Automatic Detection of Anomalies in Microservices Architectures

Valentim Romão
valentim.romao@tecnico.ulisboa.pt

Instituto Superior Técnico
(Advisors: Professor Luís Rodrigues and Professor Vasco Manquinho)

Abstract. The microservices architecture is a design style that structures an application as a set of loosely coupled services. This approach has several advantages, but also some disadvantages when compared with the traditional monolithic architecture. A key challenge that the microservices architecture brings is the potential lack of isolation among the concurrent execution of functionalities that span multiple microservices. In a monolithic application, each functionality is typically implemented by a transaction that accesses a single-database with ACID properties. In a microservices architecture a functionality may need to be broken into multiple independent sub-transactions, each executed by a different microservice, and possibly accessing a different repository. Even if each sub-transaction is isolated from other sub-transactions executed by the same microservice, the potential interleaving among sub-transactions of different executions of one or multiple functionalities may lead to unexpected results, also named anomalies. These would never occur in the monolithic implementation and correspond to unintended outcomes. In this work, we address the problem of automatically detecting anomalies that might appear when decomposing a monolithic application into a composition of microservices.

Keywords: Monolith · Microservices · Anomaly detection

Table of Contents

1 Introduction

The microservices architecture is a design style that structures an application as a set of loosely coupled services. This architecture has several advantages with regard to a more traditional monolithic architecture. First, because each microservice only implements the logic related to a small subset of the entities managed by the entire application, the code becomes highly cohesive and easier to develop and maintain. Second, because microservices are loosely coupled, it is easier to assign different teams to program each service, which allows the work to be done in parallel and with minimal interference between each team. Third, it provides more flexibility, as multiple services can be deployed in different machines, therefore managed and provisioned independently. For instance, it is possible to assign more resources to a given microservice, instead of to the entire application. Due to these advantages, many companies and software developers are now using microservices architecture when building their applications, and some are even migrating legacy monolithic applications to the microservices architecture, to ease maintenance and scaling [1].

However, the microservices architecture also has some disadvantages when compared to a monolithic architecture. An application typically implements multiple functionalities, each accessing several data objects representing different domain entities. In a monolithic application, all these data objects are often stored in a single database. Furthermore, each functionality is typically implemented by a transaction that offers ACID (**A**tomicity, **C**onsistency, **I**solation, **D**urability) properties. This simplifies the application's design, as programmers do not need to be concerned with the effects that may arise from the concurrent executions of the same or different functionalities, or even by the partial execution of functionalities. In general, this is no longer viable when using a microservices architecture. Each microservice tends to use its own storage, which can be independent of the storage used by other microservices. Thus, a functionality needs to be implemented by a set of independent sub-transactions, each executed by a different microservice. Even if each sub-transaction is isolated from other sub-transactions executed by the same microservice, the concurrent execution of functionalities may lead to an interleaving of operations that would never occur in the monolithic application. This new interleaving may lead to unexpected results, also named anomalies, that correspond to non-serializable executions of the transactions. Moreover, the execution of a functionality may no longer be atomic, since each sub-transaction commits individually. This is a problem because it no longer assures the global commit of the operations' effects, which was previously guaranteed by all the operations being in one single transaction. Anomalies that may result from the lack of atomicity and isolation are managed by executing compensating actions, which represent an additional burden to the programmer of microservices applications.

To make things even more complex, a microservice implementation often needs to read data that is managed by another microservice. Different implementations of the microservices architecture may use different techniques to support such accesses. In some cases, a microservice just makes a remote-call

to the other microservice to read a value. In other cases, updates to data values are disseminated among the relevant microservices using a publish-subscribe middleware. In the latter approach, a microservice may cache the updates received asynchronously, and subsequently read the values of remote data items from the local cache. Depending on how the updates are propagated and made visible locally at a given cache, a microservice may read remote data items with different levels of data consistency. The potential lack of consistency from different microservices reading from different database snapshots is another source of anomalies, since it allows microservices to read different versions of the same object.

When a microservice implementation is the result of decomposing a legacy monolithic implementation, the number of anomalies that may result depends on how the monolith is decomposed. Namely on how many microservices interact with each other, and on which domain entities are managed by each microservice. In theory, it should be possible to find the anomalies that result from a given decomposition in an automated manner. For that, one would need to compare all outcomes that are possible with the monolithic implementation when each functionality is executed in a single transactional context, with the outcomes that are possible when the functionalities are executed as a sequence of independent sub-transactions in the target set of microservices. Finding these anomalies can help the programmer to select the best decomposition, as well as ensuring that the appropriate compensating action is developed to fix each possible anomaly. Finding these anomalies in an automated and exhaustive manner is relevant because concurrency anomalies are known to be notoriously hard to identify via testing [2]. Unfortunately, this task is not trivial because one needs to consider not only the effects caused by chopping a transaction into a sequence of independent sub-transactions, but also the effects that may result from having sub-transactions reading mutually-inconsistent versions of remote objects from the local cache of the microservice.

In this work, we aim to study techniques that may automate the process of finding anomalies in a microservices decomposition of a monolithic application. To this purpose, we will study methods that address the problem of finding anomalies that result from transaction chopping and methods that address the problem of finding anomalies that result from reading inconsistent data. We also aim to explore techniques to address both problems together. As will be discussed later in this report, we believe that our goals can be achieved by extending an existing tool that only addresses the effect of data consistency (CLOTHO) with mechanisms to address the effects of transaction chopping.

The rest of the report is organized as follows. Section 2 briefly summarizes the goals and expected results of our work. In Sections 3 and 4, we present the background of the related work. Section 5 describes the proposed architecture to be implemented, and Section 6 describes how we plan to evaluate our results. Finally, Section 7 presents the schedule of future work, and Section 8 concludes the report.

2 Goals and Expected Results

This work addresses the problem of automating the process of finding the anomalies that arise from the decomposition of a monolithic application into a composition of microservices.

> *Goals:* We aim to create a tool that, given the source code of a monolithic application, and the source code of a target decomposition of the same application into multiple microservices, can automatically identify the anomalies that may occur when executing the functionalities in the microservices architecture.

To accomplish this goal, we intend to extend CLOTHO, an existing tool that is able to identify anomalies that may result from having transactions read inconsistent versions of data. Our extension will give it the ability to identify anomalies that result from the transactional chopping that needs to be performed as a result of the decomposition.

> *Expected Results:* The work will produce i) a study of how to identify anomalies that may appear in a microservices decomposition; ii) an implementation of a tool capable of doing such analysis; iii) an extensive evaluation with different decompositions and considering different consistency models that may be used by the implementation of the microservices architecture to support remote access to remote data.

3 Background

In this section, we will present the background knowledge required for our work. We will start by introducing the monolithic and microservices architectures, explain what are transactions and the properties associated with them, followed by a list of anomalies that can occur on a system. After that, we present mechanisms used to coordinate transactions, and a technique to divide transactions. Next, we delve into the topic of having distributed databases, and how data can be replicated between nodes. Finally, we explain how transactions are normally implemented in a microservices application, and a strategy used to mitigate anomalies when using microservices.

3.1 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 are 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. 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.

Table 1 presents a comparison between the two architectures.

Table 1: Architectures Comparison.

	Complexity	**Modularity**	**Scalability**	**Graceful Degradation**	**ACID Properties**
Monolithic	Low	Low	Low	No	Easy
Microservices	High	High	High	Yes	Hard

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 which ease the process of the monolith decomposition [3,4], 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.

3.2 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, that ensures that if the effects of one operation take place, then the effects of all operation 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 later focus on in this work, is long-lived transactions, which represent transactions that access many database objects and take a large amount of time to complete [5].

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 [6] 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 are allowed.

Strict Serializability [6] 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 isolation has some costs that depend on the concurrency method used and on the characteristics of the workload. As a result, transactions may execute slower, be delayed, or be forced to abort. 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, in turn, may lead to inconsistent results. The weaker isolation levels are read uncommitted, read committed and repeatable read.

Read Uncommitted [7] 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 [7] 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 [7] 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.

3.3 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 [8].

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 Fig. 1a.

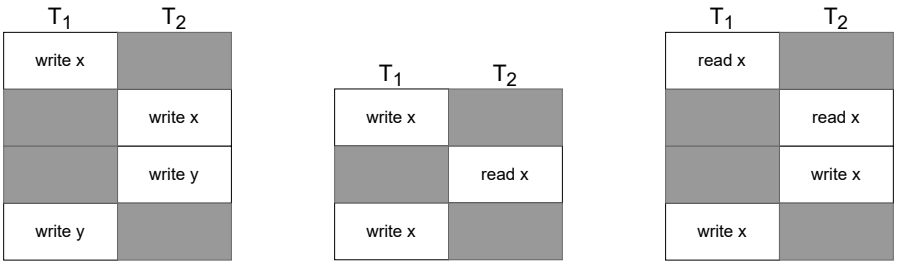
Dirty Read is an anomaly generated when the system has two transactions, T_1 and T_2, T_1 composed of two updates and T_2 composed of a read, all applied to the same variable in the database, x. Then the following scenario happens, T_1

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 Fig. 1b.

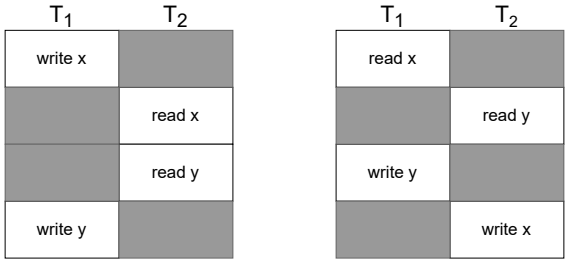
Lost Update is an anomaly similar to the dirty write, but, in this case, T_1 will not initially write to the same variable as T_2. Now, T_1 will read the variable, then T_2 reads and updates the variable, and T_1 finishes its job by updating the variable. This will result in the value prevailing to be 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 Fig. 1c.

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 Fig. 1d.

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 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 Fig. 1e.



(a) Dirty Write. (b) Dirty Read. (c) Lost Update.



(d) Read Skew. (e) Write Skew.

Fig. 1: Anomalies examples.

3.4 Concurrency Control

In order 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. In order 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 high 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 access a high 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.

3.5 Transaction Chopping

As it was mentioned 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 time 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. [9], was introduced as a technique to avoid long-lived transactions. This method consists in dividing the large transactions into smaller transactions (sub-transactions). We define sub-transaction as a transaction composed of a subsequence of operations from one transaction. The chopping method consists in identifying subsequences of operations from a transaction, considering the objects they access and the type of access, and, based on that, create 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 number of issues 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, for instance, the 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 authors consider that chopping to be *rollback-safe*.

In their work, Shasha et al [9] also delve into the topic of verifying if a chopping is correct. The authors state 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. In the paper, this graph is called *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.

To illustrate a chopping of transactions, we elaborated an example inspired by an example from the book "Microservices Patterns: With examples in Java" [10]. In this example, we have two tables, *PRODUCTS* and *ORDERS*. The first table is composed by a *product_id* and a *state* ("available", "ordered" or "out-of-stock") and the second table is composed by an *order_id* and a *product_id*. We also have four transactions, *placeOrder*, *cancelOrder*, *checkProductOrder* and *removeProduct*, their implementations are shown in the example, and they are, respectively, responsible for placing an order, canceling an order, checking which is the order id of a given product, if it was ordered, and removing a product from stock.

Fig. 2 presents the original set of transactions, and Fig. 3 presents the same set of transactions after being partially chopped. We decided to chop *placeOrder* and *cancelOrder*, since they were accessing objects from both tables, like the other transactions, but were also the biggest transactions, making them the best candidates to be chopped. Note that this was a handcrafted chopping made by us, to illustrate what a set of transactions looks like before and after being chopped, so it may be, and in fact is, incorrect. However, the chopping we made is incorrect on purpose, so that later we are able to use this example to motivate our solution.

3.6 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, fault tolerance, among others. These databases tend to run

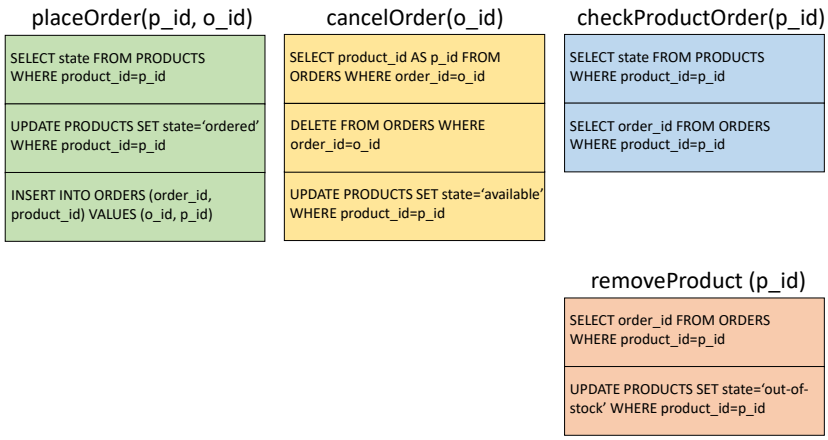


Fig. 2: Monolith transactions.

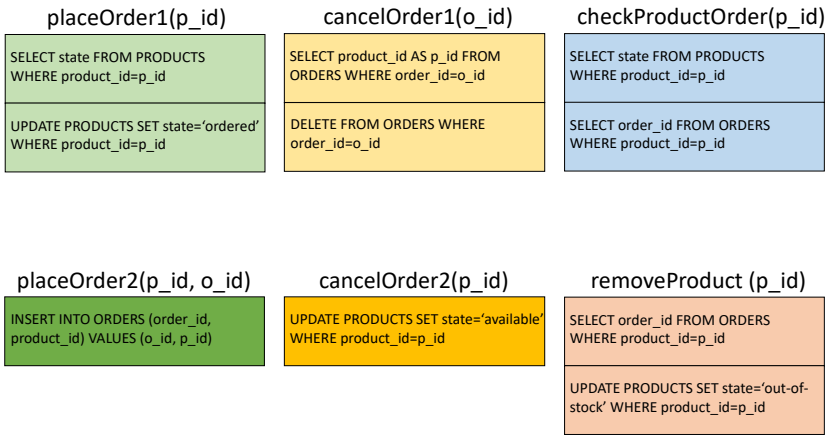


Fig. 3: Microservices transactions.

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 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 be verifying 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 [11]. 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 3.4 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 but for the database objects of T_2. If this happens, then both nodes may 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.

3.7 Data Replication

In order 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 choose

to read from a replica that is local, or at least closer, 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 an 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 [12]. 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 of 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 [8].

There are several consistency models that a programmer can use. The following list presents the consistency guarantees that we will later use in our evaluation:

Eventual Consistency [7] 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 [13] is a property that affects one specific object, and is derived from the concept of visibility and Lamport's *Happens-Before* relationships [14]. 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 [13] 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 [6] 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.

3.8 Transactions in Microservices Architectures

When implementing a microservices application, software developers opt to use some of the techniques and mechanisms previously presented. There are two aspects that 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 in order 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 (TCC), which is defined as follows:

Transactional Causal Consistency [15] 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).

3.9 Strategies to mitigate Anomalies

In order to mitigate anomalies in a monolith, developers can use stronger database consistency models, neglecting performance. On the other hand, for microservices, it is not that simple, since they usually have more than one database, and when there is data replicated in more than one database, different services may read different versions of the data. Therefore, software developers came up with several design patterns specific to microservices. In the book "Microservices Patterns: With examples in Java" [10], we can find several of these patterns, as well as examples of their applications.

A pattern that is strongly related to Transaction Chopping is the *Saga* pattern, introduced by Garcia-Molina and Salem [5]. A saga is composed of a sequence of sub-transactions, which originally belonged to the same long-lived transaction. Note that, previously, these sub-transactions were being executed as operations inside a single transaction, whereas now they are being executed as multiple, smaller transactions. Due to this change, one must take into account that each sub-transaction can fail, and leave the execution of a saga incomplete. To fix this issue, the authors introduced compensating transactions to revert the actions from partial executions of a saga. This type of transaction is helpful for software developers, since it allows the possibility of reverting work done by previous sub-transactions in case of failure. Now, when a sub-transaction fails, the system will proceed to backward recover, which consists in executing the compensating transactions of the previous sub-transactions. The authors also introduce the possibility of adding a save-point, which is a point where all the changes made by sub-transactions behind it are saved. If one sub-transaction

fails after the save-point, then the system only needs to run compensating transactions for the sub-transactions executed after the point. The Saga pattern has since evolved to use a model defined by Lars Frank and Torben U. Zahle [16], which presented three types of sub-transactions. *Compensatable*, a transaction that can be reverted by a compensating transaction. *Pivot*, a transaction that serves as the save-point. And finally, *Retriable*, a transaction that does not require a compensating transaction and can just be retried in case of failure.

4 Related Work

In this section, we will present several tools with implementations and objectives similar to the ones we intend to achieve. In Section 4.1, 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. In Section 4.2, we cover tools that use a black box approach (do not require information on how the system is built) to detect anomalies. In Section 4.3, we cover tools that use a white box approach (need to know how the system is implemented) to detect anomalies. In Section 4.4, we present a table comparing all the tools.

4.1 Heuristics for Anomaly Awareness

A Complexity Metric for Microservices Architecture Migration [17] 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 their work, the authors define a microservice as a group of entities called a *cluster*, and to aggregate the entities into clusters, a similarity measure is used. In the paper, a *similarity measure* consists in a criterion that quantifies how coupled two entities are. The paper 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' complexity. 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 write to that same entity, and vice-versa.

For the second contribution, the authors create clusters using four different similarity measures to aggregate the entities. 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.

Mono2Micro - From a Monolith to Microservices: MetricsRefinement [18] is a research work with the intent of improving the accuracy of the metric presented in A Complexity Metric for Microservices Architecture Migration [17] by providing the user a set of operations to manipulate a monolith decomposition. The authors' goal is to offer a more realistic scenario of a migration to microservices considering the *Saga Pattern*, 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*, that 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, 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 presents three new 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 *Sys-*

tem 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, cluster(s) read by the query.

4.2 Anomaly Detection using Black Box Approaches

Cobra [19] is a tool designed to test if a transactional key-value store in the cloud respects serializability. Cobra 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 *Client* is meant to mimic the actions of real clients by randomly sending requests to the database under test, and receiving the respective responses. The *History Collector* is responsible for logging all the operations made and responses received in the interaction between a specific Client and the database. Each History Collector 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 a history and verify 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 authors defined 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 be reached by the constraint definition. After applying all these techniques, they use the resultant graph, the constraints, encoded using logic notation, and the SMT solver (in their case MonoSAT [20]), to verify if the graph is acyclic, since this would mean that the execution is serializable.

MonkeyDB [21] is a system designed for users to test their program against multiple consistency levels of a DB 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 component 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*, that uses logical constraints (axioms) to represent the properties of consistency models and is responsible to check if the execution is valid under the consistency model selected 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 execute 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 absolutely 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.

4.3 Anomaly Detection using White Box Approaches

Automated Detection of Serializability Violations under Weak Consistency (ANODE) [22] presents a tool that statically analyses a given SQL-like program, to check if it contains serializability anomalies under bounded abstract executions and different weak consistency models.

To use this tool, first, 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 to convert each abstract execution to a dependency graph. To replicate the behaviour of consistency models, they use 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 a theorem prover 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.

CLOTHO [23] is based on the work of Nagar and Jagannathan [22] and is also a tool designed to analyse the transactions of a storage system that provides weak consistency semantics, to detect if there are any serializability anomalies.

The tool receives as input a Java program with a single class, containing a set of methods representing the system 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 a user to run concrete executions that will lead to the anomalies detected.

The tool's pipeline can be split into three steps for the analysis, and one last step for the anomaly concrete execution.

First, the tool converts the Java program to an *Abstract Representation* (AR), which resembles an SQL-like language and is described by the authors in the paper. This step eases the extraction of information from the input program, such as the dependencies between operations that can access the same object. The types of dependencies considered are: RW (reads from a value that will be written); WR (writes on a value that will be read); and WW (writes on a value that will be written).

Second, using the program in the AR, the goal is to construct a *First Order Logic* (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 [24]), 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, then this means that no anomalies were detected for cycles of length equal or smaller than the predefined maximum length.

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.

Microservice Decomposition for Transactional Causal Consistent Platforms (CLOTHO+) [25] is a work strongly related to ours, since it also focus on the microservices architecture, and extended CLOTHO to execute new functionalities associated with this 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 [23], 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 in different machines and avoid raising a false positive, 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 belonged to the same transaction.

4.4 Tools Comparison

Table 2 presents a comparison of the tools. We split the table into two sections, "Properties" and "Techniques". The "Properties" section refers to the characteristics of the tools. The "Techniques" section refers to the mechanisms used by the tools to fulfil their purpose. We divide the "Properties" into three columns, "Analysis", "Consistency Model", and "Microservices Oriented?". "Analysis" refers to the approach used by the tool to analyse a system. "Consistency Model" refers to the consistency guarantees that the tool expects the system under testing to respect, considering the ones that the tool can be extended to enforce ("-" means that the tool does not take into account the consistency model of the system). "Microservices Oriented?" indicates if the tool is oriented to analyse microservices or not. For the "Techniques" section, we also divide it into three columns, "Method", "SMT Solver?", and "Executions Analysed". "Method" refers to how the tool identifies the existence of anomalies. "SMT Solver?" indicates if the tool uses an SMT solver or not. "Executions Analysed" refers to the type of executions that the tool handles ("-" means that the tool does not consider different executions of the system).

Table 2: Comparison between the related tools we presented.

	Properties			**Techniques**		
	Analysis	**Consistency Model**	**Microservices Oriented?**	**Method**	**SMT Solver?**	**Executions Analysed**
Cobra	Black box	Serializability	No	Testing	Yes	Abstract
MonkeyDB	Black box	Any	No	Testing	No	Concrete
Complexity Metric	White box	-	Yes	Static Analysis	No	-
Metrics Refinement	White box	-	Yes	Static Analysis	No	Abstract
ANODE	White box	Any	No	Static Analysis	Yes	Abstract
CLOTHO	White box	Any	No	Static Analysis	Yes	Abstract/ Concrete
CLOTHO+	White box	Any	Yes	Static Analysis	Yes	Abstract

5 Architecture

For our solution, we will use as a starting point CLOTHO [23] with the extensions made by Madalena Santos [25]. Our goal is to further extend CLOTHO, to make it capable of detecting anomalies in microservices applications that originated from a monolith to microservices migration. At the moment, CLOTHO cannot do this, because it does not allow to express that two or more transactions may have resulted from the chopping of the same transaction, and that different transactions may, or may not, execute on the same machine depending on the microservice they belong to.

5.1 Tool Modules

As mentioned in Section 4.3, CLOTHO has two main components, the analyzer and the replayer. Considering our goal, we will only focus on the analyzer, more precisely on the Z3 module, since this is the module where we intend to implement our logical representation of sub-transactions, as well as represent how they are expected to behave in an abstract execution.

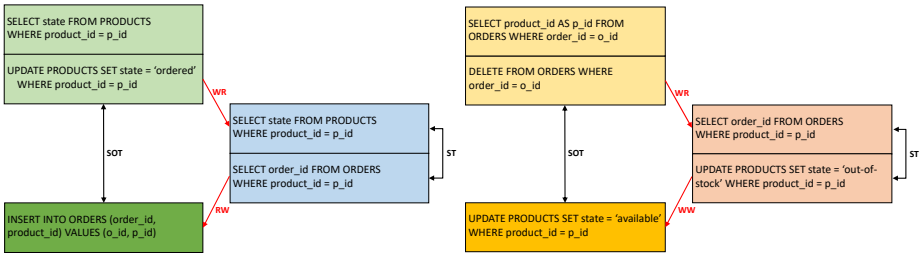
5.2 Sub-transactions Representation

To represent the sub-transactions, we will need to consider three aspects. First, the original transaction that originated the sub-transactions, so that we can adapt the analysis to consider the fact that the executions of two or more sub-transactions from a same original transaction are not completely independent from each other. Second, the order of the sub-transactions inside the original transaction, to avoid executions that would not be possible in the system, since the sub-transactions will continue to respect the operations' execution order that existed before the chopping. Third, the microservice in which the sub-transaction executes and the machine(s) it runs on, to account for the cases when a microservice needs to access data that belongs to a different microservice.

5.3 Relations between Sub-transactions and other Transactions

Considering the relations between operations in CLOTHO, we would suggest that the dependency relations between the operations of a sub-transaction and the operations of other transactions remained the same (RW, WR, WW). The only difference would be a new relation between operations of different sub-transactions that used to belong to the same original transaction. This relation would be similar to the ST (Same Transaction) relation, but less restrictive. For now, we will call this new relation SOT (Same Original Transaction). The difference between ST and SOT resides in the properties assured by each relation. For example, under serializability, an ST relation would provide ACID properties and not allow operations from different transactions to interleave, whereas SOT would be weaker and only provide ACD properties, allowing for other operations to interleave the execution of the sub-transactions.

In order to illustrate these relations, we used the transactions of the microservices system that we presented in Fig. 3, and made two examples where we drew the relations that are meant to be established between the operations. In Fig. 4a, we are executing *placeOrder1* and *placeOrder2* concurrently with *checkProductOrder*. As it can be seen from the figure, there is a cycle, so the execution of these transactions will be non-serializable and lead to an anomaly, a Dirty Read. The Dirty Read happens, because a product is marked as "ordered", but when checking for the product's order it cannot be found, since the order is still not yet in the database. In Fig. 4b, we are executing *cancelOrder1* and *cancelOrder2* concurrently with *removeProduct*. In this case, we also have a cycle, meaning that this execution is also non-serializable and has an anomaly, a Lost Update. This anomaly happens, because a given product order is deleted by *cancelOrder1*, which leads to no orders for that product in the database, allowing *removeProduct* to remove the product from stock (update its state to "out-of-stock"). However, when *cancelOrder2* executes, the product state will be placed as "available", obfuscating the fact that the product should have been removed.



(a) Dirty Read. (b) Lost Update.

Fig. 4: Examples of relations between transactions.

5.4 Anomalies Detection

To detect the anomalies, we will need to adapt CLOTHO's assertions to include the processing of the SOT relation. We are aiming for it to be processed similarly to the ST relation, with the major difference being that SOT will not provide isolation, since other transactions can interleave the sub-transactions executions. SOT also intends to account for the microservices of the sub-transactions, and the machines they will execute on, so that the analysis process is aware when data from different remote microservices is being accessed, since it may lead to inconsistencies. After doing this adaptation, the rest would be handled by CLOTHO, which would create the FOL formula considering the sub-transactions

assertions, find the assignments that would satisfy the formula using Z3, and output the assignments found (anomalies), if any were found.

6 Evaluation

The focus of our evaluation will be on how many anomalies, resulting from a monolith to microservices decomposition, our system can detect. Another aspect that we will test is how different consistency models affect the number of anomalies found in the decomposed application compared with the monolith application under the same consistency model.

6.1 Number of Anomalies Detected

We will gather several examples of transactions in a monolithic application and convert them to chopped transactions. For each example, we will first assess the number of anomalies in the monolithic application, so that, if our tool's analysis detects any new anomaly after the decomposition, we can safely assume that it originated from the migration to microservices. After that, we will check the number of anomalies detected on the microservices application and compare it with the number of anomalies detected on the monolithic application (ground-truth). By doing this procedure, we can check if our tool manages to detect the anomalies resulting from the migration from a monolith to microservices.

6.2 Impact of the Consistency Model

We will also assess the impact that the consistency model has on the number of anomalies detected. We will do this by analysing the same examples as before, but with the different consistency constraints added to CLOTHO by Madalena Santos [25]. The objective is to infer how the different consistency guarantees affect the number of anomalies detected.

7 Scheduling of Future Work

Future work is scheduled as follows:

- January 9 - March 29: Detailed design and implementation of the proposed architecture, including preliminary tests.
- March 30 - May 3: Perform the complete experimental evaluation of the results.
- May 4 - May 23: Write a paper describing the project.
- May 24 - June 15: Finish the writing of the dissertation.
- June 15: Deliver the MSc dissertation.

8 Conclusions

The microservices architecture has recently become a standard when developing large-scale and highly complex applications, due to its advantages suiting developers' needs. First, modularity allows teams to work in parallel in each service. Second, high scalability properties, due to the possibility of assigning different resources to different services, and scaling each service independently of the others. Third, the high availability of the system, since it provides graceful degradation by allowing the system to run with a partial set of services. However, some changes must be made to the code, transactions and resources, in order to adopt this architecture. From these changes, problems regarding the execution of the transactions can arise. We focus particularly on the problems originating from the chopping of transactions, which are the anomalies. These may appear after a decomposition, since the isolation property, previously provided by the original transactions, no longer holds in the execution of the sub-transactions, and this can compromise the whole system, because each transaction can now see/modify intermediate database states that previously were not exposed.

Therefore, the final goal of our work is to provide a tool capable of detecting the anomalies that might appear when one decomposes the transactions of a monolith to fit the microservices requirements.

In summary, this report starts by introducing the context of our problem, presenting some background knowledge for the reader to be acquainted with the concepts, and enumerating several works related to ours. After that, we explained the architecture of the tool we intend to build, how we propose to evaluate it, and the schedule of our future work.

Acknowledgments We are grateful to Rafael Soares and João Queirós for the fruitful discussions and comments during the preparation of this report. This work was partially supported by project DACOMICO (via OE with ref. PTDC/CCI-COM/2156/2021).

References

1. Thones, J.: Microservices. IEEE Softw. **32**(1) (jan 2015) 116
2. Papadimitriou, C.H.: The serializability of concurrent database updates. J. ACM **26**(4) (oct 1979) 631–653
3. Nunes, L., Santos, N., Silva, A.: From a monolith to a microservices architecture: An approach based on transactional contexts. In: Software Architecture: 13th European Conference, ECSA 2019, Paris, France, September 9–13, 2019, Proceedings, Paris, France, Springer-Verlag (2019) 37–52
4. Kalia, A., Xiao, J., Krishna, R., Sinha, S., Vukovic, M., Banerjee, D.: 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 2021, Athens, Greece, Association for Computing Machinery (2021) 1214–1224

5. Garcia-Molina, H., Salem, K.: Sagas. In: Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data. SIGMOD '87, San Francisco, California, USA, Association for Computing Machinery (1987) 249–259
6. : Jepsen consistency models. `https://jepsen.io/consistency` Accessed: 24/12/2022.
7. Bailis, P., Davidson, A., Fekete, A., Ghodsi, A., Hellerstein, J., Stoica, I.: Highly available transactions: Virtues and limitations. Proc. VLDB Endow. **7**(3) (nov 2013) 181–192
8. Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E., O'Neil, P.: A critique of ansi sql isolation levels. In: Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data. SIGMOD '95, San Jose, California, USA, Association for Computing Machinery (1995) 1–10
9. Shasha, D., Llirbat, F., Simon, E., Valduriez, P.: Transaction chopping: Algorithms and performance studies. ACM Trans. Database Syst. **20**(3) (sep 1995) 325–363
10. Richardson, C.: Microservices Patterns: With examples in Java. Manning (2018)
11. Coulouris, G., Dollimore, J., Kindberg, T., Blair, G.: Distributed Systems: Concepts and Design. 5th edn. Addison-Wesley Publishing Company, USA (2011)
12. Bernstein, P.A., Hadzilacos, V., Goodman, N.: Concurrency Control and Recovery in Database Systems. Addison-Wesley Longman Publishing Co., Inc., USA (1987)
13. Viotti, P., Vukolić, M.: Consistency in non-transactional distributed storage systems. ACM Comput. Surv. **49**(1) (jun 2016)
14. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21**(7) (jul 1978) 558–565
15. Akkoorath, D., Tomsic, A., Bravo, M., Li, Z., Crain, T., Bieniusa, A., Preguiça, N., Shapiro, M.: Cure: Strong semantics meets high availability and low latency. In: 36th IEEE International Conference on Distributed Computing Systems, ICDCS 2016, Nara, Japan, June 27-30, 2016, IEEE Computer Society (2016) 405–414
16. Frank, L., Zahle, T.: Semantic acid properties in multidatabases using remote procedure calls and update propagations. Softw. Pract. Exper. **28**(1) (jan 1998) 77–98
17. Santos, N., Silva, A.: A complexity metric for microservices architecture migration. In: 2020 IEEE International Conference on Software Architecture, ICSA 2020, Salvador, Brazil, March 16-20, 2020, IEEE (2020) 169–178
18. Almeida, J., Silva, A.: Monolith migration complexity tuning through the application of microservices patterns. In: Software Architecture: 14th European Conference, ECSA 2020, L'Aquila, Italy, September 14–18, 2020, Proceedings, L'Aquila, Italy, Springer-Verlag (2020) 39–54
19. Tan, C., Zhao, C., Mu, S., Walfish, M.: Cobra: Making transactional key-value stores verifiably serializable. In: Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation. OSDI'20, USA, USENIX Association (2020)
20. Bayless, S., Bayless, N., Hoos, H.H., Hu, A.J.: Sat modulo monotonic theories. In: Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence. AAAI'15, AAAI Press (2015) 3702–3709
21. Biswas, R., Kakwani, D., Vedurada, J., Enea, C., Lal, A.: Monkeydb: Effectively testing correctness under weak isolation levels. Proc. ACM Program. Lang. **5**(OOPSLA) (oct 2021)
22. Nagar, K., Jagannathan, S.: Automated detection of serializability violations under weak consistency. In Schewe, S., Zhang, L., eds.: 29th International Conference on Concurrency Theory, CONCUR 2018, September 4-7, 2018, Beijing, China.

Volume 118 of LIPIcs., Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2018) 41:1–41:18
23. Rahmani, K., Nagar, K., Delaware, B., Jagannathan, S.: Clotho: Directed test generation for weakly consistent database systems. Proc. ACM Program. Lang. **3**(OOPSLA) (oct 2019)
24. De Moura, L., Bjørner, N.: 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'08/ETAPS'08, Budapest, Hungary, Springer-Verlag (2008) 337–340
25. Santos, M.: Microservice decomposition for transactional causal consistent platforms. Master's thesis, Instituto Superior Técnico, Universidade de Lisboa (June 2022)