Automatic Detection of Anomalies in the Migration to Microservices Architectures

Valentim Dias Romão

Thesis to obtain the Master of Science Degree in

Computer Science and Engineering

Supervisors: Prof. Vasco Miguel Gomes Nunes Manquinho
Prof. Luís Eduardo Teixeira Rodrigues

Examination Committee

November 2023

Acknowledgments

I would like to thank my parents and sister for their friendship, encouragement, and caring over all these years, for always being there for me through thick and thin and without whom this project would not be possible. I would also like to thank my grandparents, aunts, uncles, and cousins for their understanding and support throughout all these years. I would also like to thank my girlfriend and best friend, Daniela Amaral, for her friendship, help, and motivation during these years.

I would like to acknowledge my dissertation supervisors Prof. Luís Eduardo Rodrigues and Prof. Vasco Miguel Gomes Nunes Manquinho for their insight, support, and sharing of knowledge that has made this Thesis possible. I would also like to acknowledge Rafael Soares for the fruitful discussions and comments during the preparation of this Thesis.

Last but not least, to all my friends and colleagues who helped me grow as a person and were always there for me during the good and bad times in my life.

To each and every one of you – Thank you.

Abstract

The microservices architecture allows structuring an application as a set of loosely coupled services. This architecture has several advantages, such as modularity and scalability, which motivate the migration of monoliths to microservices, despite the challenges posed by the lack of isolation between functionalities that require invoking multiple microservices. In a monolithic application, each functionality is typically executed as a single transaction that accesses a single database with ACID properties. In a microservices architecture, a functionality may be divided into multiple independent sub-transactions and each may be executed by a different microservice. The interleaving between these sub-transactions, when the functionalities execute concurrently, may lead to unexpected results, also called anomalies. In this work, we present a tool capable of automatically detecting these anomalies, as well as an experimental evaluation of the tool, using microbenchmarks and several real-world applications.

Keywords

Monolith; Microservices; Anomaly Detection; SMT.

Resumo

A arquitetura de microsserviços permite estruturar uma aplicação como um conjunto de serviços fracamente acoplados. Esta arquitetura tem várias vantagens, tais como modularidade e capacidade de escala, que motivam a migração de monólitos para microsserviços, apesar dos desafios colocados pela falta de isolamento entre funcionalidades que requerem a invocação de vários microsserviços. Numa aplicação monolítica, cada funcionalidade é tipicamente executada como uma única transação que acede a uma única base de dados com propriedades ACID. Numa arquitetura de microsserviços, uma funcionalidade pode estar fracionada em múltiplas sub-transações independentes e cada uma pode ser executada por um microsserviço diferente. O intercalamento destas sub-transações, quando as funcionalidades se executam concorrentemente, pode levar a resultados inesperados, também chamados de anomalias. Neste trabalho apresentamos uma ferramenta capaz de detetar automaticamente estas anomalias, assim como uma avaliação experimental da ferramenta, recorrendo a micro-testes de bancada e a várias aplicações realistas.

Palavras Chave

Monólito; Microsserviços; Deteção de Anomalias; SMT.

Contents

List of Figures

List of Tables

List of Algorithms

Listings

Acronyms

AR	Abstract Representation
FOL	First Order Logic
MAD	Microservices Anomaly Detector
SMT	Satisfiability Modulo Theories

1

Introduction

Contents

The microservices architecture structures an application as a set of loosely coupled components (services), contrasting with the traditional monolithic architecture composed of a single centralized component. This architecture has several advantages when compared with the monolithic architecture. Firstly, each microservice only implements the logic related to a small subset of the entities managed by the application, making the code of each service more cohesive and easier to develop and maintain. Secondly, each microservice can be developed independently, allowing the development of the application as a whole to be more agile. Thirdly, the architecture offers a more flexible management of the system, since the microservices can be managed independently. Due to these advantages, several companies are currently adopting the microservices architecture when developing their applications. In many cases, companies also migrate their pre-existing monolithic applications to the microservices architecture [1].

In this thesis, we address the problems that can arise when migrating from a monolith to a microservices composition, in particular, anomalies generated from previously impossible interleavings of concurrent functionalities. We make an overview of existing tools and techniques aimed at detecting anomalies and discuss how they can be applied to detect anomalies in microservices. Based on our findings, we describe the design, implementation, and evaluation of a new tool that manages to detect the anomalies that arise in a migration from monolith to microservices following a given decomposition.

1.1 Motivation

Although the microservices architecture has many advantages, it also introduces challenges. A monolithic application is traditionally composed of several functionalities, each commonly defined as a transaction that executes in a single database. These transactions offer the *ACID* (Atomicity, Consistency, Isolation, Durability) properties, therefore guaranteeing the isolation between concurrent executions of these functionalities. When migrating from a monolithic application to microservices, a functionality may need to be chopped into several sub-transactions, with each sub-transaction possibly executing in a different microservice, breaking the isolation of the functionality as a whole.

Typically, the microservices use the *database per service* [2] design pattern. Although each sub-transaction is isolated from the remaining sub-transactions that execute in the same microservice, the concurrent execution of functionalities may lead to interleavings between sub-transactions that execute in different microservices, something that did not occur in the monolith. This may lead to unexpected results, also called anomalies, which derive from non-serializable executions of transactions.

The number of anomalies that can arise during the execution of functionalities in microservices depends on how the monolith is decomposed, in particular, the number of microservices that interact with each other and which entities are managed by each microservice. Finding these anomalies using an accurate analysis technique can be useful, considering that concurrency anomalies are notoriously difficult to identify via testing [3]. One way of handling this problem is by having a tool capable of generating all the possible interleavings and comparing the executions that can occur in the monolith with the executions that can occur in a given decomposition since these new anomalous executions result from the migration. To the best of our knowledge, existing tools aimed at doing this sort of analysis do not account for all the monolith to microservices migration aspects, such as the chopping of a transaction into a sequence of independent sub-transactions, as well as the effects of the sub-transactions reading mutually inconsistent versions of remote objects by accessing their microservice's local storage. In this thesis, we describe the design and implementation of a novel tool, named *Microservices Anomaly Detector* (MAD), which can generate all the possible interleavings that originated from the decomposition of a monolith into microservices. The tool works by encoding the problem in a *Satisfiability Modulo Theories* (SMT) formula and using Z3 [4] to find the satisfiable assignments. We also performed an experimental evaluation to assess the accuracy and applicability of MAD using *microbenchmarks* and test cases inspired by real-world codebases.

1.2 Contributions

This thesis analyzes, implements, and evaluates techniques to detect anomalies that emerge from the migration of a monolithic application to a composition of microservices. Therefore, the thesis' main contributions are the following:

- A set of techniques to encode both a monolith and a microservice decomposition of that monolith in an SMT formula that can be used with an SMT solver to find satisfiable assignments, which represent executions that generate anomalies;
- A set of techniques to divide the analysis of a large search space into the analysis of several smaller search spaces, which are more adequate for combinatorial analyses.

1.3 Results

This thesis produced the following results:

- A tool named MAD capable of automatically detecting anomalies that arise when migrating from a monolith to microservices. This tool was implemented as a set of extensions and improvements to a pre-existing tool, named *CLOTHO* [5], originally designed to find anomalies in distributed databases;

- An experimental evaluation of MAD, focusing on a comparison with another tool with a similar purpose, MAD's applicability to real-world codebases, and the impact of our improvements.

1.4 Research History

This work was developed under the *DACOMICO* (Data Consistency in Microservices Composition) project, which focuses on addressing the data consistency issues that can arise in microservices compositions. Considering the goal of the project, a tool that can automatically detect anomalies when one migrates from a monolith to microservices is a relevant contribution to the project. Efforts to create such a tool have been initiated by previous researchers of the DACOMICO project [6].

Early results from this thesis have been published as:

- V. Romão, R. Soares, V. Manquinho and L. Rodrigues. Deteção Automática de Anomalias em Arquiteturas de Microsserviços. In *Actas do décimo quarto Simpósio de Informática (Inforum)*, Porto, Portugal, September 2023

This work was supported by FCT - Fundação para a Ciência e a Tecnologia, as part of the projects with references UIDB/50021/2020 and DACOMICO (financed by the OE with ref. PTDC/CCI-COM-/2156/2021).

1.5 Structure of the Document

The rest of the thesis is organized as follows: In Chapter 2, we introduce the relevant background concepts and present related works with similar purposes and techniques; In Chapter 3, we present our tool, MAD; In Chapter 4, we present the results obtained by our tool in our experimental evaluation; Finally, Chapter 5 concludes the thesis and mentions the system's current limitations and ideas for future works.

2

Related Work

Contents

In this chapter, we will introduce the background knowledge required to better understand our work. We also survey existing tools related to our goals. In Section 2.1, we describe the monolithic and microservices architectures. In Section 2.2, we explain what are transactions and the properties associated with them. In Section 2.3, we list the anomalies that can occur in a system. In Section 2.4, we present mechanisms for coordinating transactions. In Section 2.5, we discuss how transactions can be divided. In Section 2.6, we delve into the topic of having distributed databases. In Section 2.7, we address how data can be replicated between nodes. In Section 2.8, we explain how transactions are normally implemented in a microservices application. Finally, in Section 2.9, we present and compare tools whose goals and techniques are similar to ours.

2.1 Monolithic and Microservices Architectures

An application is said to follow a monolithic architecture when it is composed of a set of tightly coupled modules, organized in a single codebase, and deployed, provisioned, and executed as a single logical unit. One of the main advantages of a monolithic architecture is that it makes data management easier. It is often possible to store the application data in a single storage service, typically a database with support for transactions. This allows the functionalities to be implemented as ACID transactions, relieving the programmers from the burden of considering the effects that may result from the interleaving of concurrent executions. However, monolithic applications also have some disadvantages. As the application grows, and more functionalities are added, the codebase becomes bigger and more complex. Also, appropriate resource provisioning becomes harder, because different modules may have different resource requirements, but the application needs to be provisioned as a whole. Finally, if a component fails in a monolithic application, then the entire application becomes unavailable.

In contrast, the microservices architecture advocates the design of the application as a set of loosely coupled modules (or services), organized in multiple codebases, and that can be deployed, provisioned, and executed independently of each other. This architecture makes it easier to develop and maintain different services. It is also possible to assign different resources to different services and scale each service independently of the others. Additionally, if a service becomes unavailable, functionalities that can be executed using the remaining services continue to be available, allowing the system to suffer a graceful degradation. On the downside, executing a functionality as an atomic transaction becomes harder in microservices architectures. This happens because each service will typically use a different storage service, making it inefficient, or even impossible, to run a functionality that spans multiple services as a single transaction. Instead, operations in different services are run as independent transactions, allowing interleavings that would not occur in a monolithic implementation. Also, when a service needs to access data that is managed by another service, this requires the implementation of mechanisms to perform remote data accesses or mechanisms that allow a service to collect and cache updates performed to remote services. This requirement increases the complexity of the code and also allows the possibility of a service reading inconsistent data versions.

Many companies believe the advantages of the microservices architecture outweigh its disadvantages, and the adoption of this architecture has been increasing. In fact, some companies even initiated the process of transforming legacy monolithic applications into microservices compositions [1]. Currently, there are already some tools that ease the process of the monolith decomposition [7–9], making the migration more appealing. This trend, however, forces programmers to reason about the effects of concurrency and data consistency in a distributed setting, to mitigate and compensate for the fact that ACID transactions are not available for functionalities that span multiple services.

2.2 Transactions and Transactional Properties

A transaction is a sequence of one or more operations that are treated as a unit and whose execution appears to be indivisible. Transactions are widely used in database systems and are characterized by a set of properties known as the ACID properties, namely: **A**tomicity, which ensures that if the effects of one operation take place, then the effects of all operations take place; **C**onsistency, which guarantees that after applying a transaction to a consistent database state, the database remains consistent; **I**solation, that prevents the execution of a transaction from being affected by the concurrent execution of the same or other transactions; and **D**urability, which assures that after a transaction commits, their changes will be permanent in the database. One specific type of transactions that we will focus on in this chapter is long-lived transactions, which represent transactions that access many database objects and take a large amount of time to complete [10].

The use of transactions simplifies the development of an application because it shields the programmer from dealing with the effects of concurrency and/or partial failures. For instance, if a transaction implements a bank transfer, the atomicity property avoids the case where the withdrawal operation is executed in one account but the corresponding deposit is not performed.

Also, in the same scenario, isolation ensures that multiple transactions cannot withdraw more money than the funds available, even if they are executed concurrently.

It is possible to define different isolation properties for transactions. The strongest isolation properties are strict serializability and (the slightly weaker) serializability, which are defined as follows:

Serializability [11] defines that the results of the concurrent execution of a set of transactions should be the same as if the transactions have been executed in some serial order. This model does not account for the precedences between transactions in the real-time execution (one transaction starting after another being committed), it only focuses on guaranteeing that the ordering of the executed transactions is serial. A serial order is an order where a transaction only begins to execute after all the operations of the previous transaction are done, which means, no interleaving between transactions is allowed.

Strict Serializability [11] is the strongest isolation level, and enforces the same properties as serializability but with an additional constraint. This level does not only guarantee that the effects are perceived as they would in a serial execution (like serializability), but it also imposes that if a transaction T_2 starts after another transaction T_1 has been committed, then T_2 is serialized after T_1.

Unfortunately, ensuring serializability incurs costs that result from the need to perform concurrency control. As a result, transactions may experience longer latencies or be forced to abort. The overall throughput of the system may also be affected. These costs can be reduced if weaker isolation guarantees are provided. In practice, there is often a trade-off between the performance of the system and how strong the isolation is. Thus, in some cases, the systems are configured to offer weaker isolation guarantees, even if these guarantees allow interleaving among concurrent transactions to become visible, which may lead to inconsistent results. Examples of weaker isolation levels are read uncommitted, read committed, and repeatable read.

Read Uncommitted [12] enforces that the writes to a given object should be handled respecting the total order. However, this model also allows operations to read values that were written by other transactions but were not yet committed, resulting in a transaction being able to see intermediate values of other transactions' executions.

Read Committed [12] only allows operations to read values written by other transactions if they are already committed. This avoids reads seeing intermediate values of other transactions but does not enforce that if a transaction contains two reads to the same object, they will both see the same value.

Repeatable Read [12] is identical to Read Committed with the addition that if a transaction contains two reads to the same object, then those two reads need to see the same value.

2.3 Anomalies

Sometimes, it is easier to understand the limitations of using isolation levels that are weaker than serializability by identifying phenomena that may occur only when serializability or strict serializability are not enforced, and that may compromise the consistency of the application. These phenomena are typically called *anomalies*. In the following list, we present some of the most relevant anomalies based on a previous research work [13].

Dirty Write is an anomaly generated when the system has two transactions, T_1 and T_2, each having two updates and writing on the same two variables, x and y, respectively. Now, consider that T_1 modifies x, then T_2 interleaves T_1, and writes on x already modified by T_1, and on y. This will result in an inconsistency, because when T_1 resumes it will write on y, resulting in the final state being x written by

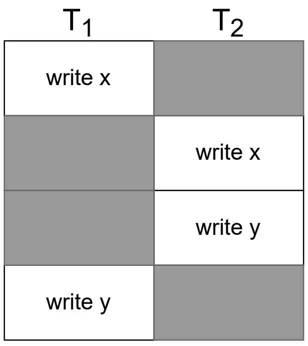
T_2 and y written by T_1. This anomaly is impossible under every isolation level. Example in Figure 2.1(a).

Dirty Read is an anomaly generated when the system has two transactions, T_1 and T_2, T_1 composed of two updates and T_2 composed of a read, all applied to the same variable in the database, x. Then the following scenario happens, T_1 executes its first update, but before it can execute its second update, T_2 reads that intermediate value from x in the database and uses it for an instruction in its code. This is an anomaly because T_2 will be using an intermediate value that would not be available in a serializable execution of the system. This anomaly is only possible under read uncommitted. Example in Figure 2.1(b).

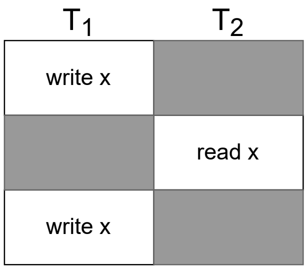
Lost Update is an anomaly similar to the dirty write, but, in this case, T_1 will not initially write to the same variable as T_2. Now, T_1 will read the variable, then T_2 reads and updates the variable, and T_1 finishes its job by updating the variable. This will result in the value prevailing to be the one written by T_1, nullifying the write made by T_2. This is an anomaly because it is not possible to order the two transactions in a serial manner. If we assume that T_1 executed before T_2, then T_2 should have read the value written by T_1, and vice-versa. Considering the scenario previously described, we cannot assume that one transaction started before the other, therefore there is no possible serial order of the transactions. This anomaly is possible under read uncommitted and read committed. Example in Figure 2.1(c).

Read Skew is an anomaly associated with a system having an invariant condition, and caused by the fact that all the updates of a transaction do not execute atomically. These two factors, together with a transaction that reads the values to verify if the invariant holds, may give the perception that the invariant was broken. Considering the scenario where we have a system's invariant with two variables, x and y, a transaction T_1 with two updates, one for each variable, and a transaction T_2 that will read both variables, x and y. If T_2 happens in the exact moment when the first update of T_1 has already executed, but the second one is still to be executed, then the program will act as if the invariant was broken, although T_1 was still executing, and would leave the system respecting the invariant when it finished. This anomaly is possible under read uncommitted, read committed, and repeatable read. Example in Figure 2.1(d).

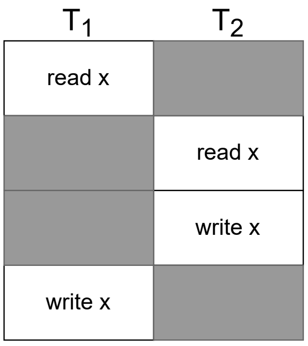
Write Skew is an anomaly, similar to the read skew, where one needs to account for the delay between the verification of a variable and the update of another variable. Considering the scenario where we have a system's invariant with two variables, x and y, a transaction T_1, which reads x and writes a value on y, if the value read from x and the updated value of y still respect the invariant, and a transaction T_2 with the same purpose as T_1 but reads y and writes on x. If T_1 starts executing and reads x, and T_2 interleaves the execution reading y. Then, T_1 and T_2 will decide if they update their respective invariant variable considering the value that they read from the other variable, not accounting for the possibility of other transaction(s) having updated its value after it was read. This sequence of events may break the system's invariant. This anomaly is possible under read uncommitted, read committed, and repeatable read. Example in Figure 2.1(e).



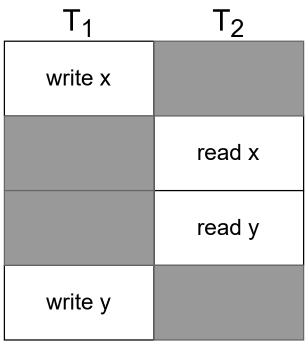
(a) Dirty Write.



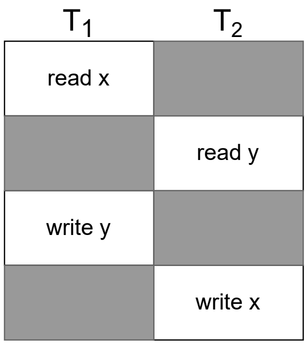
(b) Dirty Read.



(c) Lost Update.



(d) Read Skew.



(e) Write Skew.

Figure 2.1: Anomalies examples.

2.4 Concurrency Control

To assure the serial execution of the transactions, one needs to enforce isolation between transactions. One way of achieving this is by having *concurrency control* mechanisms responsible for coordinating the execution of the transactions. When two transactions are executing concurrently and access the same object, with at least one of them modifying the object, this is called a *conflict*. It is possible to ensure that potential conflicts do not cause anomalies by implementing *concurrency control* mechanisms. There are two main approaches to *concurrency control*, the pessimistic approach and the optimistic approach.

In the pessimistic approach, the goal is to prevent two transactions from accessing the same object and detect it as soon as it happens. A common way of assuring this is by assigning one or two locks to each object in the database. To access an object, a transaction needs to first obtain a read or a write lock on the object. The access mechanism ensures that no more than one transaction can own a write lock on an object and that, if some transaction owns a write lock, no transaction can own a read lock on that object and vice versa. A transaction that cannot obtain a lock is blocked until the lock(s) owner(s) releases the lock(s). Therefore, the correct use of locks can ensure serializability, since a transaction only releases the lock(s) when it commits or aborts. However, this mechanism is prone to deadlocks. Also, one issue with this approach is that long-lived transactions may hold locks for large periods, preventing other transactions from making progress or leading to deadlocks, since they tend to access a wide variety of objects.

In the optimistic approach, it is assumed that conflicts are rare, so transactions are executed without checking for conflicts until the commit time (typically in this approach updates are also not made visible until the commit time). Conflicts are only checked when a transaction attempts to commit by executing a *certification procedure*. The certification of different transactions needs to be executed in total order and verifies if the objects read and written by the transaction have not been subsequently updated by other transactions that have been committed after the operations were performed. If a transaction passes certification, the updates are applied atomically. Otherwise, the transaction is forced to abort and the resources used to execute the transaction are wasted. As stated before, long-lived transactions tend to access a wide variety of objects, which increases the risk of conflicts and makes them more prone to abort than other transactions. As a result of this, the system is punished by having long-lived transactions, since they are more likely to abort and, when it happens, it results in a significant waste of the system's resources.

2.5 Transaction Chopping

As noted previously, long-lived transactions may affect negatively the system's performance. This happens because long-lived transactions can either block the execution of other transactions for a long period (pessimistic approach) or have long executions that may need to be aborted and re-executed (optimistic approach). The concept of chopping transactions proposed by Shasha et al. [14] was introduced as a technique to avoid long-lived transactions. This method consists of dividing the large transactions into smaller transactions (sub-transactions). We define a sub-transaction as a transaction composed of a sub-sequence of operations from one transaction. The chopping method consists of identifying sub-sequences of operations from a transaction, considering the objects they access and the type of access, and based on that creating new sub-transactions. This results in an improvement of the system's performance, since the amount of time that other transactions that access the same resources will be blocked and the amount of work that may be wasted, and subsequently have to be repeated, are both decreased.

Associated with transaction chopping, a couple of aspects need to be considered. Suppose a transaction T_1 is chopped into k smaller transactions, T_1^1, T_1^2, ..., T_1^k. Although all these transactions are independent, T_1^i still needs to be executed before T_1^{i+1}, for all i between 1 and $k-1$, to provide the same logic as in the original transaction T_1. Moreover, the sub-transactions will act as the original transaction, but without providing isolation when they are executing, allowing for other transactions to execute in between sub-transactions. Another problem is that if a transaction has rollback statements, then they need to be placed in the first sub-transaction, otherwise, if one sub-transaction fails mid-execution, it might not be capable of undoing the work done by the previous sub-transactions, since they are independent of each other. When the chopping is performed on a transaction that either has no rollback statements, or all the rollback statements are in the first sub-transaction, the chopping algorithm considers that chopping to be *rollback-safe*.

In their work, Shasha et al [14] also delve into the topic of verifying if a chopping is correct. Their work states that for a chopping to be considered correct, all of its executions need to be equivalent to a serial execution of the original transactions. To accomplish this, they use an undirected graph to represent the executions, where the vertices are the transactions and the edges are the relations between transactions. The edges can be one of two types, S, between sub-transactions of the same original transaction, and C, between transactions that have conflicts with each other by accessing the same object. In the paper, this graph is called the *chopping graph*. After representing the execution as a *chopping graph*, the next step is to look for cycles that contain at least one S edge and at least one C edge. Each cycle with this format is an execution that would not preserve the serializable behaviour of the original transaction set, since it would denote that there is no possible serial ordering of the transactions executed that would respect the relations between them. Another requirement for a chopping to be correct is that the chopping is *rollback-safe*, to guarantee that the rollback behaviour is the same as the one in a serial execution. To summarize, a chopping is only correct if all of its executions can be represented as an acyclic chopping graph and the chopping is rollback-safe.

If the monoliths were decomposed using this technique, the existence of anomalies could be prevented since it is possible to ensure the correctness of the chopping. However, in the migration from monolith to microservices, the transactions are divided based on a different criteria. This criteria consists of the distribution of the entities by the microservices, which differs from only considering the operations and the transactions. In fact, in the migration case the chopping is enforced by the decomposition chosen, therefore not following the chopping algorithm and not assuring the correctness of the transactions' chopping.

To better illustrate how transactions are divided in a migration from monolith to microservices, we present an example scenario. In this example, we consider two entities with two attributes each, *Member* and *Item*, two transactions, *Txn1* and *Txn2*, and a decomposition where *Member* goes to microservice *M1* and *Item* goes to microservice *M2*. Both transactions, *Txn1* and *Txn2*, are composed of three operations. *Txn1*'s operations are reading an instance of *Member*, reading an instance of *Item* and updating the *Member*'s instance that was read. *Txn2*'s operations are reading an instance of *Member*, updating the *Member*'s instance that was read and updating an instance of *Item*. Considering that

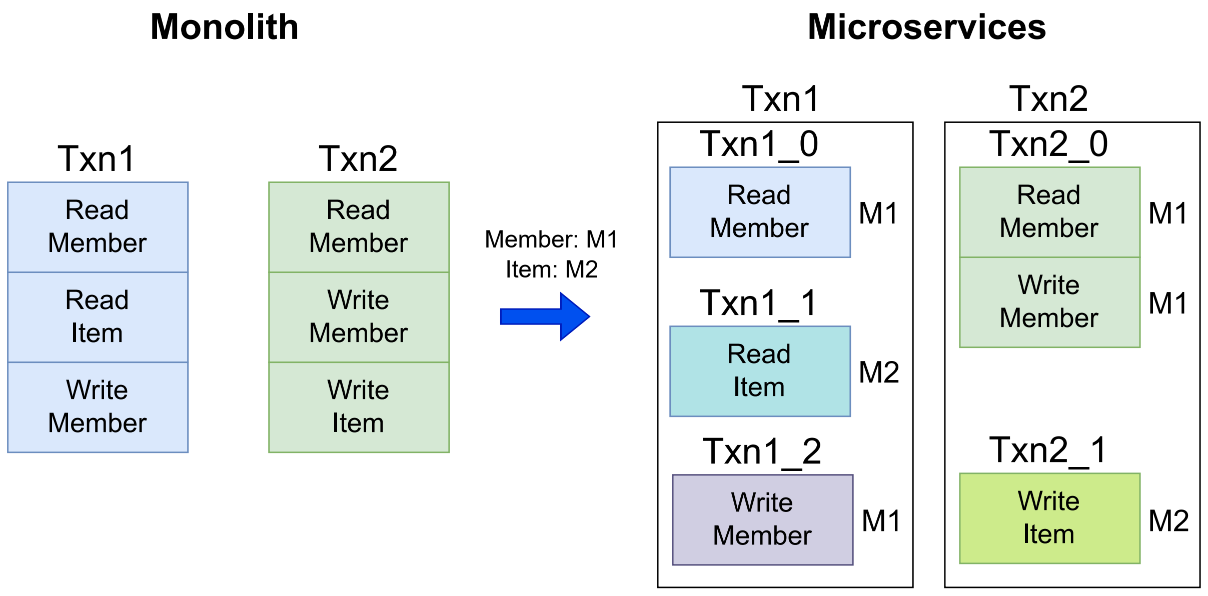


Figure 2.2: Example scenario of how the transactions are chopped in a migration from monolith to microservices.

Member and *Item* will be in different microservices, the transactions *Txn1* and *Txn2* need to be divided into several sub-transactions, each of which executes in the microservice associated with the entity they are accessing. In Figure 2.2, we present how the transactions would be divided in this scenario.

As we stated before since in the monolith to microservices migration the chopping is enforced based on the distribution of the entities between microservices, this chopping of the transactions is not guaranteed to be correct, and in this example scenario is in fact incorrect. This division of the transactions allows for multiple executions where the sub-transactions interleave with each other, something that was not possible in the monolithic version and can lead to unexpected results (anomalies). One example of a new possible interleaving that leads to an anomaly can be seen in Figure 2.3. In this case, the execution of *Txn2* interleaves with the execution of *Txn1*, which leads to a non-serializable execution of the system, since *Txn1* sees an older value for $Member_1$, assuming that it executed before *Txn2*, but at the same time it sees the new value for $Item_1$ that was written by *Txn2*, assuming that it executed after *Txn2*, which is contradictory.

2.6 Distributed Databases

Software developers sometimes opt to have more than one database in their system, since by doing this they can guarantee properties such as data separation, scalability, and fault tolerance, among others. These databases tend to run on different machines, distributing the load between them, which may also benefit performance. When the storage system is composed of several databases in different machines, it is possible to coordinate them, such that they execute as a single database. This type of storage

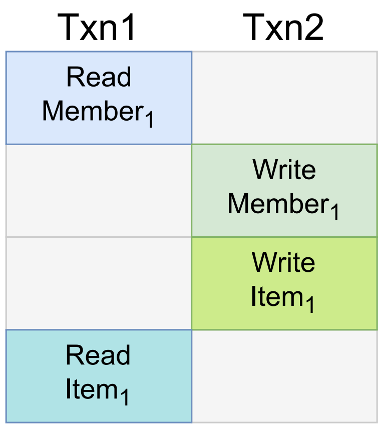


Figure 2.3: Example of a possible interleaving that leads to an anomaly.

system is called a *distributed database*. Coordination is needed to ensure both atomicity and isolation. For atomicity, one needs to ensure that if a transaction is aborted in one database, it is aborted in all databases. For isolation, one needs to ensure that all databases serialize concurrent transactions in the same order.

To ensure atomicity, it is necessary to execute an *atomic commitment protocol* among all participants in the transaction, to agree on the outcome of the transaction. A widely adopted *atomic commitment protocol* is the *Two-Phase Commit Protocol*, where one of the participants acts as a coordinator. This protocol initiates with a setup phase where the participants send a message to the coordinator, to inform it that they will participate in the execution of a transaction. After that, each participant executes the received transaction. The coordinator will verify with each participant if they want to commit or not. A participant will say “yes” if it successfully executed the transaction, and “no” if it aborted. The coordinator will collect all the responses, including its own, and decide the outcome of the transaction. If all the responses are “yes”, then the coordinator will send a message to all the participants telling them to commit. Otherwise, the coordinator will send a message to all the participants that answered with “yes”, to tell them to abort [15]. As one can tell, this protocol assures the atomicity of a transaction even in a distributed scenario, by having all database nodes either commit or abort the transaction.

The concurrency control will be done by using the techniques from Section 2.4 adapted to the context of distributed databases. In this context, the pessimistic approach will continue to hold locks for each object in the databases. In the distributed setting, this approach introduces a new problem, the distributed deadlock. This problem may occur because each database may execute the transactions in a different order. For example, consider we have two transactions T_1 and T_2 that will execute concurrently and two nodes N_1 and N_2, that will choose different orders to execute the transactions. Assume that in N_1 the order is T_1, T_2, and in N_2 the order is T_2, T_1. Considering this scenario, N_1 will begin by obtaining the locks of the database objects of T_1, and N_2 will do the same for the database objects of T_2. If this happens, then both nodes will be blocked, since they will not be capable of accessing the database objects of the transaction that they still need to execute. In the optimistic approach, the certification is done in parallel by an independent set of servers, each of them validating the transactions that access their respective objects. However, one needs to ensure that all participants validate transactions in the same order. A transaction only commits if it passes the certification at all databases.

2.7 Data Replication

To guarantee that the data of the system is available most of the time, one can resort to *data replication*. The way this method works is by creating copies of the original data and distributing them between different nodes of the system. By doing this, the system becomes tolerant to faults, since one node failing would not result in the loss of data, because there would be another node with the same data (replica). Another advantage is that, by distributing the copies between nodes that can be in distinct geographical places, one can decide to read from a local, or at least closer, replica to the origin of the request, decreasing the latency time and providing a faster response.

However, keeping all the replicas consistent with each other represents a hindrance of using this process. Ideally, one would want the replicated system to be 1-copy-equivalent, which means that the replicated system behaves the same way as a non-replicated system [16]. One naïve way of achieving this is to apply each update to all the replicas, and only proceed after all the replicas were updated. As one might expect, assuring 1-copy-equivalence is costly, and requires mechanisms that are strict regarding how data is managed. These mechanisms either slow down the execution or block it until a certain event occurs, resulting in the system offering a poorer performance, as well as a decrease in its availability.

Therefore, programmers tend to avoid these costs by adopting weaker consistency models and accepting the fact that data may not always be consistent in all replicas. A consistency model reflects the consistency guarantees offered by the system to the user when executing its functionalities. Depending on the model, only certain sequences of operations can be perceived. The weaker the consistency model, the more permissive the system is regarding its valid executions, implying more concurrency between transactions, which ultimately benefits performance. However, this can also lead to anomalous behaviours, if not handled correctly. These behaviours may occur because now operations from different transactions can interleave with each other and modify or read objects that will still be modified or read by mid-execution transactions. On the opposite side, we have stronger consistency models, which are stricter on how transactions interleave with each other. The stronger consistency models lead to safer and less anomalous executions, but neglect performance, since the system will offer less concurrency [13].

There are several consistency models that a programmer can use. The following list presents several consistency guarantees:

Eventual Consistency [12] guarantees that if several nodes are working with the same object, then, after some time without seeing new updates, every copy of that object will converge to the same value for all the nodes.

Causal Visibility [17] is a property that affects one specific object, and is derived from the concept of visibility and Lamport's *Happens-Before* relationships [18]. This property represents a causality relation between two operations (one operation needs to happen before the other operation), making all threads respect this order, and forcing the write effect of the first operation to be visible for the second operation. For example, if we have an operation $o1$ and an operation $o2$, and we establish a causal visibility relation between $o1$ and $o2$ ($o1 \rightarrow o2$), then we are expressing that $o1$ happens before $o2$ and that $o2$ sees the effect of the write made by $o1$.

Causal Consistency [17] enforces an ordering of the operations that respects their causal relations. We establish a causal relation (*Happens-Before*) between two operations when the operations execute one after the other in the same thread, or when they execute in different threads but one of them reads a value written by the other one. As an example, consider an operation, $o1$, that reads from a variable x, and an operation, $o2$, that writes on a variable y. If one thread executes $o1$, reading a version i of x, and after executes $o2$, creating a new version of y, j. Based on this example, the model will guarantee that all the threads that read version j of y cannot read a version of x that is older than version i.

Linearizability [11] is a property that affects one specific object and consists of all the operations applied to that object being seen as atomic, preserving the object's single-threaded semantics. For example, if an object is updated, then all the subsequent reads will see the updated version of that object.

2.8 Transactions in Microservices Architectures

When implementing a microservices application, software developers opt to use some of the techniques and mechanisms previously presented. Two aspects need to be considered regarding the microservices architecture. First, microservices are inherently distributed, since different services run in different machines, and manage different data. Second, we may have cases where a given service needs to access data from a different service, which may lead to consistency issues.

One way of assuring strong consistency is to run distributed transactions using an *atomic commitment protocol*. Although this type of protocol fixes the consistency issue, it worsens the system's performance, since, as we saw before, *atomic commitment protocols* are expensive procedures. The costs of using them come from the communication between the coordinator and the participants to run the protocol, as well as the time they need to wait for other participants to respond to proceed. For this reason, distributed transactions tend to be avoided in a microservices architecture.

Since distributed transactions are not a feasible option, developers opt for alternatives. These alternatives consist of chopping transactions and using weaker consistency models. The chopping alternative aims for each service to only have transactions that they can execute by accessing their local data. Note that by doing this, the operations of a transaction will be divided between several independent sub-transactions, and can no longer provide properties such as atomicity and isolation by themselves. Also, guaranteeing that one can chop transactions to be fully executed by running only on one service, may not always be possible, and we might have cases where one service still needs to access the objects of another service. When opting for the weaker consistency model alternative, one is aiming for the system to provide a better performance, by accepting the fact that the replicas of the system may not be consistent with each other all the time. To implement this alternative, the first requirement is that each service has a replica of the contents of all the other services that it needs to access. After that, when an update is made on a given service, for example, S_1, then the update will be propagated to all the other services that have a replica of the contents of S_1. Although this assures that each service has the objects it requires to execute, one needs to account for the fact that when an update is made on a service, the propagation to the remaining services is asynchronous and not instantaneous, which may lead to inconsistencies. These inconsistencies occur because there is a time window when the replicated data on a service is not consistent with the data on the original service. One example of a consistency model oriented towards the execution of transactions is Transactional Causal Consistency, which is defined as follows:

Transactional Causal Consistency [19] is a model similar to Causal Consistency, where transactions read from a causally consistent snapshot, which includes all causal dependencies of the objects read. This model extends Causal Consistency by adding the notion of causal relations between transactions based on the objects they access. Another feature it provides is that the updates of a transaction can be seen as if they are executed atomically (all of them are executed or none of them are executed).

2.9 Tools

In this section, we will present several tools with implementations and objectives similar to the ones of our tool. In Section 2.9.1, we describe tools whose goal is to give an estimate of how complex the decomposition of a monolith will be, considering patterns that may raise anomalies after the migration. In Section 2.9.2, we cover tools that use a black box approach (do not require information on how the system is built) to detect anomalies. In Section 2.9.3, we cover tools that use a white box approach (need to know how the system is implemented) to detect anomalies. In Section 2.9.4, we present a table comparing all the tools presented and ours (MAD).

2.9.1 Heuristics for Anomaly Awareness

2.9.1.A A Complexity Metric for Microservices Architecture Migration

"A Complexity Metric for Microservices Architecture Migration" (*CMMAM*) [20] is a work with two contributions. These are an estimate of the effort required to migrate from a monolith to microservices and the impact of the similarity measure chosen for the decomposition of the monolith. In this work, a microservice is defined as a group of entities called a *cluster*, and to aggregate the entities into *clusters*, a *similarity measure* is used. The paper defines a *similarity measure* as a criterion that quantifies how coupled two entities are and presents several *similarity measures*, each accounting for how frequently a certain event is common to both entities and the obtained frequency represents how related the entities are. The complexity metric takes into account certain patterns in the code, which are likely to raise anomalies after the migration to microservices.

For the first contribution, the paper defines the complexity of the decomposition as the average of the functionalities' complexities. The complexity of a functionality is the sum of the complexities of the clusters accessed by a sequence of operations. The complexity of accessing a cluster is the number of elements in the union of the complexities of the accessed entities inside the cluster. Finally, the complexity of accessed entities depends on the access mode. The complexity of an entity read is the number of other functionalities that access more than one cluster and write to the same entity, and the complexity of an entity written is the number of other functionalities that access more than one cluster and read the same entity.

For the second contribution, four different similarity measures are used to aggregate the entities into clusters. The first similarity measure is given by the number of functionalities that access both entities divided by the number of functionalities that access the first entity. The second similarity measure is given by the number of functionalities that read both entities divided by the number of functionalities that read the first entity. The third similarity measure is given by the number of functionalities that write on both entities divided by the number of functionalities that write on the first entity. The last similarity measure is given by the number of times that both entities are accessed consecutively in all functionalities divided by the maximum number of consecutive accesses.

The first contribution is more related to our work since their complexity estimate is based on the occurrences of certain patterns in a monolith that can possibly lead to anomalies after a decomposition.

2.9.1.B Mono2Micro - From a Monolith to Microservices: MetricsRefinement

"Mono2Micro - From a Monolith to Microservices: MetricsRefinement" [21] is a research work with the intent of improving the accuracy of the metric presented in "A Complexity Metric for Microservices Architecture Migration" [20] by providing the user a set of operations to manipulate a monolith's decomposition. The paper's goal is to offer a more realistic scenario of a migration to microservices considering the *Saga Pattern* [10], local transactions, which are transactions that can commit only by executing on one machine, and semantic locks, application-level locks that indicate when the local transaction of a saga already wrote on a given entity.

The first operation is *Sequence Change*, which receives a pair of transactions, representing a local transaction remotely invoking another transaction, and changes the invoking transaction to a different transaction that happens before the invoked transaction. The second operation is *Local Transaction Merge*, which given two local transactions that either execute sequentially or execute both after a common local transaction, allows the user to merge these two local transactions. The third operation is *Add Compensating*, which, inspired by Sagas [10], allows the user to create a local transaction that will act as a compensating transaction. The last operation is *Define Coarse-Grained Interactions*, which is composed of Sequence Change and Local Transaction Merge. This operation considers as input two remote invocations with four different local transactions, T_1, T_2, T_3, T_4. T_1 and T_2 are in the same cluster. T_3 and T_4 are in the same cluster, but a cluster different than the one from T_1 and T_2. T_1 happens before T_2, and T_3 happens before T_4. The remote invocations are (T_1, T_3) and (T_2, T_4), meaning that T_1 calls T_3 and T_2 calls T_4, respectively. Based on this information, the operation allows the user to simultaneously reorder the two remote invocations to generate remote invocations that can be merged, (T_1, T_2) and (T_3, T_4), finally resulting in one remote invocation with the two merged local transactions (T_{12}, T_{34}).

This paper also introduces three complexity metrics. First, the *Functionality Redesign Complexity*, which is the sum of the complexities of the local transactions in the functionality. The complexity of a local transaction is given by the number of semantic locks in the entities accessed by the local transaction added to the number of other functionalities that write, or have semantic locks, in the same entities that the local transaction accesses. Second, the *System Added Complexity*, which is given by the number of functionalities that read from at least two clusters that are modified by another functionality. Third, the *Query Inconsistency Complexity*, which is only briefly mentioned and consists of the complexity of a query (transaction with only reads) being given by the number of other functionalities that write to one or more clusters read by the query.

2.9.2 Anomaly Detection using Black Box Approaches

2.9.2.A Cobra

Cobra [22] is a tool designed to test if a transactional key-value store in the cloud respects serializability. This tool focuses on the constraints of serializability, to encode the problem in a way that it can be solved by an SMT solver. It uses a black box testing approach, since it suits the tool's target environment, due to, most of the time, applications running in the cloud being considered black boxes.

The tool is composed of three components, the *Clients*, the *History Collectors*, and the *Verifier*. The *Clients* are meant to mimic the actions of real clients by randomly sending requests to the database under test and receiving the respective responses. The *History Collectors* are responsible for logging all the operations made and the responses received in the interaction between the *Clients* and the database. Each *History Collector* logs the operations made and responses received by a specific *Client* and creates a *fragment of history*, which is a sequence of the operations and responses it saw. All the fragments together result in a *history*, which is a sequence of all the operations and responses that occurred in the system. The *Verifier* assembles the *fragments of history* to create the *history* and verifies if it is serializable.

Focusing on the verification, the *Verifier* creates a serialization graph from the *history*, where the transactions are the vertices and the dependencies are the edges. The tool also considers *constraints*, which the paper defines as a set of possible dependencies, represented as a pair of edges. For the tool to provide a faster response, the paper also defines three techniques to reduce the solution space. First, *Combining Writes*, which consists in only considering one of the writes to create an outwards edge, if there are two or more consecutive transactions with writes to the same object. Second, *Coalescing Constraints*, this is done by considering all the reads after the same write as one read. Third, *Pruning Constraints*, this technique consists in decreasing the number of possible combinations by discarding the edges of a constraint that generate cycles in the graph, as these executions are impossible to reach by the constraint definition. After applying all these techniques, *Cobra* uses the resultant graph, the constraints, encoded using logic notation, and the SMT solver (in this case MonoSAT [23]), to verify if the graph is acyclic since this would mean that the execution is serializable.

2.9.2.B MonkeyDB

MonkeyDB [24] is a system designed for users to test their program against multiple consistency levels of a database by simulating the behaviour of a normal storage system.

This tool provides APIs for SQL and key-value store applications, both commonly used by developers to interact with their storage system. One of the components of this system is the *history*, which aggregates the dependency relations between operations, as well as all the operations that are executed.

Another component is the *consistency checker*, which uses logical constraints (axioms) to represent the properties of the consistency models and is responsible for verifying if the execution is valid under the consistency model chosen by the user.

To use the tool, first, the user needs to define a program to test. This program will consist of two or more *sessions*, which the paper defines as sets of transactions that can be executed in parallel. After that, *MonkeyDB* will simulate the executions of these sessions running in parallel and will log each operation executed in the *history*. If the operation is a read, it will compute the possible returned values, based on the *history* and the consistency model used in the *consistency checker*, randomly returning one of the possible values. Note that, by using this procedure, the user can never be sure that the system does not have an anomaly. They can only be more or less confident depending on how many times they run the tool with the test program and the outputs they observe.

2.9.3 Anomaly Detection using White Box Approaches

2.9.3.A Automated Detection of Serializability Violations under Weak Consistency

"Automated Detection of Serializability Violations under Weak Consistency" (*ANODE*) [25] presents a tool that statically analyses SQL-like programs, to check if they contain serializability anomalies under bounded abstract executions and different weak consistency models.

As the first step to use this tool, the user gives as input their SQL program with some small adaptations to fit the syntax of the tool. Then, the tool generates abstract executions of the program by creating instances of the transactions in the program and simulating how they would execute. The next step is converting each abstract execution to a dependency graph. To replicate the behaviour of consistency models, *ANODE* uses logical constraints (axioms) so that the abstract execution is compliant with the chosen consistency model properties. The tool also assumes that each dependency graph will have a cycle. After that, the tool creates a *First Order Logic* (FOL) formula with three clauses, encoding the program characteristics and the consistency model, the dependencies between transactions, and the length of the cycle that is assumed to exist. Finally, the tool uses an SMT solver (Z3 [4]) to check if the FOL formula is unsatisfiable (there is no possible assignment that would respect all the constraints). In this case, unsatisfiability means that the cycle assumed, or any cycle of a smaller length, cannot exist in the dependency graphs. Therefore, all the executions are serializable, unless they have a dependencies cycle of a bigger length.

2.9.3.B CLOTHO

CLOTHO [5] is based on the work of Nagar and Jagannathan [25] and is also a tool designed to analyse the transactions of a storage system that provides weak consistency semantics, to detect if there are

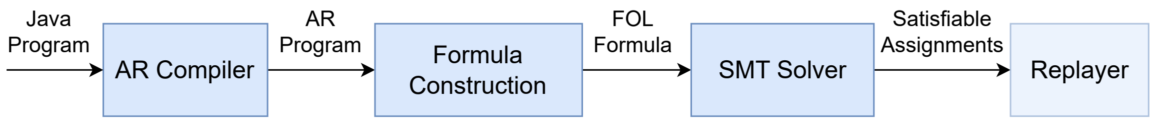


Figure 2.4: *CLOTHO*'s pipeline.

any serializability anomalies. The tool receives as input a Java program with a single class, containing a set of methods representing the system's transactions, and returns as output the number of anomalies found, as well as concrete tests to reproduce the anomalous executions. Regarding the architecture, it is composed of two main components, the *analyzer* and the *replayer*. The former is responsible for doing the analysis of the program and detecting anomalies, and the latter is used to generate an environment that allows the user to run concrete executions that will lead to the anomalies detected.

The way *CLOTHO*'s analysis works consists of creating FOL formulas where the satisfiable assignments correspond to cyclic graphs, whose vertices are the operations and the edges are the relations between the operations. The types of edges considered are: *ST* (same transaction); *RW* (read followed by a write); *WR* (write followed by a read); and *WW* (write followed by a write). The cyclic anomalous graphs are defined as having at least one *ST* edge and at least two dependency edges (*RW*, *WR*, *WW*). These graphs represent non-serializable executions of the transactions, therefore anomalies [26]. *CLOTHO*'s pipeline can be split into three steps for the analysis and one last step for the anomaly concrete execution, as illustrated in Figure 2.4.

First, *CLOTHO* converts the Java program to an *Abstract Representation* (AR), which resembles an SQL-like language and is described in *CLOTHO*'s paper. This step eases the extraction of information from the input program, such as the edges between operations that can access the same object.

Second, using the program in the AR, the goal is to construct a FOL formula. This formula is the conjunction of five sets of constraints. The objective of these sets of constraints is to represent as accurately as possible the environment where the transactions will run, the relationship between them, and the characteristics of the anomalies the user is looking for. The paper represents the sets of constraints using $\varphi_{context}$, φ_{db}, $\varphi_{dep\rightarrow}$, $\varphi_{\rightarrow dep}$ and $\varphi_{anomaly}$. $\varphi_{context}$ represents the values that are plausible to be in the database. φ_{db} represents the consistency and isolation guarantees provided by the database. $\varphi_{dep\rightarrow}$ represents the dependency arrows between operations that belong to the dependency cycle. $\varphi_{\rightarrow dep}$ represents the dependency arrows between operations that do not belong to the dependency cycle. $\varphi_{anomaly}$ bounds the possible anomaly structures to a maximum number of transactions in a serial execution, a maximum number of transactions in a concurrent execution, and a maximum length for a dependency cycle.

Third, after having the FOL formula built, the tool uses an SMT solver (in *CLOTHO*'s case Z3 [4]), to compute the assignments that satisfy the formula, each of them representing an abstract execution that contains an anomaly. In the cases where the formula is unsatisfiable, this means that no anomalies

Table 2.1: Characterization of existing tools.

	Properties			Techniques		
	Analysis	Consistency Model	Microservices Oriented	Method	SMT Solver	Executions Analysed
CMMAM	Heuristic	Eventual Consistency	Yes	Static Analysis	No	-
Metrics Refinement	Heuristic	Eventual Consistency	Yes	Static Analysis	No	Abstract
Cobra	Black Box	Serializability	No	Testing	Yes	Abstract
MonkeyDB	Black Box	Any	No	Testing	No	Concrete
ANODE	White Box	Any	No	Static Analysis	Yes	Abstract
CLOTHO	White Box	Any	No	Static Analysis	Yes	Abstract/ Concrete
CLOTHO+	White Box	Any	Partial	Static Analysis	Yes	Abstract
MAD	White Box	Any	Yes	Static Analysis	Yes	Abstract

were detected within the bounds defined for the formula construction.

Finally, one can opt to use the replayer to simulate a concrete execution that will make the anomaly emerge, however, this aspect falls outside the scope of our work.

2.9.3.C Microservice Decomposition for Transactional Causal Consistent Platforms

"Microservice Decomposition for Transactional Causal Consistent Platforms" (*CLOTHO+*) [6] is a work strongly related to ours since it also focuses on the microservices architecture.

Besides providing relevant insights regarding the decomposition into microservices, it also addressed how prepared *CLOTHO* is to face microservices architectures and introduced useful features to the tool. The first addition was the implementation of other consistency models in *CLOTHO*, since, although they were formalized in *CLOTHO*'s paper [5], *CLOTHO* only had eventual consistency implemented. The second addition was a label to differentiate the transactions that were running in one microservice from the transactions running in another microservice. This is used to simulate microservices running on different machines and prevent false positives in cases where the two conflicting transactions would be running on the same machine. The last addition was a relation to represent that two operations belong to the same class/type of transaction.

2.9.4 Comparison

Table 2.1 presents a comparison of the tools. We split the table into two sections. The first section (Properties) refers to the characteristics of the tools. We divided this section into three columns which capture, respectively, the approach used by the tool to analyse a system (Analysis), the consistency guarantees that the tool is expecting the system under test to respect, considering the ones the tool can be extended to support (Consistency Model), and if the tool is oriented to analyse microservices architectures (Microservices Oriented). The second section (Techniques) refers to the mechanisms used by the tools to fulfil their purpose. We divided this section also in three columns which capture, respectively, how the tool identifies the existence of anomalies (Method), if the tool uses an SMT solver (SMT Solver), and which type of executions the tool analyses (Executions Analysed) ("-" means that the tool does not consider different executions of the system). Note that our tool, MAD, offers a white box analysis applicable to any consistency model and that takes into account all the aspects of the microservices architecture, which is something none of the other tools do.

Summary

In this chapter, we introduced background concepts associated with our work. In particular, the monolithic and microservices architectures, what are transactions and their properties, what are anomalies, how concurrency control mechanisms work, how one can divide transactions into several sub-transactions, how distributed databases and data replication can be achieved, and how transactions operate in a microservices architecture. We have also surveyed other tools with similar purposes and that use adequate techniques for our goals. In the end, we explained how our tool (MAD) differs from the remaining tools by offering an analysis that cannot be done by any of the other tools.

3

MAD

Contents

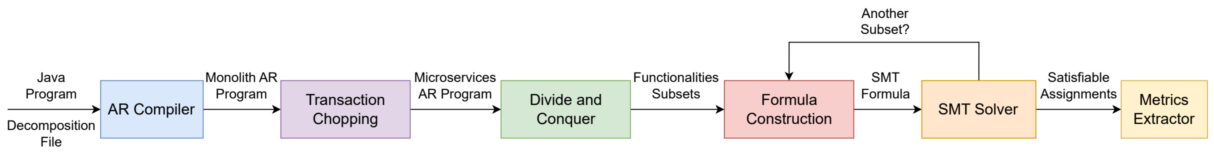


Figure 3.1: MAD's pipeline.

In this chapter, we describe the design and implementation of the *Microservices Anomaly Detector* (MAD) tool, which automatically detects anomalies that may occur when applying a given decomposition from monolith to microservices. We focused our efforts on developing a precise tool that avoided false negatives and false positives while considering all the aspects related to the migration from monolith to microservices. In Section 3.1, we present an overview of MAD's design. In Section 3.2, we explain the implementation of MAD. Section 3.3 presents the adaptations that we had to make to the analysis previously made by *CLOTHO*. Finally, in Section 3.4, we discuss more details about MAD's development.

3.1 Overview

MAD takes as input the source code of a monolithic implementation of an application and a high-level description of how the monolith is decomposed into multiple microservices. In the current version, the source code must be a *Java program* written using the JDBC syntax (uses SQL queries to access the entities, which are maintained by the application in a database), although it is possible to extend the tool to support other additional programming languages in the future. The decomposition of the monolith is expressed as the clustering of the domain entities into aggregates [7] (entities grouped in the same cluster are assumed to be managed by the same microservice) and is represented by a JSON file (*Decomposition File*). Using this input, MAD executes the pipeline presented in Figure 3.1. We will now explain each of the steps of the pipeline.

During the compilation to the AR, MAD performs two steps. First, the *AR Compiler*, which extracts the monolith's transactions (designated by us as original transactions), parameters, and expressions, originating the *Monolith AR Program*. Second, the *Transaction Chopping* where the representations of the sub-transactions are generated based on the original transactions and the decomposition considered, originating the *Microservices AR Program*.

Often, the *AR* program is too complex to be represented in a single encoding that can be easily analysed. Employing a divide and conquer strategy, MAD generates subsets of the original transactions (*Functionalities Subsets*). Analysing these subsets independently allows for the analysis of the whole problem to be simpler and done in a reasonable time. Therefore, instead of creating a single SMT formula with all the assertions, MAD creates several formulas, one for each subset, with each formula only having the assertions related to the original transactions of their given subset.

When constructing the formulas, MAD uses already existing assertions to encode the basic behaviour of a distributed system and the logic to detect cyclic graphs (executions with anomalies). However, since those were not enough to model the context of our problem, we had to add and adapt assertions of the formula. To illustrate that in a graph there is a relation between two operations of the same original transaction that are in different sub-transactions, we consider a new edge type, *SOT*. To model where each operation would execute and the operations' visibility effects, we add the notion of microservices together with the usage of consistency models assertions.

After the SMT solver (Z3 [4]) finishes its analysis, we obtain the *Satisfiable Assignments*, which represent the anomalies found. To present more metrics, MAD has a *Metrics Extractor*, which applies a pattern-matching technique to enumerate the number of anomalies of each type, groups the anomalies by sets of sub-transactions considering the sub-transactions involved in each anomaly, and counts the number of occurrences of each sub-transaction in the anomalies.

3.2 Architecture

In this section, we address how each of the different aspects of MAD's design are implemented in the tool. More specifically, the input files, the notions of sub-transactions and microservices, the consistency models, the search algorithm, and the metrics extractor.

3.2.1 Input Files

MAD receives as input a Java file with all the original transactions and their respective operations using JDBC and SQL queries to access the entities, and a JSON file that represents the decomposition and indicates the existent microservices and the entities assigned to each microservice. Considering the example from Figure 2.2, its respective Java and JSON files can be seen in Listing 3.1 and Listing 3.2, respectively.

3.2.2 Sub-transactions and Microservices Notions

To represent the migration, two essential aspects need to be considered. These aspects are: (1) sub-transactions, transactions that originated from the division of a transaction, and (2) microservices, the nodes where the sub-transactions will execute. We divided the task of modeling these aspects into three steps: 1) representing the sub-transactions; 2) introducing a feature to indicate that two sub-transactions are related since they will be executed under the same functionality (original transaction); 3) introducing the concept of microservice and associating it with the operations.

Listing 3.1: Example of MAD's input Java file representing the example scenario.

```java
public class exampleScenario {
    private Connection connect = null;
    private int _ISOLATION = Connection.TRANSACTION_READ_COMMITTED;
    private int id;
    Properties p;

    public exampleScenario(int id) {
        this.id = id;
        p = new Properties();
        p.setProperty("id", String.valueOf(this.id));
        Object o;
        try {
            o = Class.forName("MyDriver").newInstance();
            DriverManager.registerDriver((Driver) o);
            Driver driver = DriverManager.getDriver("jdbc:mydriver://");
            connect = driver.connect("", p);
        } catch (InstantiationException | IllegalAccessException |
                ClassNotFoundException | SQLException e) {
            e.printStackTrace();
        }
    }

    public void Txn1(int memberId, int itemId, int newStatus) throws SQLException {
        PreparedStatement stmt1 = connect.prepareStatement("SELECT status " + "FROM " +
                                                            "MEMBER" + " WHERE id = ?");
        stmt1.setInt(1, memberId);
        ResultSet rs = stmt1.executeQuery();
        rs.next();
        int read_status = rs.getInt("status");

        PreparedStatement stmt2 = connect.prepareStatement("SELECT price " + "FROM " +
                                                            "ITEM" + " WHERE id = ?");
        stmt2.setInt(1, itemId);
        ResultSet rs2 = stmt2.executeQuery();
        rs2.next();
        int read_price = rs2.getInt("price");

        PreparedStatement stmt3 = connect.prepareStatement("UPDATE MEMBER SET status = ?" +
                                                            " WHERE id = ?");
        stmt3.setInt(1, newStatus);
        stmt3.setInt(2, memberId);
        stmt3.executeUpdate();
    }

    public void Txn2(int memberId, int newStatus, int itemId, int newPrice) throws SQLException {
        PreparedStatement stmt1 = connect.prepareStatement("SELECT status " + "FROM " +
                                                            "MEMBER" + " WHERE id = ?");
        stmt1.setInt(1, memberId);
        ResultSet rs = stmt1.executeQuery();
        rs.next();
        int read_price = rs.getInt("status");

        PreparedStatement stmt2 = connect.prepareStatement("UPDATE MEMBER SET status = ?" +
                                                            " WHERE id = ?");
        stmt2.setInt(1, newStatus);
        stmt2.setInt(2, memberId);
        stmt2.executeUpdate();

        PreparedStatement stmt3 = connect.prepareStatement("UPDATE ITEM SET price = ?" +
                                                            " WHERE id = ?");
        stmt3.setInt(1, newPrice);
        stmt3.setInt(2, itemId);
        stmt3.executeUpdate();
    }
}
```

Listing 3.2: Example of MAD's input JSON file representing a microservices decomposition.

```
{
    "M1": [
        "Member"
    ],
    "M2": [
        "Item"
    ]
}
```

3.2.2.A Sub-transactions Representation

Upon compiling the *Java program* to the *Monolith AR Program*, MAD proceeds to generate the representation of the sub-transactions considering the assignments between entities and microservices from the *Decomposition File*. For each original transaction, MAD applies the procedure described in Algorithm 3.1.

The rationale behind Algorithm 3.1 is to iterate over the operations of an original transaction (*for* loop in Line 9) and generate sub-transactions based on the entities that are accessed by the operations. In each iteration, a verification is performed (*if else* in Lines 11 and 19) to assess if the entity accessed by the iteration's operation (*entityName*) belongs to a different microservice from the previous operation's entity (*currentMicroservice*), or not. If the entity belongs to a different microservice (Line 11), then the current microservice is updated and a new sub-transaction is created with the seen operation since it would execute in a different microservice. Else (Line 19), the operation is added to the most recent sub-transaction created (*subTxns.get(currentSubTransactionIdx)*) since it would execute in the same microservice. The output from applying this algorithm to the original transactions from the example in Figure 2.2 originates a representation equivalent to the right side of the figure (Microservices side).

3.2.2.B Same Original Transaction Edge

In the migration to microservices, some transactions may need to be chopped, which originates sub-transactions. Although these sub-transactions can be seen as independent transactions, they are still related to each other in the sense that they are part of the same functionality (original transaction) and they need to be executed sequentially, to provide the same behaviour as the original transaction. Therefore, to represent the relation between operations from different sub-transactions that belong to the same original transaction, we introduce an edge type, *SOT* (same original transaction).

To accomplish this, we add to the SMT formula a sort (*OT*), a data type (*OTType*), two formula functions (*ottype*, which receives an instance of an original transaction and returns its type, *OTType*;

Algorithm 3.1: MAD's Transaction Chopping

```
Input: body: decompiled Java body of the original transaction function, entitiesMicrosMap:
       mapping between entities and microservices, tables: list of input program's tables
Output: origTxn: representation of the original transaction
1  Function MADTransactionChopping(body, entitiesMicrosMap, tables)
2      name ← body.getMethod().getName()
3      origTxn ← OriginalTransaction(name)
4      unitHandler ← UnitHandler(body, tables)
5      unitHandler.extractParams()
6      currentSubTransactionIdx ← -1
7      currentMicroservice ← “”
8      subTxns ← $\emptyset$
       // Iterate over all the statements (operations)
9      foreach s in unitHandler.data.getStmts() do
10         entityName ← ((InvokeStmt) s).getQuery().getTable().getName()
           /* Create a new sub-transaction when the microservice of the operation's
              entity is different from the current sub-transaction's microservice   */
11         if currentMicroservice $\neq$ entitiesMicrosMap.get(entityName) then
12             currentMicroservice ← entitiesMicrosMap.get(entityName)
13             currentSubTransactionIdx ← currentSubTransactionIdx + 1
14             newSubTxn ← Transaction(name + “_” + currentSubTransactionIdx)
15             newSubTxn.setOriginalTransaction(name)
16             newSubTxn.setMicroservice(currentMicroservice)
17             newSubTxn.addStmt(s)
18             subTxns ← subTxns $\cup$ newSubTxn
           /* Add the operation to the current sub-transaction when the microservice
              of the operation's entity is the same                                  */
19         else
20             subTxns.get(currentSubTransactionIdx).addStmt(s)
21         origTxn.addStmt(s)
```

and *original_transaction*, which receives an instance of an operation and returns the instance of original transaction where it belongs, *OT*) and one predicate (*step_sibling*, which receives two instances of operations and returns true if they are related as operations of different sub-transactions but the same instance of an original transaction). The *ottype* function values are initialized at the beginning of the formula by expressing that for all the operations of a given operation type (*OType*), the instance of original transaction they belong to has to be of a specific *OTType*. The assertion can be found in Equation (3.1) with *otype* being a function that receives an instance of an operation and returns its type, *OType*. The *original_transaction* function is involved in the establishment of dependency edges and has its values assigned during the analysis when the SMT solver is looking for satisfiable assignments.

$$\forall o1((otype(o1) = op1) \Rightarrow (ottype(original\_transaction(o1)) = orig\_txn1)) \tag{3.1}$$

When the *SOT* edge is established between two operations, it indicates that the operations are in different sub-transactions but belong to the same original transaction, which allows MAD to understand that the operations will always follow a specific order and that an interleaving between the two operations may lead to an anomaly.

Considering this new edge type, our definition of a cyclic anomalous graph is a cycle with at least one *ST* or *SOT* edge and at least two dependency edges (*RW*, *WR*, *WW*). In Equation (3.2) and Equation (3.3), we present the anomalous cycles definition for lengths three and four, respectively. For these definitions, we use two predicates besides the edge types: *D*, which receives two instances of operations and returns true if there is any dependency relation between them (*RW*, *WR* or *WW*); and *Any*, which also receives two instances of operations and returns true if there is any relation between them (*ST*, *SOT*, *RW*, *WR* or *WW*).

$$\forall o1, o2, o3((ST(o1, o2) \vee SOT(o1, o2)) \wedge D(o2, o3) \wedge D(o3, o1)) \tag{3.2}$$

$$\forall o1, o2, o3, o4((ST(o1, o2) \vee SOT(o1, o2)) \wedge D(o2, o3) \wedge Any(o3, o4) \wedge D(o4, o1)) \tag{3.3}$$

3.2.2.C Microservices Assignment

Regarding the microservices notion, we defined a new data type (*MType*) and a new formula function (*mtype*, which receives an instance of an operation and returns the microservice it executes on, *MType*) in the SMT formula. These are used to associate the operations with the microservice where they will be executed. The *mtype* function values, similar to the *ottype* function, are initialized at the beginning of the formula by expressing that for all the operations of a given operation type (*OType*), their microservice has to be of a specific *MType*. The assertion can be found in Equation (3.4).

$$\forall o1((otype(o1) = op1) \Rightarrow (mtype(o1) = M1)) \tag{3.4}$$

These new features are then used to restrict the consistency models and to control the visibility between operations, as we will later discuss in Sections 3.2.3 and 3.3.1.

3.2.2.D Anomaly Example Graph

Consider the example anomaly presented in Figure 2.3. MAD detects this anomaly by finding the cyclic graph that can be seen in Figure 3.2. The cycle contains two *SOT* edges and two dependency edges, more concretely, one *RW* edge and one *WR* edge, and represents the interleaving of the original transactions *Txn1* and *Txn2*.

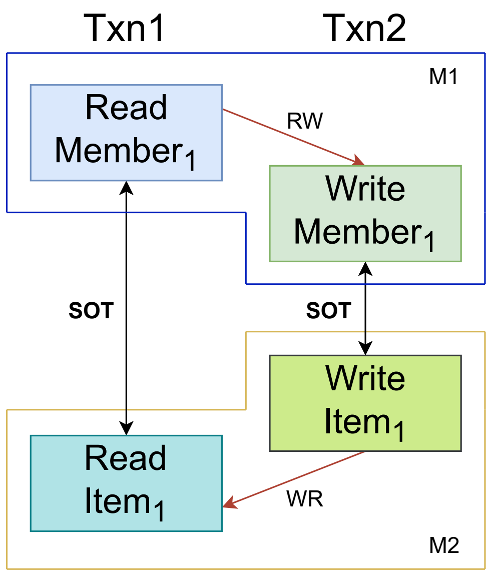


Figure 3.2: MAD's cyclic graph for the example anomaly.

3.2.3 Consistency Models

To make an analysis faithful to the environment where the microservice systems will execute, we assume two consistency models: Serializability and Eventual Consistency. Between operations of the same sub-transaction and other operations of the same microservice, we assume that they will respect Serializability. Between operations of different sub-transactions or that execute on different microservices, we assume Eventual Consistency [1]. By default, MAD enforces Eventual Consistency between the visibility effects of the operations. However, to enforce Serializability, we use the consistency models' assertions defined in *CLOTHO*'s paper [5] and implemented in *CLOTHO+* [6] with some adjustments so that Serializability is only applicable between operations that execute in the same microservice.

The consistency models' assertions rely on two relevant predicates, *vis* and *ar*. *vis* receives two instances of operations and returns true if the effects of the left operation are visible to the right operation. *ar* also receives two instances of operations and returns true if the left operation is executed before the right operation. The assertions are implemented as follows (our adjustments are highlighted in blue):

[1]https://martinfowler.com/articles/microservice-trade-offs.html

$$Read\_Committed = \forall o1, o2, o3(ST(o1, o2) \land vis(o1, o3) \land (mtype(o1) = mtype(o3)) \Rightarrow vis(o2, o3)) \tag{3.5}$$

$$Repeatable\_Read = \forall o1, o2, o3(ST(o1, o2) \land vis(o3, o1) \land (mtype(o1) = mtype(o3)) \Rightarrow vis(o3, o2)) \tag{3.6}$$

$$Linearizability = \forall o1, o2(ar(o1, o2) \land (mtype(o1) = mtype(o2)) \Rightarrow vis(o1, o2)) \tag{3.7}$$

$$Serializability = Read\_Committed \land Repeatable\_Read \land Linearizability \tag{3.8}$$

3.2.4 Search Algorithm

MAD's search algorithm follows a *Divide and Conquer* principle. Instead of taking into account all the original transactions of the system at the same time, MAD restricts the search for cycles to only consider a subset of the original transactions per iteration. By using this method, we can divide a large problem into several smaller problems, which can be solved in a much shorter period of time. This search algorithm can also be applied to other cases, such as *CLOTHO*'s analysis, by adapting the algorithm to use the transactions (in MAD designated as sub-transactions), instead of the original transactions.

3.2.4.A Original Transactions Combinations

The first step of the search algorithm is to generate all the combinations of size smaller than the maximum cycle length (in our case four, as we will later justify in Section 4.1). The number of combinations that will be generated when there are at least three original transactions can be obtained by replacing the value of n in $\sum_{x=1}^{3} {}^{n}C_{x}$ with the number of original transactions.

After generating the combinations, we proceed to restrict the original transactions of the operations that we are considering for the cycle in the SMT formula to be from the combination that is being assumed at that iteration. The anomalies can have at most the current cycle length minus 1 ($current\_cycle\_length - 1$) original transactions involved because they must have at least one ST or SOT edge, which implies that at least two operations share the same original transaction. Therefore, MAD only iterates over the combinations of size $current\_cycle\_length - 1$ in each cycle length analysis. As an example, if we have five original transactions, *order*, *sell*, *buy*, *look* and *update*, assume that the current cycle length is 3 and the combination [*order*, *sell*], then, complementing the cycle assertions from Equation (3.2), we would have the assertions defined in Equation (3.9) (for simplicity, we only present the assertions related with o1, since for o2 and o3 they would be the same only changing the operation instance used as argument).

$$\forall o1, o2, o3(((ottype(original\_transaction(o1)) = order) \vee (ottype(original\_transaction(o1)) = sell))\wedge$$
$$(\neg(ottype(original\_transaction(o1)) = buy)) \wedge (\neg(ottype(original\_transaction(o1)) = look))\wedge$$
$$(\neg(ottype(original\_transaction(o1)) = update)) \wedge ...) \quad (3.9)$$

3.2.4.B Original Transactions Assertions Filtering

Besides restricting each of the cycle's operations to belong to one of the original transactions of the combination considered, we also restricted the assertions that are added to the formula, so that it only includes the assertions related to the original transactions of the combination. This procedure minimizes the overall complexity of the SMT formula since it will only have the essential assertions for the analysis. By doing this, the search space is reduced and subsequently, the analysis of each combination is done faster, allowing MAD to analyse a system in a shorter period of time.

3.2.5 Metrics Extractor

Besides presenting the total number of anomalies found, MAD also displays two additional metrics. These metrics are the number of anomalies by type of anomaly and the number of anomalies considering the sub-transactions that are involved in each anomaly.

3.2.5.A Number of Anomalies by Type

The first additional metric is the number of anomalies by type of anomaly (dirty read, dirty write, lost update, write skew, and read skew). The technique we use consists of having a set of patterns only considering the types of edges between operations and, when an anomaly is detected, checking if the anomaly cycle matches any of the patterns. In the cases where the anomaly execution is more complex than our anomaly types' patterns and the cycle found does not match any of the patterns, then the anomaly is not classified and is considered "Unclassified".

The output format for this metric can be seen in Listing 3.3. The *Write Skews* are together with the *Lost Updates* since the write skew pattern also corresponds to one of the lost update patterns. The only way to distinguish them would be to also consider the tables and rows accessed. If the same row was being accessed, then it would be a *Lost Update*, otherwise, it would be a *Write Skew*. However, the patterns we use in our approach only consider the graph cycle edges, therefore not having enough information to make this distinction. To do so, we had to add the notion of row accessed to our patterns, which would also require the additional task of extracting the row that each of the anomaly's operations accessed to do the pattern-matching.

Listing 3.3: Number of anomalies by type format.

```
+++ Dirty Reads found:                <#dirty_read_anmls>
+++ Dirty Writes found:               <#dirty_write_anmls>
+++ Lost Updates found:               <#lost_update1_anmls>
+++ Lost Updates/Write Skews found: <#lost_update2/write_skew_anmls>
+++ Read Skews found:                 <#read_skews_anmls>
+++ Unclassified found:               <#unclassified_anmls>
```

3.2.5.B Number of Anomalies by Sub-Transactions

Regarding the second additional metric, it focuses on presenting two aspects. First, the number of anomalies that occur when the exact same set of sub-transactions is involved. This is used to capture the impact caused by the interaction between certain sub-transactions towards the number of anomalies. Second, the number of occurrences of each sub-transaction in the anomalies. This aspect helps to understand how relevant each sub-transaction is regarding the existence of anomalies in the system.

These results are returned as output following two formats, which are, respectively, presented in Listing 3.4.

Listing 3.4: Number of anomalies by sub-transactions formats.

```
[sub_0, sub_1, sub_2]: <#[sub_0, sub_1, sub_2]_anmls>/<#total_anmls>
...
<orig_txn> (sub_0): <#sub_0_anmls>/<#total_anmls>
...
```

3.3 *CLOTHO* Adaptations

Besides the contributions described in the previous sections, we also had to adapt/fix a few assertions that we are using and were developed in *CLOTHO*, since they were not assuming the semantics that suited our context and were incorrectly representing some elements in the SMT formula. There are two adaptations worth mentioning. First, we assume a more fine-grained representation of the operations' visibility, considering the microservices where the operations execute, besides the network partitions. Second, we fixed the formula representation of values read from the database when they are used in instructions of the Java program.

3.3.1 Visibility Adaptation to Microservices

This first adaptation was required due to the fact that we had to restrict an assertion regarding indirect visibility of the operations' effects, whose goal was to enforce that if a write operation happens before a read operation and both happen inside the same partition, then the write operation would always be visible to the read operation. Our change makes this assertion only applicable in the cases where both operations belong to the same microservice. The adapted assertion can be found in Equation (3.10) with our change highlighted in blue. It uses two formula functions and one predicate, besides the ones already mentioned. The two formula functions are *partition*, which receives an instance of an operation and returns the partition where it executes, and *otime*, which also receives an instance of an operation and returns the time when the operation is executed. The predicate is *is_update*, which receives an instance of an operation and returns true if that operation is an update operation (update, insert or delete).

$$\begin{aligned}
&\forall o1, o2((partition(o1) = partition(o2)) \wedge (otime(o2) > otime(o1)) \\
&\wedge\ (is\_update(o1)) \wedge (\neg(is\_update(o2))) \\
&\wedge (mtype(o1) = mtype(o2)) \Rightarrow (vis(o1, o2)))
\end{aligned} \tag{3.10}$$

3.3.2 Correct Version processing for Database Read Values

The second relevant adaptation is a fix regarding the version considered when representing values read from the database that are used in instructions of the Java program. Note that the *next* method that will be mentioned is a Java method that in each call sequentially returns a row from the set of rows that matched the query of a read operation. To better explain the problem, three formula functions need to be introduced:

- **<orig_txn>_r<index>-next<index2> (origTxn)**, which receives an instance of an original transaction, and returns the row read in that instance of <orig_txn> in the <index>th read operation of the program considering the <index2>th call to the *next* method;
- **<table>_VERSION (row, operation)**, which receives a row and an instance of an operation, and returns the version of the row from table <table> seen by the operation;
- **<table>_PROJ_<column> (row, version)**, which receives a row and a version, and returns the value of <column> from table <table> that is expected in that row considering the given version.

Now, we present the snippet in Listing 3.5 as an example scenario to highlight the version problem. In this scenario, there are three operations, o1, o2, and o3, in lines 1, 5, and 7, respectively. The snippet execution consists of reading the price of a row in table "Item" and incrementing it by 1 if it is below 50 or decrementing it by 1 if it is above or equal to 50.

Listing 3.5: Example snippet for the version problem.

```
rs = executeQuery('Select price from Item where id = 1');
q = rs.next();
int p = q.get('price');
if (p < 50)
    executeUpdate('Update Item set price = ' + (p + 1) + ' where id = 1');
else
    executeUpdate('Update Item set price = ' + (p - 1) + ' where id = 1');
```

Having this example in mind, the problem is related to how variable “p” will be represented in operations o2 and o3. Assuming that the snippet is from an original transaction named “order” and there is an instance of that original transaction represented by “ot!1”, “p” representation for o2 and o3 can be seen in Equations 3.11 and 3.12, respectively.

$$Item\_PROJ\_price\ (order\_r1\text{-}next1\ (ot!1),\ Item\_VERSION\ (order\_r1\text{-}next1\ (ot!1),\ o2)) \tag{3.11}$$

$$Item\_PROJ\_price\ (order\_r1\text{-}next1\ (ot!1),\ Item\_VERSION\ (order\_r1\text{-}next1\ (ot!1),\ o3)) \tag{3.12}$$

As we can see, the representation of “p”, the price read from the database, in o2 is different from the one in o3. This occurs since the table’s row version considered is obtained by using the operation instance where the value read from the database is being used, instead of using the table’s row version seen by the read operation. This leads to two problems. First, the analysis considers that both paths can be executed at the same time since the path condition depends on “p”, whose representation is different depending on the operation. Second, by considering the operation instance where the read value is being used to obtain its version, then this leads to a scenario where the updates’ representation uses the same value for the left-hand side and the right-hand side of the update expression. For example, considering o2, the update targets the same row’s value that was previously read, therefore the update operation (price = p + 1) would be translated to “the row’s price in o2’s version equals the row’s price in o2’s version plus 1” (price(r, o2) == price(r, o2) + 1), which is always impossible.

To fix this problem, we introduce a formula function:

- **<orig_txn>_r<index>-next<index2>_READ_VERSION (origTxn)**, which receives an instance of an original transaction, and returns the version that was seen in that instance of <orig_txn> in the <index>th read operation of the program considering the <index2>th call to the *next* method.

With this new formula function, we can guarantee that the representation of a value read from the database does not vary depending on the operation where it is being used, as well as achieve a correct representation when updating a database row column using a value previously read from the database. For example, “p” representation for both operations now would be the one presented in Equation (3.13).

$$Item\_PROJ\_price\ (order\_r1\text{-}next1\ (ot!1),\ order\_r1\text{-}next1\_READ\_VERSION\ (ot!1)) \tag{3.13}$$

3.4 Discussion

To develop MAD, we used as a starting point *CLOTHO* [5] with several consistency models [6]. We chose *CLOTHO* due to its advantages when compared with the other tools and the fact that having access to the source code proves useful for objectives complementary to this work, such as choosing the best decomposition. None of the other tools is capable of doing the analysis we intend, since they do not capture all the aspects related to the migration to microservices. For example, understanding that the monolith’s functionality that executes in a single transaction may correspond to the sequential execution of several sub-transactions when executing in a microservices composition. Besides that, the capability to represent that different sub-transactions may execute in different microservices is required, to simulate the visibility of the operations’ effects considering the microservices where the operations execute. Note that how the decomposition is generated is orthogonal to our work and there are already tools in the literature to help with this step [7–9].

Summary

In this chapter, we presented MAD, the tool we developed to achieve our goal of automatically detecting anomalies when migrating from a monolith to microservices following a given decomposition. We started by presenting an overview of MAD, and then described how each of the aspects considered for the analysis was implemented. We concluded by addressing some adaptations that were needed and discussed MAD’s development.

4 Evaluation

Contents

In this chapter, we present the evaluation that we performed to assess MAD capabilities. In Section 4.1, we describe all the benchmarks that we use and the conditions under which the evaluation is performed. Section 4.2 presents a comparison between MAD and a heuristic approach tool oriented to microservices, “A Complexity Metric for Microservices Architectures Migration” (*CMMAM*) [20], also displayed in Table 2.1. At last, in Section 4.3, we present the results obtained by MAD when applied to real-world codebases.

4.1 Experimental Setup

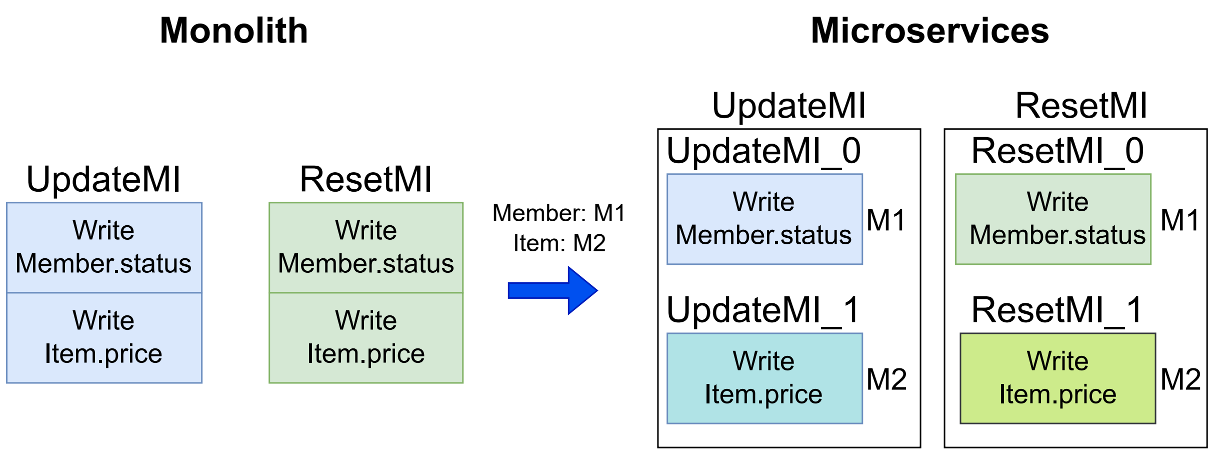
For this experimental evaluation, we use two types of benchmarks: (1) handcrafted benchmarks that we name *microbenchmarks*; (2) real-world codebases that can be found on GitHub.

Regarding the *microbenchmarks*, we created three instances, each with a different set of transactions, however, all assuming the same context. The context consists of an application that has two entities, *Member* and *Item*, with *Member* having three attributes, *id*, *status* and *money*, and *Item* also having three attributes, *id*, *price* and *stock*. All the *microbenchmarks* assume the same decomposition, *Member* executing on microservice *M1* and *Item* executing on microservice *M2*. The purpose of these *microbenchmarks* is to illustrate the differences between MAD and *CMMAM*. The *microbenchmarks* descriptions can be found in Figure 4.1. *Microbenchmark1* (Figure 4.1(a)) targets dependencies between writes (WW) and is composed of two functionalities, *UpdateMI* that updates a member’s status and an item’s price, and *ResetMI* that resets a member’s status and an item’s price (writes 0 on both). *Microbenchmark2* (Figure 4.1(b)) targets accessing different instances of the same entity and is composed of two functionalities, *UItem1* that reads the status of a specific instance of *Member* (Member_1) and updates the price of a specific instance of *Item* (Item_1), and *UMember2* that reads the price of a specific instance of *Item* (Item_2) and updates the status of a specific instance of *Member* (Member_2). At last, *Microbenchmark3* (Figure 4.1(c)) targets accessing different attributes of the same entity and is composed of two functionalities, *UItem* that reads a member’s status and updates an item’s stock, and *UMember* that reads an item’s price and updates a member’s money.

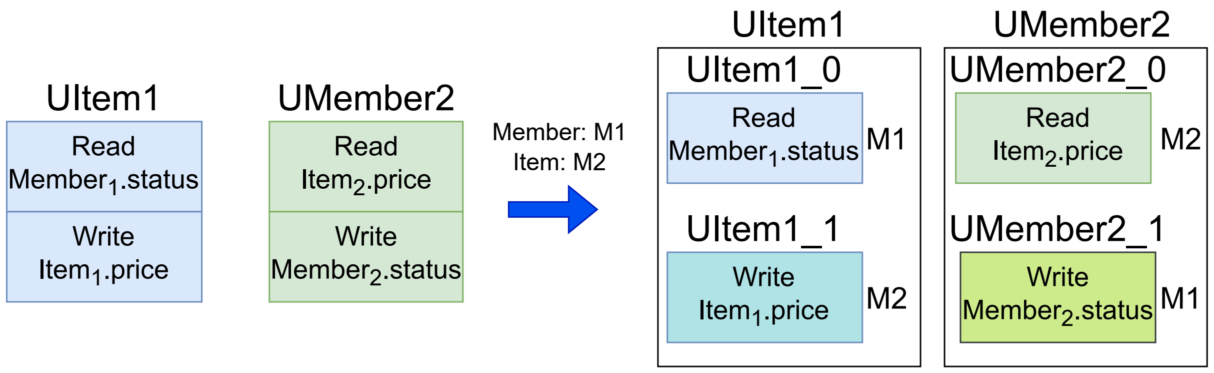
To evaluate MAD’s applicability to real-world scenarios, we gathered several software applications that can be found on GitHub. The list of applications and their respective description is the following:

- **TPC-C** [1] is defined in the OLTP-Bench [27] project and simulates the behaviour of a delivery and warehouse management system;
- **jpabook** [2] simulates a shop where members can order items and track the delivery process;

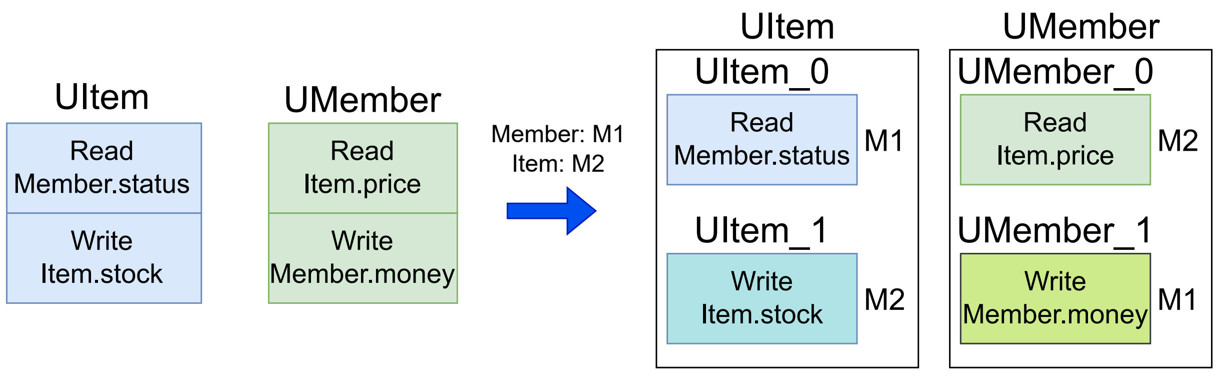
[1] https://github.com/oltpbenchmark/oltpbench/tree/master/src/com/oltpbenchmark/benchmarks/tpcc
[2] https://github.com/holyeye/jpabook/tree/master/ch12-springdata-shop



(a) *Microbenchmark1* (WW dependencies).



(b) *Microbenchmark2* (different instances).



(c) *Microbenchmark3* (different attributes).

Figure 4.1: *Microbenchmarks.*

Table 4.1: MAD and *CMMAM microbenchmarks* results.

	MAD #Anomalies	**CMMAM Complexity**
Microbenchmark1	3	0
Microbenchmark2	0	4
Microbenchmark3	0	4

- **spring-framework-petclinic** (petclinic) [3] simulates how a pet clinic operates regarding the interactions between owners, pets, and veterinarians, as well as the visits to the pet clinic;
- **myweb** [4] is an application that simulates the behaviour of the web allowing users to have roles and perform operations to interact with resources. The operations are create, read, update, and delete (CRUD);
- **spring-mvc-react** (react) [5] is a platform where users can post questions and answers with tags associated with them. Besides that, the system also allows users to upvote or downvote publications, which has an impact on the users' popularity.

For both types of benchmarks, we use MAD considering four as the maximum cycle length. The process to choose this value consisted of starting with value three since it is the minimum number of edges required to detect a cycle with an anomaly, and incrementing it until MAD detected a significant number of anomalies but still within a reasonable period of time. The evaluation was performed in a virtual machine with 32 CPUs and 128GB of RAM using Ubuntu 18.04.4 LTS, Java 8, and version 4.12.3 of Z3.

4.2 Comparison between MAD and a Heuristic Approach Tool

The comparison between MAD and *CMMAM* consists of the analysis of the results that can be obtained when applying each of the tools to the three *microbenchmarks*. The results can be found in Table 4.1.

First, in *Microbenchmark1*, the migration leads to anomalies when instances of the *UpdateMI* and *ResetMI* functionalities execute concurrently, since their interleaving results in an inconsistent database state at the end of the execution. MAD detects three anomalies, whereas *CMMAM* returns that the given decomposition leads to no problems. The anomalies MAD detects are the interleaving between different instances of *UpdateMI*, and instances of *UpdateMI* with instances of *ResetMI*. *CMMAM* does not alert this type of situations, since it only accounts for observable states of the application.

Second, in *Microbenchmark2*, there are no anomalies, because functionalities *UItem1* and *UMember2* always access different instances of the entities, $Member_1$ and $Item_1$, and $Member_2$ and $Item_2$,

[3] https://github.com/spring-petclinic/spring-framework-petclinic
[4] https://github.com/Jdoing/myweb
[5] https://github.com/noveogroup-amorgunov/spring-mvc-react

respectively. Since MAD considers the instances that are being accessed in each operation, it returns that there are zero anomalies. On the other hand, *CMMAM* returns that the migration has complexity four, due to the fact that it only considers the entity that is being accessed and not the instance.

Third, in *Microbenchmark3*, similar to *Microbenchmark2*, there are no anomalies. However, in this case, both functionalities, *UItem* and *UMember*, can access the same instances of the entities. The difference is that the operations are accessing different attributes. Therefore, there are no conflicts between the values that are read and the values that are written. MAD considers the columns/attributes accessed and returns zero anomalies, whereas *CMMAM*, by not considering those aspects, returns complexity four for this migration.

4.3 MAD Results for Real-World Applications

To evaluate MAD's applicability to real-world scenarios, our process can be divided into four steps: 1) gathering five monolithic applications that can be found on GitHub; 2) adapting them to the syntax processed by MAD (adjusted some queries and used JDBC instead of JPA considering each controller method as a transaction); 3) generating two decompositions of each application with the help of a migration tool [7] that supports programmers on the task of grouping the entities by the microservices; 4) using MAD on each decomposition. We assume that the monolithic version of each application is correct and contains no anomalies. Therefore, any anomaly that arises in the microservices versions must have resulted from the migration. Besides the overall results, we also showcase MAD's ability to display the number of anomalies found in each decomposition by anomaly type and set of sub-transactions, together with the number of occurrences of each sub-transaction in the anomalies. We are only considering the analysis of the sub-transactions metrics for one decomposition, but it could be done for the other decompositions as well. At last, we discuss the improvement provided by the search algorithm by comparing MAD's performance without and with it when applied to the same decompositions.

4.3.1 Overall Results

As we previously stated, for this part of the evaluation, we use five monolithic applications (*mono*) with two decompositions to microservices for each application. The microservices decompositions are: *"best"*, which is the decomposition with the highest *Silhouette Score* (a metric used to assess how well the clustering of the entities is done) calculated by the migration tool [7]; and *full*, which is a decomposition where each entity is managed by a different microservice, resulting in the largest functionalities' division possible in a microservices migration, illustrating the worst case scenario in terms of anomalies. In Table 4.2, we present the results from applying MAD to each of the decompositions.

By analysing Table 4.2, we can observe several relevant aspects associated with MAD's analysis of

Table 4.2: MAD overall results.

Application	#Entities	#Functionalities	Decomposition	#Microservices	#Sub-Transactions	#Anomalies	Execution Time [s]
TPC-C	9	5	mono	1	5	0	13
			"best"	2	6	0	13
			full	9	22	98	994
jpabook	5	10	mono	1	10	0	33
			"best"	2	15	70	356
			full	5	21	79	465
petclinic	6	14	mono	1	14	0	77
			"best"	2	14	0	79
			full	6	27	0	119
myweb	5	20	mono	1	20	0	259
			"best"	2	22	0	265
			full	5	32	7	400
react	5	23	mono	1	23	0	571
			"best"	2	28	45	1677
			full	5	39	58	1951

the decompositions. As expected, the number of anomalies found in the *full* decompositions is bigger than in the *"best"* decompositions since they originate more sub-transactions. By looking at the number of anomalies found in each application's decompositions, MAD allows programmers to assess how problematic each decomposition will be, therefore enabling them to make a more informed decision when migrating to microservices. For instance, in *jpabook* and *react*, even though their *"best"* decompositions have the highest *Silhouette Scores* out of their possible decompositions, they still have anomalies. Note that in the *petclinic* application no decomposition led to anomalies. After analysing the application, we noticed that this occurred because the application's operations are mostly reads and the functionalities tend to be short and access few entities.

Although MAD can provide these results with precision, in some cases, it may take a relatively long period of time to perform its analysis, as can be seen from the results in column "Execution Time [s]". This problem is less common in simple applications or when the microservices version of the application does not originate many sub-transactions. However, in complex applications with a high number of functionalities and/or sub-transactions, MAD will tend to take a larger amount of time.

Another aspect one can notice is that all the *"best"* decompositions have two microservices. From our understanding, this phenomenon may be related to Martin Fowler's Strangler Fig pattern [6], where the migration to microservices is done incrementally by extracting a few services at a time, considering the coupling between entities. The *Silhouette Score* might indirectly take this into account, resulting in its value suggesting that from the monolithic implementation, the most appropriate decomposition to microservices is to migrate to two microservices. Moreover, in most cases, the *"best"* decompositions lead to a migration where no anomalies emerge. However, in two applications (*jpabook* and *react*), this does not occur since the *"best"* decompositions for these cases required the functionalities to be more divided, therefore generating more sub-transactions and allowing more interleaving between functionalities.

[6] https://martinfowler.com/bliki/StranglerFigApplication.html

Table 4.3: MAD anomalies found per type.

Application	Decomposition	#Dirty Reads	#Dirty Writes	#Lost Updates	#Lost Updates/ Write Skews	#Read Skews	#Unclassified	#Total
TPC-C	mono							0
	"best"							0
	full		13			27	58	98
jpabook	mono							0
	"best"			3	6	10	51	70
	full			3	7	15	54	79
petclinic	mono							0
	"best"							0
	full							0
myweb	mono							0
	"best"							0
	full				2	4	1	7
react	mono							0
	"best"					12	33	45
	full					12	46	58

4.3.2 Anomalies Found per Type

In Table 4.3, we present the number of anomalies found for each decomposition by type of anomaly. These results may be helpful to the programmers since they allow them to anticipate unexpected behaviours that the application might have after the migration to microservices following a given decomposition. Another aspect to point out is the relatively high number of anomalies in the column "#Unclassified". This happens due to the fact that the types' counters are not incremented when there is an execution with two or more anomalies and our patterns only consider the minimum number of operations to describe each type of anomaly. For example, in a dirty read, it is only required two writes in one functionality but in different sub-transactions and one read in a different functionality, so that it is possible to have an execution where an intermediate value is read. However, if there is any extra operation in the execution considered, then that execution will not be identical to the one of our dirty read pattern, and, therefore, will be considered "Unclassified".

4.3.3 *TPC-C* Anomalies Found per Sub-Transactions

To illustrate the anomalies found grouped by sets of sub-transactions and the number of occurrences of each sub-transaction in the anomalies, we will focus only on one of the decompositions (*TPC-C full*). However, we could perform the same analysis for the other decompositions.

In Listing 4.1, we present the number of anomalies found by sets of sub-transactions and the number of occurrences of each sub-transaction in the anomalies of the *TPC-C full* decomposition. By analysing these results, one can notice that two functionalities alone cause most of the anomalies. These functionalities are "payment" and "delivery", which are responsible for 44 (4+20+20) and 24 (4+4+4+4+4+4) anomalies, respectively, out of the 98 anomalies found. As expected, the sub-transactions of these functionalities (“payment_0”, “payment_1”, “payment_2”, and “delivery_0”, “delivery_1”, “delivery_2”, “delivery_3”) occur in a large number of anomalies. Based on these results, one possible approach to mitigate this issue is to analyse the entities involved in those functionalities, and either reorder the operations, so that the operations that access the same entity are sequential, or design a decomposition where the entities managed by each of these functionalities are in the same microservice.

Listing 4.1: Number of *TPC-C full* anomalies by sub-transactions.

```
[newOrder_3, newOrder_7, orderStatus_1, orderStatus_2]: 2/98
[delivery_0, delivery_2, newOrder_4, newOrder_7]: 8/98
[delivery_1, delivery_2, newOrder_3, newOrder_7]: 8/98
[delivery_0, delivery_1, newOrder_3, newOrder_4]: 4/98
[newOrder_6, newOrder_7, stockLevel_1, stockLevel_2]: 1/98
[newOrder_2, newOrder_7, stockLevel_0, stockLevel_1]: 1/98
[newOrder_1, newOrder_2, payment_0, payment_1]: 2/98
[payment_0, payment_1]: 4/98
[payment_1, payment_2]: 20/98
[payment_0, payment_2]: 20/98
[delivery_0, delivery_3]: 4/98
[delivery_1, delivery_3]: 4/98
[delivery_0, delivery_2]: 4/98
[delivery_0, delivery_1]: 4/98
[delivery_1, delivery_2]: 4/98
[delivery_2, delivery_3]: 4/98
[newOrder_2, newOrder_6]: 4/98
delivery (delivery_0): 24/98
delivery (delivery_1): 24/98
delivery (delivery_2): 28/98
delivery (delivery_3): 12/98
newOrder (newOrder_1): 2/98
newOrder (newOrder_2): 7/98
newOrder (newOrder_3): 14/98
newOrder (newOrder_4): 12/98
newOrder (newOrder_6): 5/98
newOrder (newOrder_7): 20/98
orderStatus (orderStatus_1): 2/98
orderStatus (orderStatus_2): 2/98
payment (payment_0): 26/98
payment (payment_1): 26/98
payment (payment_2): 40/98
stockLevel (stockLevel_0): 1/98
stockLevel (stockLevel_1): 2/98
stockLevel (stockLevel_2): 1/98
```

Table 4.4: MAD performance comparison without (w/o) and with (w/) the search algorithm (SA).

Application	Decomposition	w/o SA [s]	w/ SA [s]
TPC-C	mono	7	13
	“best”	6	13
	full	2767	994
jpabook	mono	6	33
	“best”	5996	356
	full	7133	465
petclinic	mono	9	77
	“best”	7	79
	full	13	119
myweb	mono	11	259
	“best”	11	265
	full	248	400
react	mono	114	571
	“best”	(timeout)	1677
	full	(timeout)	1951

4.3.4 Performance of the Search Algorithm

To conclude MAD’s evaluation, we address the impact that our divide and conquer search algorithm has regarding MAD’s performance, as well as its capability to mitigate the existence of applications/decompositions that are too complex for MAD to analyse in a reasonable period of time. Table 4.4 presents MAD’s execution time to analyse each of the decompositions without and with the usage of the search algorithm that we described in Section 3.2.4.

From the analysis of Table 4.4, we can observe that the divide and conquer search algorithm does not always provide a performance improvement. However, for long analyses, it can significantly shorten their analysis time. The reason behind this is that the search algorithm was developed to mitigate the time and space complexities of MAD for large applications/decompositions, unintentionally neglecting the performance for simpler cases. This occurs because the search algorithm originates an overhead to MAD’s analysis by requiring that MAD unnecessarily iterates over combinations with no anomalies. Without the search algorithm, MAD would not need to do that since if after a search iteration it did not find any anomalies, it would increment the cycle length considered or end the analysis if it reached the maximum cycle length assumed. For complex cases, the search algorithm presents a significant performance improvement since it manages to simplify the SMT formula and mitigate the time and space complexities by not having to find anomalies while considering all the original transactions at the same time. Another positive aspect of the search algorithm is that it enables the analysis of complex cases to be performed under a timeout period (4 hours = 14400 seconds) that we defined as the reasonable amount of time a programmer would wait for the analysis to be complete. Without the search algorithm, MAD exceeds the timeout limit when analysing decompositions *“best”* and *full* of the *react* application, only finding 6 anomalies out of 45 and 6 anomalies out of 58, respectively, during that period of time.

Summary

In this chapter, we presented the experimental evaluation of MAD. We started by describing all the benchmarks we used, *microbenchmarks* and real-world applications, and the conditions under which we performed the evaluation. We compared MAD's analysis capabilities with the ones of a heuristic approach tool, "A Complexity Metric for Microservices Architectures Migration" which we designated as *CMMAM*. We applied MAD to real-world codebases and analysed the obtained results from an overall perspective, as well as using MAD's additional metrics. At last, we assessed the impact of our search algorithm regarding MAD's performance.

5

Conclusions and Future Work

In this thesis, we proposed an approach to automatically detect the anomalies that may appear when one migrates from a monolith to microservices following a given decomposition. We developed MAD, a tool that implements the proposed approach. The thesis described MAD's design and implementation, which includes several strategies to improve its performance and scalability. We performed an experimental evaluation of MAD, aiming to compare the results with alternative tools and assess its performance when used with real-world applications. The results highlighted not only how MAD's analysis can be more accurate than a heuristic approach, but also that it can be applied to existing applications, providing insights regarding how one can migrate their monolithic application to a microservices architecture.

Although we have incorporated in MAD techniques to process realistic use cases with reasonable performance, MAD still has some limitations. First, even though our search algorithm reduced the time and space complexities of the analysis, it can still be a problem for more complex cases. Second, our pattern-matching technique to determine the anomaly type is simple, lacking the notions that would allow it to assign the anomalies to each type better. Third, MAD only considers one replica of each node, therefore not being able to reproduce scenarios where there is replication of the nodes, and where the impact of the consistency model enforced between replicas would also have to be taken into account. At last, MAD lacks the notion of association between entities when they are in different microservices, such as JPA relationships, foreign keys, and semantic invariants that the system must respect. For future work, we consider that addressing these limitations may represent a significant improvement to MAD since it would allow the tool to be faster, more accurate, and able to target a wider range of scenarios.

Bibliography

[1] J. Thones, "Microservices," *IEEE Software*, vol. 32, no. 1, p. 116, January 2015. [Online]. Available: https://doi.org/10.1109/MS.2015.11

[2] C. Richardson, *Microservices Patterns: With examples in Java*, November 2018.

[3] C. Papadimitriou, "The serializability of concurrent database updates," *Journal of the ACM*, vol. 26, no. 4, p. 631–653, October 1979. [Online]. Available: https://doi.org/10.1145/322154.322158

[4] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems (TACAS)*, ser. TACAS'08, Budapest, Hungary, April 2008.

[5] K. Rahmani, K. Nagar, B. Delaware, and S. Jagannathan, "Clotho: Directed test generation for weakly consistent database systems," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, October 2019. [Online]. Available: https://doi.org/10.1145/3360543

[6] M. Santos, "Microservice decomposition for transactional causal consistent platforms," Master's thesis, Instituto Superior Técnico, Universidade de Lisboa, June 2022.

[7] L. Nunes, N. Santos, and A. Silva, "From a monolith to a microservices architecture: An approach based on transactional contexts," in *Proceedings of the 13th European Conference on Software Architecture (ECSA)*. Paris, France: Springer-Verlag, September 2019, p. 37–52. [Online]. Available: https://doi.org/10.1007/978-3-030-29983-5_3

[8] A. Kalia, J. Xiao, R. Krishna, S. Sinha, M. Vukovic, and D. Banerjee, "Mono2micro: A practical and effective tool for decomposing monolithic java applications to microservices," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. Athens, Greece: Association for Computing Machinery, August 2021, p. 1214–1224. [Online]. Available: https://doi.org/10.1145/3468264.3473915

[9] M. Brito, J. Cunha, and J. Saraiva, “Identification of microservices from monolithic applications through topic modelling,” in *Proceedings of the 36th Annual ACM Symposium on Applied Computing*, ser. SAC ’21. New York, NY, USA: Association for Computing Machinery, March 2021, p. 1409–1418. [Online]. Available: https://doi.org/10.1145/3412841.3442016

[10] H. Garcia-Molina and K. Salem, “Sagas,” in *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*. San Francisco (CA), USA: Association for Computing Machinery, December 1987, p. 249–259. [Online]. Available: https://doi.org/10.1145/38713.38742

[11] Jepsen consistency models. https://jepsen.io/consistency. Accessed: 24/12/2022.

[12] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. Hellerstein, and I. Stoica, “Highly available transactions: Virtues and limitations,” *Proceedings of the VLDB Endowment*, vol. 7, no. 3, p. 181–192, November 2013. [Online]. Available: https://doi.org/10.14778/2732232.2732237

[13] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O’Neil, and P. O’Neil, “A critique of ansi sql isolation levels,” in *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD ’95. San Jose, California, USA: Association for Computing Machinery, May 1995, p. 1–10. [Online]. Available: https://doi.org/10.1145/223784.223785

[14] D. Shasha, F. Llirbat, E. Simon, and P. Valduriez, “Transaction chopping: Algorithms and performance studies,” *ACM Transactions on Database Systems*, vol. 20, no. 3, p. 325–363, September 1995. [Online]. Available: https://doi.org/10.1145/211414.211427

[15] G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair, *Distributed Systems: Concepts and Design*, 5th ed. USA: Addison-Wesley Publishing Company, 2011.

[16] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. USA: Addison-Wesley Longman Publishing Co., Inc., 1987.

[17] P. Viotti and M. Vukolić, “Consistency in non-transactional distributed storage systems,” *ACM Computing Surveys*, vol. 49, no. 1, June 2016. [Online]. Available: https://doi.org/10.1145/2926965

[18] L. Lamport, “Time, clocks, and the ordering of events in a distributed system,” *Communications of the ACM*, vol. 21, no. 7, p. 558–565, July 1978. [Online]. Available: https://doi.org/10.1145/359545.359563

[19] D. Akkoorath, A. Tomsic, M. Bravo, Z. Li, T. Crain, A. Bieniusa, N. Preguiça, and M. Shapiro, “Cure: Strong semantics meets high availability and low latency,” in *Proceedings of the 36th IEEE International Conference on Distributed Computing Systems, (ICDCS)*. Nara, Japan: IEEE Computer Society, June 2016, pp. 405–414. [Online]. Available: https://doi.org/10.1109/ICDCS.2016.98

[20] N. Santos and A. Silva, "A complexity metric for microservices architecture migration," in *Proceedings of the 17th IEEE International Conference on Software Architecture (ICSA)*, Salvador, Brazil, March 2020. [Online]. Available: https://doi.org/10.1109/ICSA47634.2020.00024

[21] J. Almeida and A. Silva, "Monolith migration complexity tuning through the application of microservices patterns," in *Proceedings of the 14th European Conference on Software Architecture (ECSA)*, L'Aquila, Italy, September 2020. [Online]. Available: https://doi.org/10.1007/978-3-030-58923-3_3

[22] C. Tan, C. Zhao, S. Mu, and M. Walfish, "Cobra: Making transactional key-value stores verifiably serializable," in *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Virtual Event, November 2020.

[23] S. Bayless, N. Bayless, H. H. Hoos, and A. J. Hu, "SAT modulo monotonic theories," in *Proceedings of the 25th AAAI Conference on Artificial Intelligence (AAAI)*. Austin (TX), USA: AAAI Press, January 2015, p. 3702–3709.

[24] R. Biswas, D. Kakwani, J. Vedurada, C. Enea, and A. Lal, "Monkeydb: Effectively testing correctness under weak isolation levels," *Proceedings of the ACM on Programming Languages*, vol. 5, no. OOPSLA, October 2021. [Online]. Available: https://doi.org/10.1145/3485546

[25] K. Nagar and S. Jagannathan, "Automated detection of serializability violations under weak consistency," in *Proceedings of the 29th International Conference on Concurrency Theory (CONCUR)*, S. Schewe and L. Zhang, Eds., Beijing, China, September 2018. [Online]. Available: https://doi.org/10.4230/LIPIcs.CONCUR.2018.41

[26] A. Adya, B. Liskov, and P. O'Neil, "Generalized isolation level definitions," in *Proceedings of the 16th International Conference on Data Engineering (ICDE)*. San Diego (CA), USA: IEEE Computer Society, February 2000.

[27] D. Difallah, A. Pavlo, C. Curino, and P. Cudre-Mauroux, "Oltp-bench: An extensible testbed for benchmarking relational databases," *Proceedings of the VLDB Endowment*, vol. 7, no. 4, p. 277–288, December 2013. [Online]. Available: https://doi.org/10.14778/2732240.2732246