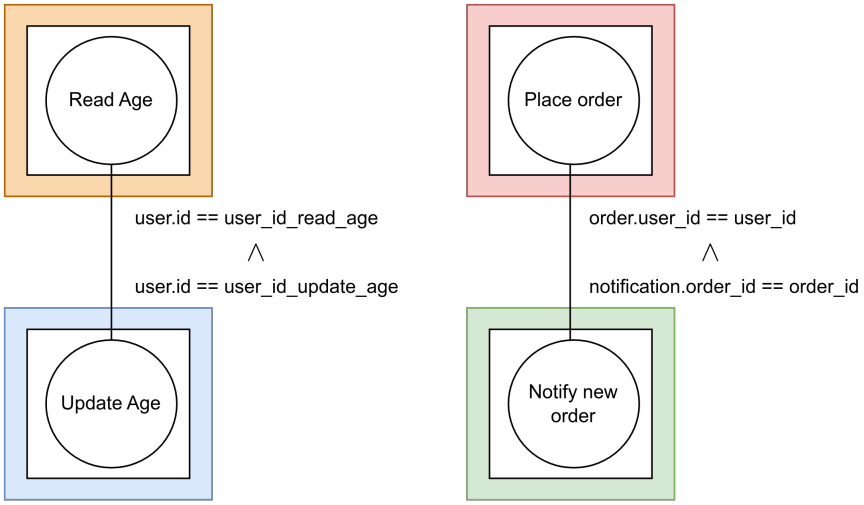

Figure 6.4: Edges established in the age and order examples

user.id == admin_ids ∧ (user.id == user_id ∧ ¬ user.admin) seems satisfiable to the SMT solver without anymore context, but the admin_ids variable was actually an output of a operation where the predicate user.admin was asserted, which contradicts the ¬ user.admin. Past predicates of all previous LocalTransactions also need to be considered in a logical conjunction with the predicates of the pair that is being evaluated, in this case: ((user.age == target_age ∧ user.admin) ∧ (user.id == admin_ids ∧ (user.id == user_id ∧ ¬ user.admin)), which is now unsatisfiable. This ensures the edge establishing process respects all historical conditions affecting the current operation, thereby maintaining the integrity and consistency of the data flow and logical dependencies.

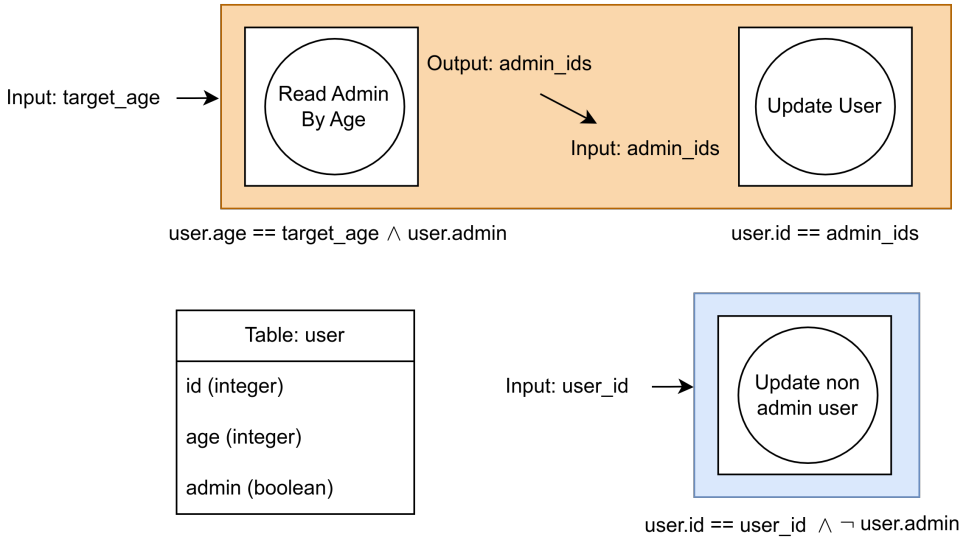

Figure 6.5: Example where past assertions affect subsequent edges

As mentioned before, it is also possible to establish edges without predicates. However this approach generates more false positives due to the lack of context verification, but it speeds up the writing of the

DSL description since there is no need to write predicates or even define input/output variables. The analysis is also faster because the SMT solver is not used. This method shifts much of the responsibility to the user to verify if the detected anomalies are meaningful, knowing that the cycles were found in a forced manner. For smaller systems, this trade-off between the effort required to describe the system and the accuracy of detected anomalies might be acceptable.

Algorithm 1: Pseudocode of the edge establishment algorithm

```
1  Function establish_edge:
2  |  LT1, LT2
3  solver ← smt_solver;
4  assertions ← [ ];
5  foreach pair of operations (o1, o2) between LocalTransaction LT1 and LT2 do
6  |  if LT1 and LT2 are in the same BusinessTransaction then
7  |  |  variable_dependency ← intersect(o1.output, o2.input) or intersect(o2.output,
   |  |    o1.input);
8  |  end
9  |  table_dependency ← (operationsInteractSameTableColumns(o1, o2) and (o1.type is
   |    WRITE or o2.type is WRITE)
10 |  if not variable_dependency and not table_dependency then
11 |  |  skip;
12 |  end
13 |  assertions.append(o1.predicate ∧ o2.predicate);
14 end
15 solver.add(z3.Or(assertions));
16 foreach LT in past of LT1 and LT2 do
17 |  foreach operation in LT.operations do
18 |  |  if intersection(o.output, LT1.input) or intersection(o.output, LT2.input) then
19 |  |  |  solver.add(o.predicate)));
20 |  |  end
21 |  end
22 end
23 result ← solver.check;
24 return (result == satisfiable, solver.assertions);
```

Algorithm 1, presents the consolidated version of the edge establishment procedure. The establish_edge function is applied to each pair of LocalTransaction nodes, attempting to establish an edge between them, as cycle detection occurs at this node level. An edge exists if dependencies arise between the database operations executed within the pair.

The algorithm begins by iterating through all operation pairs and evaluating whether there is a table and/or variable dependency between the operations. The variable dependency check is inside a condition (line 6) that specifies this is only checked if the LocalTransactions being evaluated are in the same BusinessTransaction, as the input and output variables are propagated within the context of a single endpoint. If a dependency exists, a logical conjunction of both predicates is added to a list of assertions (line 14). After all pairs have been iterated through, the list of assertions is added to the SMT solver with a logical disjunction between each element (line 14), since at least one of the logical conjunctions between the operation pairs has to be true to establish an edge.

Next, as previously mentioned, all predicates of past operations must be added to the solver to include past assertions. In the context of a BusinessTransaction, all LocalTransactions are executed in order, so ”past” refers to all LocalTransactions up to the current one.

Finally, the function returns the solver's result. If the set of assertions is satisfiable, the edge is established. The edges established in the graph are not directed, reflecting the symmetric nature of the conflicts between transactions. The function also returns the assertions used, which will later be used to validate the candidate cycles.

In summary, the predicates capture complex dependencies and add row-level precision, ensuring that operations are only considered dependent under specific conditions, which minimizes false positives. They also improve debugging and troubleshooting by clearly outlining the conditions for dependencies, as it will be shown in the output analysis section. This level of detail aids in modeling real-world constraints and business rules more accurately, increasing the reliability and robustness of the anomaly detection algorithm.

6.2 Cycle Detection

As previously discussed in Section 4.4.2, the primary goal of the algorithm is to identify cycles between BusinessTransaction nodes, aiming to discover interleavings within the endpoint execution. To achieve this, the search is conducted within the LocalTransaction node layer of the multi-graph. Each search begins at a valid starting node (see Section 6.2.1), explores the edges established by the algorithm in Section 6.1.4, and terminates when it encounters a different LocalTransaction node from the starting point, provided that they belong to the same BusinessTransaction node. There are also some additional conditions for a cycle to be valid, explained in Section 6.2.2. Considering the valid starting nodes and the valid cycle conditions, the pseudocode in Algorithm 2 illustrates the cycle detection algorithm.

6.2.1 Valid Starting Nodes

While the algorithm can theoretically start from any node, there are potential performance optimizations. For instance, if a BusinessTransaction node contains only one LocalTransaction node, it is impossible to return to the same BusinessTransaction node, rendering any search starting at that BusinessTransaction node futile.

Additionally, nodes in cloned endpoints created through the concurrent transaction cloning process are not considered as starting nodes, as all relevant anomaly cycles for these cloned endpoints can be found through the original ones. In other words, all cycles detected through the clone have a corresponding cycle that starts in the original node.

Another optimization involves not starting the search from LocalTransaction nodes that are accessible from other LocalTransaction nodes within the same BusinessTransaction. This is because initiating a search from one LocalTransaction node in a BusinessTransaction will naturally encompass the search through its subsequent neighboring nodes.

Besides performance optimizations, some endpoints might only be called in specific flows, and not be called directly, on their own. In the DSL, this can be done by marking the endpoints as ”internal”. This will eliminate cycles that start from these endpoints.

Consider the scenario depicted in Figure 6.6. The valid starting nodes for the search, based on the outlined conditions, are highlighted in red. Node B is excluded because it is reachable from node A. For instance, a cycle like B→1→D that starts from node B can also be detected by starting the search from node A as A→B→1→D. Since the search from A will naturally include all paths that pass through B, starting from B is redundant. Similarly, nodes 1 and 2 are excluded because their corresponding endpoint, Z, is marked as an internal endpoint, meaning that cycles originating from these nodes are not considered. Finally, node M is excluded because it is the only node within its respective endpoint, making any search starting from it irrelevant.

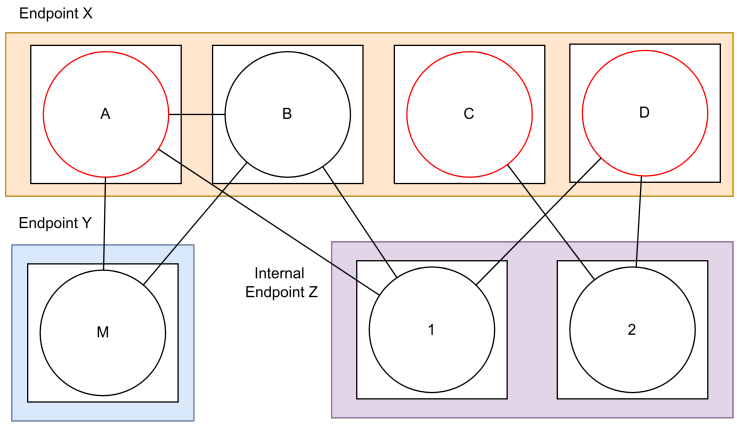

Figure 6.6: Example of endpoints X and Y with highlighted starting nodes

6.2.2 Filtering Invalid Cycles

When a cycle is found, by the search starting at one LocalTransaction node of one endpoint and ending in a subsequent LocalTransaction node of the same endpoint, it does not necessarily mean that cycle corresponds to potential anomalies. Firstly, the cycle path must also include nodes from at least two distinct endpoints to indicate the presence of interleaving. Furthermore, the cycle paths must follow the chronological sequence, aligning with the sequence of LocalTransaction nodes within the endpoint nodes to map to potential executions accurately. It would not make sense to consider cycles that violate the temporal sequence as anomalies as they would never occur in reality. Considering this filter, node D from Figure 6.6 can also be excluded from a valid starting node, since D is the last node of that BusinessTransaction node and there is no cycle that would respect the temporal sequence. Finally, as mentioned in Section 6.1.4, the edges are annotated with specific assertions. During cycle validation, it is necessary to assert whether the combination of all assertions associated with the edges used in the cycles can be satisfied or not. This step eliminates impossible anomalies with respect to low-level constraints.

6.3 Anomaly Reporting

Once cycles are identified, the next step is to create a sequence of operation execution that would cause an anomaly (modulo the abstraction of our model). Each cycle contains only some of the LocalTrans-

Algorithm 2: Cycle Detection Algorithm

```
Data: starting_node
Result: cycles
cycles ← [];
queue ← [];
queue.append((starting_node, [starting_node]));
while queue is not empty do
    current_node, current_path ← queue.pop();
    if current_node is in the same business transaction as starting_node and current_node is
     not starting_node then
        isCycle, assertions ← valid_cycle (current_path);
        if isCycle then
            cycles.append((current_path, assertions));
            continue;
        end
    end
    for neighbor in neighbors of current_node do
        if neighbor not in current_path then
            queue.append((neighbor, current_path + [neighbor]));
        end
    end
end
return cycles;
```

action nodes of the involved endpoints. However, to address anomalies related to endpoint execution interleavings, it is essential to explore the complete endpoint contexts. This is achieved by constructing a Directed Acyclic Graph (DAG) for each cycle. These DAGs model the dependencies of each node within the cycle, with edge directions indicating the execution sequence. After constructing the DAGs, they are topologically sorted to obtain valid sequences, representing the execution order of the original cycle. These sequences are valuable for investigating anomalies, reproducing them in real systems, or preventing them in future systems.

6.3.1 DAG Generation

The construction algorithm of the cycle's dependency DAG is outlined in Algorithm 3. The algorithm generates a Directed Acyclic Graph (DAG) for each cycle detected in the previous phase. For each node in the cycle, it retrieves the associated BusinessTransaction and iterates through its LocalTransactions. It establishes edges in the DAG to represent the execution sequence of these LocalTransactions, up to the current node being iterated. Finally, it adds an edge from the current node to the next node in the cycle. This ensures that the resulting DAG maintains the chronological order of the execution as defined by the cycle. By constructing a DAG, we can visualize and analyze the sequence of operations that originates the corresponding anomaly.

Let us consider again the previously mention cycle A→B→1→D from Figure 6.6. This cycle indicates that there is an anomaly when Endpoint X and Z are executed concurrently, particularly when node 1 is executed between node B and D. Node C is also executed since it is a part of endpoint X,

but the cycle does not contain node C, and the same goes for node 2. Therefore, creating a DAG is an important step to consider every LocalTransaction that is being executed, and to explicitly show the user what LocalTransaction sequences trigger the anomaly, which are not obvious to extract just by reading the cycle. The dependency DAG for this cycle is showed in Figure 6.7.

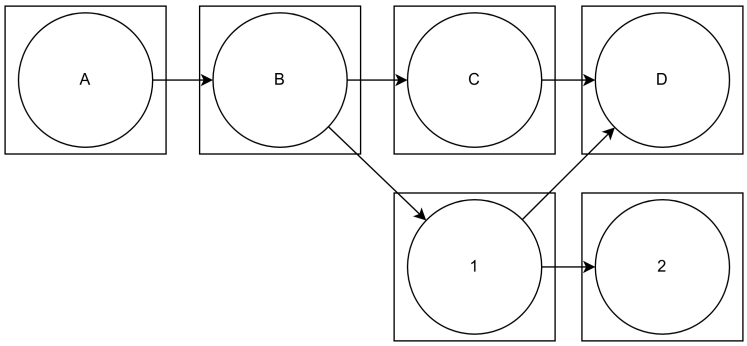

Figure 6.7: Dependency DAG of cycle A→B→1→D of Figure 6.6

Algorithm 3: Generate Dependency DAG

Input : graph, cycle
Output: dag

```
1  dag ← empty_graph;
2  foreach (node, next_node) in cycle do
3      node_business_transaction ← graph.get_business_transaction_of (node);
4      foreach (current_LT, next_LT) in node_business_transaction do
5          if current_LT == node then
6              break;
7          end
8          dag.add_edge(current_LT, next_LT);
9      end
10     dag.add_edge(node, next_node);
11 end
12 return dag;
```

6.3.2 Topological Sorting

After constructing the DAG for each cycle, the paths to reproduce the anomaly need to be extracted. This is done by performing topological sorting on the DAGs, generating sequences of LocalTransactions that maintain the constraints defined by the DAG and are necessary to reproduce the anomaly. In the context of anomaly reporting, these sequences help map out the exact sequence of transactional events that lead to anomalies. They provide a clear understanding of the interleavings and conflicts within the transactions, making it possible to reproduce and debug anomalies in real systems or prevent them from occurring in hypothetical systems. Considering the DAG from Figure 6.7, these are the possible paths:

- A → B → 1 → 2 → C → D
- A → B → 1 → C → 2 → D

- A → B → 1 → C → D → 2
- A → B → C → 1 → D → 2
- A → B → C → 1 → 2 → D

Each of these paths trigger the anomaly provoked by the interleavings found from the original cycle, in particular, LocalTransaction 1 being executed between B and D.

This process abstracts the complexity of identifying anomalies embedded in the cycles and graphs and simplifies the overall analysis by reporting problematic step-by-step executions of operations that were originally described in a high-level system's description.

Chapter 7

Case Studies and Evaluation

This chapter provides an overview of various case studies that were used to evaluate the functionality of STRIVE. It covers different scenarios to observe how STRIVE identifies potential anomalies in microservice architectures.

Section 7.1 discusses the migration from a monolithic system to microservices, with a focus on how STRIVE might be used in this context.

Section 7.2 compares the results of STRIVE with other tools in detecting anomalies.

The evaluation process of this tool heavily relies on a specific description language for defining the system and analyzing the output, consequently, the examples were manually constructed and adapted.

7.1 Migrating a Monolith to Microservices

Migrating from a monolithic architecture to microservices is a common approach for improving scalability, flexibility, and maintainability in software systems. This section provides an overview of this migration process, using the Food To Go (FTGO) system as an example. FTGO, discussed in "Microservices Patterns" by Chris Richardson [22], illustrates how a monolithic system can be broken down into microservices. This section explores this migration and how STRIVE can help identify and prevent potential anomalies.

7.1.1 Food To Go

Food To Go (FTGO) is a popular system with a microservice architectured that is studied in Microservices Patterns [22], as an example of migrating a monolith to microservices. Its business logic consists of users placing food orders, managing deliveries, restaurants and payments.

The backend is composed of multiple services: Order service that manages orders, Consumer Service that manages the user's information; Delivery service that manages the delivery of orders from restaurants to consumers; Restaurant Service that maintains information about restaurants; Kitchen service that manages the preparation of orders; Accounting service, that handles billing and payments. As mentioned in the book, a simple view of the main flow of creating a new food order is as follows: The Order Service creates a new order, the consumer service verifies that the user can place an order, the kitchen service validates the order's details and creates a new ticket, The accounting service authorizes the user's credit card, and if successful, the ticket can then proceed and eventually the order is fulfilled. These interactions between the different services are critical, and need to mantain data consistency.

In a monolith architecture, the data consistency is maintained trivially if all services share the same database with ACID properties. Taking a description of this monolith architecture as input for STRIVE the read-write dependency graph, and no cycles are detected since there is only one node, confirming that no serialization anomalies can occur.

Now, considering a decomposition of the monolith in microservices, a natural decomposition that follows a 1-1 mapping from services is presented in Figure 7.1. This can be modeled into STRIVE by creating a microservice for each of the domain's services. The operations that occur between the create and approve order transactions are opportunities of interleaving, that can be detected.

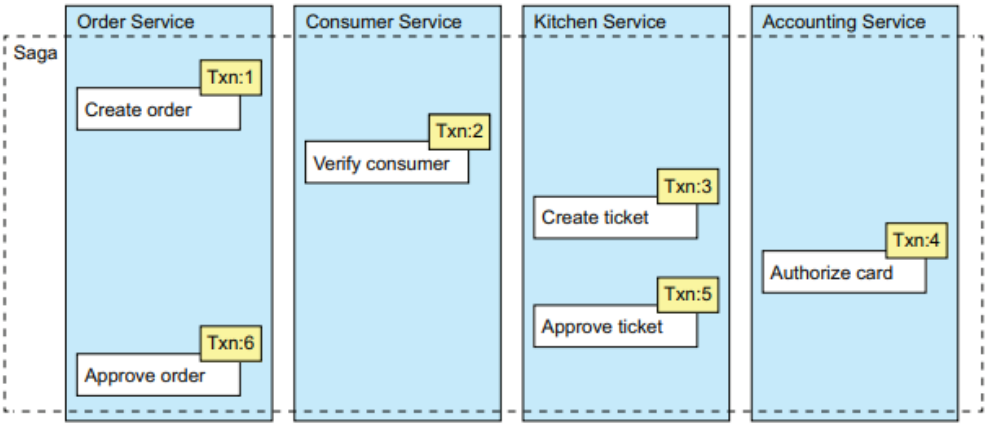

Figure 7.1: FTGO Order operation in a microservice architecture (Figure 4.2 from [22])

One critical point of data consistency between all these services is the connection between an order and a ticket. The order service creates an order and keeps track of its status, and the Kitchen service creates tickets associated to orders. Some of the detected cycles show interleavings between the order and ticket operations, meaning that if a new order is created while a new ticket is also being created for the same order in parallel, some anomalies can occur. These errors are preventable since the order service acts like an orchestrator, and the kitchen service is never accessed directly as an entry point. This approach is a common practice in microservice-based systems and can also be restricted with STRIVE, by making all microservices internal, except the order service. This eliminates the anomalies created by conflicts of parallel calls between all microservices.

The remaining reported cycles are conflicts between parallel new order operations, which represents a common issue of a concurrent anomaly, and if the developers know that the system does not allow this to happen, then the cycles are taken care of, and no anomalies are left, which is a good indicator that the current microservice structure is good.

Anomalies can arise when errors occur in the system, such as when a new order transaction fails and a reject order must be executed, which can also be vulnerable. For instance, in a SAGA, compensation transactions are commonly used to handle such errors. STRIVE can model these compensation transactions just as it models the original order transaction, making them targets for anomaly detection. This approach also allows for this detection to check interactions between happy paths and error paths.

If the services were to be refactored, such as by merging the order and kitchen services to prevent anomalies between the two domains, STRIVE would assist in identifying cycles within this refactor and

any other microservice compositions.

7.2 Comparison with other tools

As an attempt to test STRIVE with benchmark-like examples, BenchBase was used. BenchBase [8] is a test suite specifically designed for benchmarking relational databases, providing a framework for generating and executing workloads that simulate online transaction processing (OLTP) scenarios. CLOTHO was also tested with examples from BenchBase, so some results will be compared. Additionally, STRIVE will evaluate a specific case from the same authors of CLOTHO, featuring a tool that automatically proposes refactors to avoid anomalies. This use case highlights the importance of refactoring the database schema to avoid serialization anomalies. Finally, STRIVE will explore case studies from ACIDRain to compare the results between the two tools.

7.2.1 Comparison with CLOTHO on OLTP Benchmarks

An OLTP (Online Transaction Processing) benchmark simulates real-world scenarios where multiple users concurrently perform transactions on a database system. These transactions involve various operations such as inserting, updating, or querying data. The benchmark measures different performance metrics like transaction throughput, response times, and scalability. In this evaluation, the benchmarks' SQL queries are used as a basis for constructing inputs for STRIVE, to capture the essence of OLTP workloads and evaluate the algorithm's capability to detect serialization anomalies.

Four benchmarks were tested: TPC-C, Smallbank, Twitter, and Voter. However, directly comparing the results between STRIVE and CLOTHO presents significant challenges due to the fundamental differences in the underlying principles and objectives of each algorithm. CLOTHO is designed to detect serialization anomalies in distributed databases, focusing primarily on cyclic dependencies across concurrent operations in a weakly consistent environment. In contrast, STRIVE is tailored specifically for microservices architectures, where it models and analyzes the interaction between services to detect potential anomalies. These differences in focus lead to variations in the types of anomalies each tool is designed to detect, and how the anomalies are reported. For instance, CLOTHO's methodology emphasizes detecting cycles in operations across different database replicas, which may not directly correspond to the inter-service communication patterns analyzed by STRIVE. Therefore, direct comparisons what and how many anomalies each tool found will not be made, but Twitter and Voter showed interesting differences, which will be explored in the following sections.

Twitter Benchmark

The Twitter benchmark is a benchmark that simulates a microblogging social media platform similar to Twitter. It involves various operations such as tweeting, getting followers, and getting tweets from users.

These are the SQL procedures used in this benchmark:

- **GetTweet:**
  ```
  SELECT * FROM "tweets" WHERE id = ?
  ```
- **GetTweetsFromFollowing:**

  - **getFollowing:** `SELECT f2 FROM "follows" WHERE f1 = ?  LIMIT 20`
  - **getTweets:** `SELECT * FROM "tweets" WHERE uid IN (??)`

- **GetFollowers:**
  - **getFollowers:** `SELECT f2 FROM "followers" WHERE f1 = ?  LIMIT 20`
  - **getFollowerNames:** `SELECT uid, name FROM "user_profiles" WHERE uid IN (??)`

- **GetUserTweets:**
  `SELECT * FROM "tweets" WHERE uid = ?  LIMIT 10`

- **InsertTweet:**
  `INSERT INTO "added_tweets" VALUES (default, ?, ?, ?)`

Each procedure was converted to an endpoint to simulate a system with a microservice with these operations. For this benchmark, CLOTHO reports 2 anomalies, whereas STRIVE detects no anomalies. Upon examination of the dependency graph created by STRIVE, it was observed that the only write operation is an isolated node, since the target table is `added_tweets`, and there is no read targeting this table. This results in no interleavings occuring between any of the procedures, resulting in no cycle being found, suggesting that there are no serialization anomalies within the benchmark. Despite the anomalies reported by CLOTHO, no definitive conclusion could be drawn regarding the nature of these anomalies. Given that Clotho is designed to work in distributed environments, it likely detected anomalies where the system's operations across different replicas could lead to non-serializable behaviors. These detected anomalies might not be present in a single-instance or isolated microservice environment (as modeled by STRIVE), but Clotho's ability to simulate distributed environments with replicated databases makes it sensitive to potential inconsistencies that arise from the replication process itself.

Voter Benchmark

The Voter benchmark simulates a phone-based talent show voting system, where voters are allowed a limited number of votes.

This benchmark consists of a single transaction with four SQL operations:

- **CheckContestant:**
  `SELECT contestant_number FROM contestants WHERE contestant_number = ?`

- **CheckVoter:**
  `SELECT num_votes FROM v_votes_by_phone_number WHERE phone_number = ?`

- **CheckState:**
  `SELECT state FROM area_code_state WHERE area_code = ?`

- **CheckState:**
  `INSERT INTO votes (phone_number, state, contestant_number) VALUES (?, ?, ?)`

To model a microservice environment, this procedure was implemented with four LocalTransactions of one endpoint, maximizing the possibility of interleavings between all operations. Since there is only one transaction, any anomalies that may occur can only arise from concurrent executions of that same transaction. A vote in excess can be wrongly inserted by a user with the same phone number if it is executed at the same time. CLOTHO does not detect an anomaly on this example, but it may have considered an execution with higher isolation than the one used by this algorithm. CLOTHO might also be focusing on detecting serialization anomalies that involve operations across different database replicas. Since the Voter benchmark involves a single transaction with sequential operations and the anomaly arises from concurrent execution, CLOTHO might not have flagged it as an issue, particularly if it assumes that the operations are properly serialized or isolated by the database system.

7.2.2 Refactoring Database Schema

The authors of CLOTHO have also worked on database schema refactoring to repair serializability bugs [21]. The motivating example is an online course management application, which presented several anomalies, described in Figure 7.2, and after the transactions and database schema were refactored, the anomalies were no longer present (Figure 7.3). After modeling the transactions, STRIVE identified nine cycles with the original schema, and no cycles with the refactored schema, showing that this algorithm can correctly detect anomalies according to the author's refactoring tool.

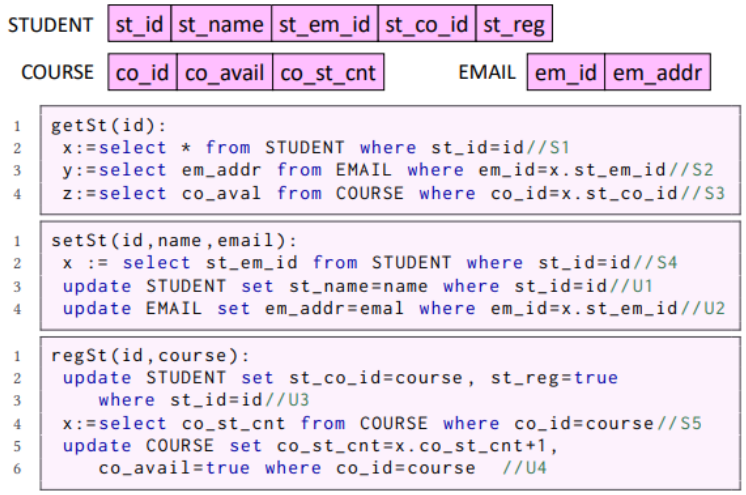

Figure 7.2: Database schemas and code snippets from an online course management program (Figure 1 from [12])

7.2.3 Comparison with ACIDRain on E-Commerce Examples

ACIDRain focuses on detecting and preventing concurrency-related attacks on database-backed web applications by identifying anomalies that could compromise application integrity, using runtime logs as input. While ACIDRain and STRIVE differ in some respects, both algorithms share the goal of detecting anomalies caused by concurrent operations that lead to non-serializable states in their respective systems.

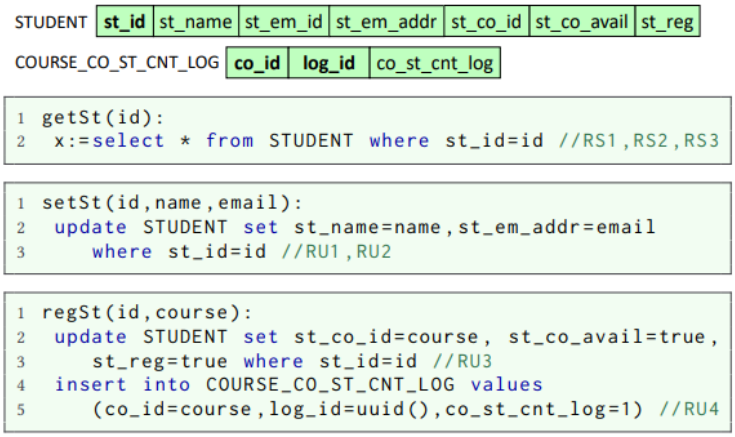

Figure 7.3: Refactored transactions and database schemas (Figure 3 from [12])

ACIDRain has reported anomalies in several e-commerce applications, and we will examine whether STRIVE can also identify these anomalies.

Oscar

The Oscar eCommerce application has a voucher system. For a single-use voucher to be used, a read operation checks if it was already used, and then a write operation applies the voucher. Logs from this application show that a voucher vulerability can occur if these two operations are not isolated, and two applications of the same voucher happen at the same time, allowing a voucher to be applied twice. After modeling these two operations, STRIVE can detect a path between the read operation to the write operation of a parallel call, and back to the write operation of the first call, creating a cycle and reporting this anomaly.

To demonstrate, the following python code from STRIVE models the operations involved in the voucher application process:

```
def oscar(self):
 voucher_application_t =
  Table("voucher_application_t",
   [Column("id", int), Column("user_id", int)])

 select_voucher = Operation(
   "select_voucher",
   [InputParameter("voucher_id", int)],
   OperationType.READ,
   voucher_application_t,
   lambda ctx: voucher_application_t.column("id") == ctx["voucher_id"],
   ["voucher_id"])

 insert_voucher = Operation("insert_voucher",
   [InputParameter("voucher_id", int)],
   OperationType.WRITE,
   voucher_application_t,
   lambda ctx: voucher_application_t.column("id") == ctx["voucher_id"],
   ["voucher_id"])

 select_voucher_lt = LocalTransaction([select_voucher],
  select_voucher.params)
```

```
insert_voucher_lt = LocalTransaction([insert_voucher],
 insert_voucher.params)

checkout_bt = BusinessTransaction(
  "checkout",
  [select_voucher_lt, insert_voucher_lt],
  [InputParameter("voucher_id", int)])

ms = Microservice("oscar", [checkout_bt])

system = System([ms])

graph = create_graph_from_system(system)

cycles, paths, topological_paths_for_cycles, cycle_assertions =
 get_cycles_and_dag_paths(graph)
```

Running this script produces the following output, which confirms the detection of an anomaly:

```
"System endpoints:
  [['select_voucher']checkout (1), ['insert_voucher']checkout (2)]
  [['select_voucher']checkout2 (3), ['insert_voucher']checkout2 (4)]
Number of detected cycles: 1
cycle:
 (['select_voucher']checkout (1),
 ['insert_voucher']checkout2 (4),
 ['insert_voucher']checkout (2))

assertions: [And(voucher_application_t.id == voucher_id_checkout,
    voucher_application_t.id == voucher_id_checkout2)]

Number of topological paths 2
[('3', '1', '4', '2'), ('1', '3', '4', '2')]"
```

Magento

The Magento checkout anomaly is a vulnerability in the Magento application that allows an attacker to exploit a race condition in the checkout process. The vulnerability arises because Magento attempts to use SELECT FOR UPDATE to lock the database row before writing it, but the read operation used in the inventory check is made outside of the transaction, thus Magento is still vulnerable. This vulnerability is caused by database accesses that are not properly encapsulated in transactions, allowing concurrent API requests to trigger the vulnerability independent of the level of isolation provided by the database backend. STRIVE is also capable of detecting this anomaly, since the SQL operations form a cycle reflecting the mentioned lost update.

To demonstrate, the following Python code from STRIVE models the operations involved in the stock management process of the Magento eCommerce platform:

```
def magento(self):
 catalog_inventory_stock_item_t =
  Table("catalog_inventory_stock_item", [Column("id", int), Column("stock", int)])

 check_stock = Operation(
  "check_stock",
  [InputParameter("item_id", int)],
  OperationType.READ,
  catalog_inventory_stock_item_t,
  lambda ctx: catalog_inventory_stock_item_t.column("id") == ctx["item_id"],
```

```
    ["stock"])

update_stock = Operation(
 "update_stock",
 [InputParameter("item_id", int)],
 OperationType.WRITE,
 catalog_inventory_stock_item_t,
 lambda ctx: catalog_inventory_stock_item_t.column("id") == ctx["item_id"],
  ["stock"])

check_stock_lt = LocalTransaction([check_stock],
 check_stock.params)
decrease_stock_lt = LocalTransaction(
 [check_stock, update_stock], check_stock.params)

checkout_bt = BusinessTransaction(
 "checkout",
 [check_stock_lt, decrease_stock_lt],
 [InputParameter("item_id", int)])

ms = Microservice("magento", [checkout_bt])

system = System([ms])

graph = create_graph_from_system(system)
cycles, paths, topological_paths_for_cycles, cycle_assertions =
 get_cycles_and_dag_paths(graph)
```

Running this script produces the following output, which confirms the detection of an anomaly:

```
"System endpoints:
 [['check_stock']checkout (1), ['check_stock', 'update_stock']checkout (2)]
 [['check_stock']checkout2 (3), ['check_stock', 'update_stock']checkout2 (4)]

Number of detected cycles: 1

cycle: (['check_stock']checkout (1),
 ['check_stock', 'update_stock']checkout2 (4),
 ['check_stock', 'update_stock']checkout (2))

assertions: [And(catalog_inventory_stock_item.id == item_id_checkout,
 catalog_inventory_stock_item.id == item_id_checkout2)]

Number of topological paths 2
[('3', '1', '4', '2'), ('1', '3', '4', '2')]"
```

The output indicates that STRIVE successfully identifies a cycle involving concurrent operations in the Magento stock management process. Specifically, it detects a sequence of operations where a read (check_stock) is followed by an update (update_stock) in a parallel transaction (checkout2), forming a cycle.

Lightning Fast Shop

ACIDRain analyzed logs from the Lightning Fast Shop application revealing a shopping cart vulnerability. There is a read operation that gathers the pricing information from all items in the shopping cart. The order items are calculated with this read operation but the order total is calculated using a different read operation. This means that in between the calculation of the order items and the order total, a new item can be inserted into the order and not be accounted for, and allow an exploiter to purchase items at

a lower price. This lack of isolation between the two read operations creates an interleaving between the operations, which also allows STRIVE to detect this cycle and report this anomaly.

To demonstrate, the following Python code from STRIVE models the operations involved in the shopping cart and checkout process of the Lightning Fast Shop platform:

```
def lightning_fast_shop(self):

 cart_item_t =
  Table("cart_item_t",
   [Column("id", int), Column("cart_id", int), Column("product_id", int)])

 order_t = Table("order_t", [Column("id", int)])

 order_items_t = Table("order_items_t", [Column("id", int)])

 insert_to_cart = Operation(
    "insert_to_cart",
    [InputParameter("product_id", int),
     InputParameter("amount", int),
     InputParameter("cart_id", int)],
    OperationType.WRITE,
    cart_item_t,
    [],
    ["id", "cart_id", "product_id"])

 select_cart_items = Operation(
    "select_cart_items",
    [InputParameter("cart_id", int)],
    OperationType.READ,
    cart_item_t,
    lambda ctx: cart_item_t.column("cart_id") == ctx["cart_id"],
    ["id", "cart_id", "product_id"])

 insert_order = Operation("insert_order",[], OperationType.WRITE, order_t, [], [])

 insert_order_items =
  Operation("insert_order_items", [], OperationType.WRITE, order_items_t, [], [])

 insert_to_cart_lt = LocalTransaction([insert_to_cart],
    insert_to_cart.params)

 insert_order_lt = LocalTransaction(
    [select_cart_items, insert_order],
    select_cart_items.params)

 insert_order_items_lt = LocalTransaction(
    [select_cart_items, insert_order_items],
    select_cart_items.params)

 insert_to_cart_bt = BusinessTransaction(
    "insert_to_cart",
    [insert_to_cart_lt],
    [InputParameter("amount", int),
     InputParameter("product_id", int),
     InputParameter("cart_id", int)])

 order_bt = BusinessTransaction(
    "order",
    [insert_order_lt, insert_order_items_lt],
    [InputParameter("cart_id", int)])
```

```
ms = Microservice("lfs", [insert_to_cart_bt, order_bt])

system = System([ms])

graph = create_graph_from_system(system)
cycles, paths, topological_paths_for_cycles, cycle_assertions =
   get_cycles_and_dag_paths(graph)
```

Running this script produces the following output, which confirms the detection of an anomaly:

```
"System endpoints:
[['insert_to_cart']insert_to_cart (1)]

[['select_cart_items', 'insert_order']order (2),
 ['select_cart_items', 'insert_order_items']order (3)]

[['insert_to_cart']insert_to_cart2 (4)]

[['select_cart_items', 'insert_order']order2 (5),
 ['select_cart_items', 'insert_order_items']order2 (6)]

Number of detected cycles: 1 cycle:
(['select_cart_items', 'insert_order']order (2),
 ['insert_to_cart']insert_to_cart (1),
 ['select_cart_items', 'insert_order_items']order (3))

assertions: [cart_item_t.cart_id == cart_id_order]

Number of topological paths 1 [('2', '1', '3')]"
```

The output indicates that STRIVE successfully identifies a cycle in the Lightning Fast Shop's workflow, where operations in the cart and order systems are executed concurrently. Specifically, a sequence of operations starting with a read (select_cart_items) followed by a write (insert_to_cart) and another write (insert_order_items) creates a cycle.

Chapter 8

Future Work

This work has defined an approach for detecting transaction consistency problems in microservices. However, there are still various aspects that remain unexplored or could be improved. This chapter suggest possible directions for future research and development.

8.1 Support for Additional Isolation Levels

The current model primarily focuses on anomalies that occur due to weak isolation levels in microservices. Future work could explore the extension of this model to support a broader range of isolation levels, such as Read Committed, Repeatable Read, or Serializable. This would allow for more comprehensive anomaly detection across different database configurations and usage patterns.

8.2 Branching

Currently, transactions are modeled as a sequence of operations, but in reality, transactions can change behavior during runtime. While STRIVE can model these different behaviors, by creating a transaction for each possible path, doing so is computationally inefficient and imposes additional effort on the user by requiring each behavior to be described as a separate transaction. A potential solution could be to expand the transaction definition to support conditional operations, such as using an if-else syntax in the DSL. In the graph algorithms, this could be represented by nodes with different edges to the respective nodes based on those if-else conditions, while the cycle detection process would remain unchanged.

8.3 Edge Pruning

Currently, STRIVE establishes edges based on the transactions and conditions defined by the user. However, if the user is aware that the system prevents certain conflicts but STRIVE still defines edges, they could manually prune those edges. This would eliminate cycles involving those edges. While this pruning can currently be done by directly modifying the internal graph generated by the algorithm, the possibility of incorporating this input directly from the user through the DSL or another mechanism has not been explored. Potential solutions could include allowing the user to define boolean expressions in the DSL to be treated as system invariants and add them to the SMT solver's context when establishing

edges, or making the algorithm interactive, enabling the user to modify the graph and run subsequent iterations with the updated graph.

8.4 Case Study Expansion

Additional case studies involving a variety of systems could be conducted to validate the effectiveness of the proposed solution in diverse environments. This would not only help to generalize the findings but also uncover new challenges and edge cases that need to be addressed in future iterations of the tool.

8.5 Remote Calls in Isolated Atomic Blocks

Currently a remote call is modeled in an isolated atomic block, as the algorithm treats a remote call as an explicit loss of isolation from the previous transactions, implying that the database's state up to the remote call is committed and thus visible. The concept of permitting remote calls without exposing the database's state, a scenario where reads and writes remain uncommitted until the outcome of the remote call is determined, can be valuable to allow STRIVE to model systems where this occurs. The solution to support this is probably around copying remote nodes into an existing node and not breaking isolation, but this was not tested.

8.6 Parser Development

The DSL was developed to understand what is needed to be extracted from a system in order to identify anomalies, however, the parser itself was not developed. During the development and testing, the DSL descriptions were manually converted to the internal meta-model of the algorithm. Developing the parser would allow to automate this process and make this tool a more user-friendly experience.

Chapter 9

Conclusion

The shift from monolithic to microservices architectures, while beneficial in terms of scalability and flexibility, introduces complexities in ensuring that data remains consistent across distributed services. This thesis proposed a methodology n the detection of distributed transaction consistency problems.

STRIVE, leveraging static analysis techniques, was developed to detect potential serialization anomalies within microservices. By modeling system operations and their interactions as a graph, the algorithm identifies problematic cycles that could lead to data inconsistencies. The process begins with the user providing a high-level description of the system using the Domain-Specific Language (DSL). This description serves as the input, which the algorithm then transforms into an internal graph representation of the system's operations and dependencies. The algorithm proceeds by analyzing the graph to detect cycles that represent potential anomalies, ultimately producing a report that highlights these issues as its output. This approach allows for the early detection of anomalies, aiding developers in addressing these issues during the development phase, rather than after deployment.

The evaluation of STRIVE was carried out through case studies, including comparisons with similar tools an one example drawn from the literature. Although these tools vary in their approaches and results, the evaluation confirmed STRIVE's effectiveness in identifying anomalies within distributed systems.

Despite the advancements presented in this work, several areas remain open for future research. These include extending the algorithm to support additional features, refining the DSL, and developing the parser.

In summary, the work presented in this thesis provides a valuable foundation for improving the reliability of microservices architectures by enabling early detection of transaction consistency issues. While progress has been made, continued research and development will be crucial in enhancing these tools and approaches to fully meet the challenges posed by increasingly complex and distributed software systems.

Bibliography

[1] Sagas. *ACM SIGMOD Record*, 16, 1987. ISSN 01635808. doi: 10.1145/38714.38742.

[2] Ashish Aggarwal and Pankaj Jalote. Integrating static and dynamic analysis for detecting vulnerabilities. In *30th Annual International Computer Software and Applications Conference (COMPSAC'06)*, volume 1, pages 343–350, 2006. doi: 10.1109/COMPSAC.2006.55.

[3] Luciano Baresi, Martin Garriga, and Alan De Renzis. Microservices identification through interface analysis. volume 10465 LNCS, 2017. doi: 10.1007/978-3-319-67262-5_2.

[4] Lucas Brutschy, Dimitar Dimitrov, Peter Müller, and Martin Vechev. Static serializability analysis for causal consistency. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, page 90–104, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356985. doi: 10.1145/3192366.3192415. URL https://doi.org/10.1145/3192366.3192415.

[5] Tomas Cerny, Andrew Walker, Jan Svacina, Vincent Bushong, DIpta Das, Karel Frajtak, Miroslav Bures, and Pavel Tisnovsky. Mapping study on constraint consistency checking in distributed enterprise systems. 2020. doi: 10.1145/3400286.3418257.

[6] Rui Chen, Shanshan Li, and Zheng Li. From monolith to microservices: A dataflow-driven approach. volume 2017-December, 2018. doi: 10.1109/APSEC.2017.53.

[7] Transation Processing Performance Council. Tpc-c, 2019.

[8] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudré-Mauroux. Oltp-bench: An extensible testbed for benchmarking relational databases. *PVLDB*, 7(4):277–288, 2013. URL http://www.vldb.org/pvldb/vol7/p277-difallah.pdf.

[9] Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil, and Dennis Shasha. Making snapshot isolation serializable. *ACM Trans. Database Syst.*, 30(2):492–528, jun 2005. ISSN 0362-5915. doi: 10.1145/1071610.1071615. URL https://doi.org/10.1145/1071610.1071615.

[10] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman Publishing Co., Inc., USA, 2002. ISBN 0321127420.

[11] J. N. Gray. Notes on data base operating systems. volume 60 LNCS, 1978. doi: 10.1007/3-540-08755-9_9.

[12] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1992. ISBN 1558601902.

[13] Wuxia Jin, Ting Liu, Yuanfang Cai, Rick Kazman, Ran Mo, and Qinghua Zheng. Service candidate identification from monolithic systems based on execution traces. *IEEE Transactions on Software Engineering*, 47(5):987–1007, 2021. doi: 10.1109/TSE.2019.2910531.

[14] Sudhir Jorwekar, Alan Fekete, Krithi Ramamritham, and S. Sudarshan. Automating the detection of snapshot isolation anomalies. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, page 1263–1274. VLDB Endowment, 2007. ISBN 9781595936493.

[15] Anup K. Kalia, Jin Xiao, Rahul Krishna, Saurabh Sinha, Maja Vukovic, and Debasish Banerjee. Mono2micro: A practical and effective tool for decomposing monolithic java applications to microservices. 2021. doi: 10.1145/3468264.3473915.

[16] Daniel A. Menascé. Tpc-w: A benchmark for e-commerce. *IEEE Internet Computing*, 6, 2002. ISSN 10897801. doi: 10.1109/MIC.2002.1003136.

[17] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. volume 4963 LNCS, pages 337–340, 2008. ISBN 3540787992. doi: 10.1007/978-3-540-78800-3_24.

[18] Kartik Nagar and Suresh Jagannathan. Automated detection of serializability violations under weak consistency. In *29th International Conference on Concurrency Theory (CONCUR 2018)*, Leibniz International Proceedings in Informatics, page 41:1–41:18, 2018. doi: 10.4230/LIPIcs.CONCUR.2018.41.

[19] Vikram Nitin, Shubhi Asthana, Baishakhi Ray, and Rahul Krishna. Cargo: Ai-guided dependency analysis for migrating monolithic applications to microservices architecture. *ArXiv*, abs/2207.11784, 2022.

[20] Kia Rahmani, Kartik Nagar, Benjamin Delaware, and Suresh Jagannathan. Clotho: Directed test generation for weakly consistent database systems. *Proceedings of the ACM on Programming Languages*, 3, 10 2019. ISSN 24751421. doi: 10.1145/3360543.

[21] Kia Rahmani, Kartik Nagar, Benjamin Delaware, and Suresh Jagannathan. Repairing serializability bugs in distributed database programs via automated schema refactoring. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 32–47, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383912. doi: 10.1145/3453483.3454028. URL https://doi.org/10.1145/3453483.3454028.

[22] C. Richardson. *Microservices Patterns: With examples in Java*. Manning, 2018. ISBN 9781617294549. URL https://books.google.pt/books?id=UeK1swEACAAJ.

[23] Marc Shapiro and Pierre Sutra. Database consistency models, 2019. URL http://link.springer.com/10.1007/978-3-319-77525-8_203. https://pages.lip6.fr/Marc.Shapiro/papers/DBconsistency-Springer2018-authorversion.pdf.

[24] Gottfried Vossen. *ACID Properties*, pages 19–21. Springer US, Boston, MA, 2009. ISBN 978-0-387-39940-9. doi: 10.1007/978-0-387-39940-9_831. URL https://doi.org/10.1007/978-0-387-39940-9_831.

[25] Todd Warszawski and Peter Bailis. Acidrain: Concurrency-related attacks on database-backed web applications. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, page 5–20, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450341974. doi: 10.1145/3035918.3064037. URL https://doi.org/10.1145/3035918.3064037.