Detection of Invariant Violations in Microservices

PIC2 - Master in Computer Science and Engineering
Instituto Superior Técnico, Universidade de Lisboa

João Quelhas Guterres Serrão Fitas — 95609*
joao.quelhas.fitas@tecnico.ulisboa.pt

Advisor: Professor António Rito Silva and Professor Luís Rodrigues

Abstract In a monolith, functionalities are executed as transactions, that are isolated from each other. In a microservice architecture, functionalities may be composed of multiple transactions, each executed in a different microservice. When functionalities execute concurrently, these individual transactions may interleave in unexpected ways, generating global states that violate correctness invariants. This work studies techniques to: 1) detect automatically executions that may cause invariants to be violated, and 2) automatically generate test cases that illustrate those executions.

Keywords — Microservices, Concurrency, Invariants, Anomaly Detection

*I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa (https://nape.tecnico.ulisboa.pt/en/apoio-ao-estudante/documentos-importantes/regulamentos-da-universidade-de-lisboa/).

Contents

1 Introduction

Microservices is an architectural style that promotes the development of applications through the composition of multiple small services that are loosely coupled, in contrast to the traditional monolithic architecture in which all functionalities are provided by a single software component [1–3]. Microservice architectures have several advantages over monolithic architectures. In particular, they are easier to scale, both from the perspective of the software development process and from the perspective of the deployment and execution.

Unfortunately, managing the effect of concurrency becomes more challenging in a microservice architecture. In a monolith, functionalities are executed as transactions that are isolated from each other. In a microservice architecture, functionalities may be composed of multiple transactions, each executed in a different microservice. When functionalities execute concurrently, these sequences of independent transactions may interleave in unexpected ways, generating global states that violate correctness invariants.

Executions that generate interleavings that cause application invariants to be violated are notably hard to detect manually or by testing, because the number of interleavings is combinatorial and not all interleavings generate violations. Techniques that automate the process of finding problematic interleavings can be extremely helpful in this scenario. In addition, it is interesting to automatically generate test cases that help programmers understand how to change the code to prevent invariants from being violated.

1.1 Goals

The objective of this work is to facilitate the process of designing complex applications when using a microservice architecture by automatically discovering and reproducing executions that may cause correctness invariants to be violated, so that the programmer can analyze each of these executions and correct them if needed. To do so, we aim to produce a tool that, given a representation of a Microservice application and its invariants, will automatically discover every problematic chaining of functionalities the composition may allow and generate concrete executions that trigger the discovered violations.

1.2 Expected Results

Our work is expected to produce a tool for the automated detection and reproduction of invariant violations in microservice architectures. We also plan to perform an extensive evaluation of the tool using different realistic test cases.

1.3 Organization of the Document

The remainder of the document is organized as follows. Section 2 provides the necessary background and motivation for our work. After that, Section 3 presents some of the tools and frameworks that were analyzed and highlights the more relevant aspects for this work. Section 4 discusses the limitations of previous works to detect invariant violations. Then, Section 5 presents the design of the proposed tool. Leading to the proposed evaluation of the tool in Section 6 and the scheduling of future work in Section 7. Lastly, Section 8 contains the conclusions of this work.

2 Background and Motivation

2.1 The Microservices Architecture

In the traditional monolithic architecture, a centralized piece of software is responsible for handling all the functionalities of the application. In a microservice architecture [1–3], the application is supported by multiple independent services, each with a single simple responsibility, that interact with each other with little centralized control to provide the application's functionalities. Each of these components should be independently deployable with its own infrastructure and potentially, storage. This design has several advantages, but it is not without some drawbacks. Some of the most appealing aspects of breaking up an application's functionalities into microservices are [3–5]:

- *Scalability:* Each component of the application is now isolated into a microservice that can be individually scaled to accommodate varying loads. By scaling only the required microservice instead of the whole application, developers can save resources.
- *Team Independence:* The existence of individually deployable services allows easier separation between components. This makes individual and parallel development of each application component possible by different teams, speeding up development cycles.
- *Robustness:* Service independence allows faults and failures to be contained to a single microservice environment, avoiding its propagation to the application as a whole, increasing robustness, although application functionalities may be limited during said failures.
- *Technology Diversity:* Each service has its own code base, and as such, the technologies chosen for each service may be different, allowing developers to choose the technologies that best fit its requirements. Moreover, it allows individual microservices to evolve with time, changing technologies over time without affecting the remaining system, allowing for easier maintenance and continuous development.

However, an application developed as several loosely coupled services has several complications, like [3–5]:

- *Operation Costs:* Without proper development infrastructure and monitoring tools, the increased number of components of this architecture may be unmanageable for some organizations.
- *Data Sync across multiple services:* Multiple microservices may require access to the same set of data. To maintain microservice independence, shared data is required to be copied to each required microservice's data store, allowing data to be read and modified locally by each microservice, whilst in a monolithic architecture, data could be managed by a single database. This creates the need for data to be synchronized across microservices.
- *Consistency:* When designing a microservice application, the functionalities can span multiple microservices, creating space for interleavings. With increased business logic complexity, functionalities tend to increase in size, making the discovery of all possible interleavings more difficult. This means that it is more likely that the developer will fail to handle some of them, leading to anomalous behavior.

Microservice architectures have become popular and are often considered when developing new systems. However, microservice compositions can introduce complications that would not exist in a monolith or suffer from unforeseen complications with deployment and maintenance. Ultimately, in some cases, microservice applications may need to be reimplemented as monoliths to solve these problems [6]. Essentially, each application's domain determines whether the benefits of a microservice architecture are worth its downsides. One of the main indicators is the level of coupling within the application's functionalities. Tightly coupled functionalities that manipulate complex invariants are likely to cause invariant violations.

2.2 Domain-driven Design

Designing microservice applications is a difficult task when the application has complex business logic and tightly coupled functionalities [7]. One common way to do so is through a domain-driven design approach [8]. Central to this approach is the notion of *aggregate*: an aggregate is a cluster of objects that are tightly connected by the domain logic and are considered the atomic unit of the domain. The state of an aggregate is updated by the execution of functionalities, which are operations over aggregates. A key design challenge is to determine the optimal distribution of aggregates between microservices.

Aggregates capture relevant aspects of the domain by including data types, hierarchy, and relationships among objects. However, this information is often not sufficient to represent the domain's data entirely, as it does not contemplate any business logic that may constrain what values can be taken by each object. For this purpose, it is possible to express predicates over one or more aggregates that define a correct state. These predicates are called invariants [8]. If the invariant only involves data from one aggregate, then it is said to be an intra-invariant. Conversely, if it involves data from several aggregates, then it is called an inter-invariant. Another relevant distinction is that while intra-invariants must always be upheld for an aggregate to be consistent and guarantee

that the business logic is respected, inter-invariants can often be temporarily violated, without breaking the correctness of the application, as long as they are eventually satisfied. This second aspect is intrinsic to the expected eventual consistent model of the behavior of a microservices application.

Often, business logic requires different aggregates to share a given piece of data, requiring the performed changes to a given aggregate's data to be propagated to other aggregates. Note that each aggregate may have a different view of the same data and, thus, the multiple copies of the data may not be exact replicas of the same content. Propagation is often ensured by events [8]. Events are published by upstream aggregates and subscribed by downstream aggregates. An aggregate is considered downstream of another aggregate if it uses information from an upstream aggregate, while the opposite does not happen, essentially making dependencies one way, avoiding bidirectional and circular dependencies. As such, if relevant changes are made on an upstream aggregate, events will be used to propagate them downstream, preserving one-way dependencies.

To clarify these concepts and introduce an example domain that is used in the next sections, this work uses an application named Quizzes Tutor [9][1], a multiple choice quiz question tool for assessment and self-assessment that supports quiz answer tournaments.

Listing 1 presents the relevant entities of the Tournament and Course Execution aggregates from Quizzes Tutor, where the former is downstream of the latter.

```
Aggregate Tournament {
    Root Entity Tournament {
        Long id;
        DateTime startTime;
        DateTime endTime;
        Creator creator;
        List<Participant> participants;
        TournamentCourseExecution tournamentCourseExecution;
    }

    Entity Creator {
        Long id;
        String name;
    }

    Entity Participant {
        Long id;
        String name;
        DateTime registration;
    }

    Entity TournamentCourseExecution{
        Long courseExecutionId;
    }
}

Aggregate CourseExecution {
    Root Entity CourseExecution {
        Long id;
        Set<Student> students;
    }

    Entity Student {
        Long id;
        String name;
    }
}
```

Listing 1: Quizzes Tutor Aggregates

Listing 2 expands the Tournament aggregate by introducing some of its invariants. There are two intra-invariants: the first states that the start time must always be before the end time, and the second states that if the creator is enrolled as a participant, then their name in both roles must be the same. Furthermore, there are three inter-invariants. The first states that there must exist a course execution in the course execution aggregate

[1]Source code can be found at: https://github.com/socialsoftware/quizzes-tutor

that matches the course execution in the tournament. The second, that the name of a tournament's creator must be synced with their name in the course execution. And the third is that the name of each of the tournament's participants must be synced with their respective names in the course execution.

```
Aggregate Tournament {
    IntraInvariants {
        root.startTime < root.endTime;
        root.participants.filter(p -> p.number == root.creator.number).allMatch(p -> p.name ==
     root.creator.name);
    }

    InterInvariants {
        exists(CourseExecution : ce -> ce.id == root.tournamentCourseExecution.
    courseExecutionId)
        CourseExecution.get(root.tournamentCourseExecution.courseExecutionId).students.filter(
    sc -> sc.id == root.creator.id).allMatch(sc -> sc.name == root.creator.name)
        root.students.foreach(st -> CourseExecution.get(root.tournamentCourseExecution.
    courseExecutionId).students.filter(sc -> sc.id == st.id).allMatch(sc -> sc.name == st.name
    ))
    }
}
```

Listing 2: Tournament aggregate's Invariants

Tournaments are created by students, and their participants are also students. More precisely, the tournament's creator and participants are a subset of the students in the course execution. A consequence of this business logic is that the same information, the student's id and name, is held by more than one aggregate.

Figures 1 and 2 present two functionalities of Quizzes Tutor: update student name and enroll student in tournament, which are responsible for changing the student's name and enrolling a student in a tournament, respectively. The first one is also an example of a data change being made in an upstream aggregate, Course Execution, that is propagated downstream to the Tournament aggregate via an event emission. Therefore, in Figure 1 the execution of the update name transaction in the Tournament microservice is triggered by the emission of the update name event.

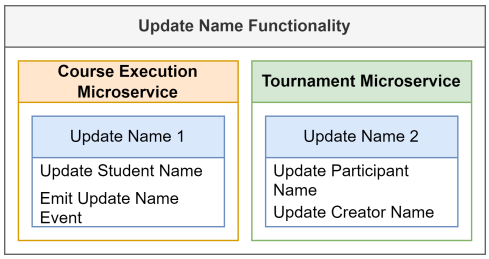

Figure 1: Update Student Name Functionality

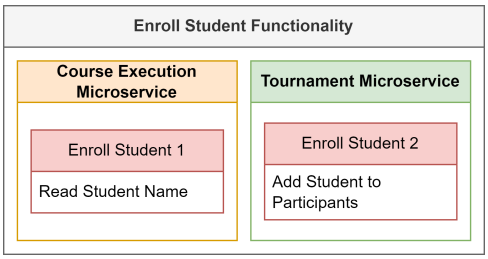

Figure 2: Enroll Student Functionality

2.3 Invariant Violations

There are two main possible causes for invariant violations to occur in a microservice composition. The first is the existence of bugs in the code that implements a functionality that, under some combination of system state and inputs, may drive the application to a state when there is an invariant violation. Such a violation of invariants could happen even in a monolithic implementation. The second is the occurrence of anomalies during the concurrent execution of functionalities, due to the interleavings that are made possible by the microservice composition. Both of these cases are relevant to our work and are discussed in Section 5.

Furthermore, not all invariant violations are the same. Some may be more relevant to the application logic than others because of the invariants breached in each case. As mentioned in the previous Section 2.2, not all invariants need to be upheld at all times, some may be relaxed during the intermediate state of functionalities and only need to eventually be met in quiescence states. As such, this work defines two invariant categories, Absolute Invariants and Eventual Invariants. The first must always be upheld, and any state when they are not

will be considered a violation. The second can not be upheld in intermediate states, but must be upheld after all the functionalities manipulating it have terminated, meaning that these will only cause violations if they are breached in a state when no functionality is manipulating data related to them.

2.4 Useful Requirements for Invariant Violation Detection Tools

Given the problem at hand, the following requirements are considered useful for a tool attempting to fulfill the goals of this work:

- *Verify Microservices Invariants*: The tool must be designed to verify microservices invariants, preferably using a domain driven design-like representation.
- *Make A Complete Analysis*: The tool should detect all possible invariant violations that may occur in the application.
- *Analyze Source Code*: The tool should directly analyze the source code of the application and not an abstract representation of its logic.
- *Distinguishing Between Eventual and Absolute Invariants*: The tool should provide syntax to distinguish between eventual and absolute invariants, as defined in Section 2.3
- *Generate Test Cases*: The tool should provide a way to reproduce the discovered anomaly violations, ideally through the generation of test cases that reproduce them.

3 Related Work

There are two main distinct ways to analyze the execution of an application, namely by following a black box approach or by following a white box approach. The difference between the two is that while the first focuses only on the effects of an execution, having no knowledge of the application's inner workings, the second one also covers the internal operation of the application. Both approaches have advantages, as discussed in Section 4.4. However, the use of white box tools might not always be possible, namely if the source code of an application is not accessible. Tools following white box approach are discussed in Section 3.1 and tools following black box approach in Section 3.2, the tools in both groups are designed for or related to invariant analysis. Furthermore, this work also addresses the generation of tests for microservices applications to aid their development, thus Section 3.3 covers tools related to the development and testing of microservices.

3.1 White box Analysis Tools

3.1.1 Harmony

Harmony [10] is a white box tool designed to detect errors in concurrent programming by finding executions where the application's invariants are breached. Harmony performs its analysis by model checking the representation of an application and its invariants in HarmonyLang, which is a Python-like language designed to simplify the analysis of concurrent programming applications. As such, to use this tool, the application under test must be translated manually into HarmonyLang. This tool discovers any possible execution sequence in which at least one of the invariants is breached, presenting them in a graph that displays the smallest path from the application initial state to the state where the invariant breach occurred.

Due to its design and the use of a model checker, this tool provides a complete analysis of the representation of the application under test application. It is also important to mention that HarmonyLang does not support the full syntax of the many programming languages, for instance, it has no support for classes. Lastly, since this tool uses a model checker, the time required to run complex scenarios is significant, or it may not finish within the useful period.

Discovering invariant breaches in concurrent programming has similarities to discovering invariant violations in microservices. This is explored further in Section 4. Although Harmony does not support all the desired features for our work, it may prove useful as a starting point for a new tool.

3.1.2 Alloy

Alloy [11] is a modeling language designed to help developers verify the properties of their domain. More specifically, the Alloy Analyzer is a solver that, given a set of domain constraints, produces a sample instance that meets the constraints if one can be generated. In addition to constraint, Alloy supports the validation of invariants and supports the analysis of methods, that is, it can verify if the execution of a method or methods can lead to invariant breaches or if the methods can be executed given the domain constraints.

The versatility of this tool has led to its use in several domain verification tools, but most of the research is focused on the verification of domain representations like UML, such as [12, 13] due to the similarity between the two [14], which makes translation easier than coming from an application source code.

Alloy's model checking abilities combined with extensive syntax for describing domain entities and invariants could be used to represent a microservice application and discover invariant violations, fulfilling the goal of this work. This option is discussed in Section 4.

3.1.3 Ucheck

Ucheck [15] is a runtime-static hybrid white box tool that provides invariant verification during runtime based on traffic between microservices. More specifically, this tool uses a model for each microservice along with its invariants for two things. First, it validates if the models verify the invariants and second, it uses them to determine validation criteria for the messages exchanged between the microservices during runtime.

Essentially operating in three stages, the first two are statically validating the application and determining what is a valid message, and the third occurs during runtime, when the application's communication is validated against the previously defined rules. Any message that goes against the rules is flagged and dropped. Ucheck's hybrid approach to invariant validation could prove insightful for this work, because it provides a complete analysis of a microservice application's invariants, considering the domain entities.

3.1.4 MAD

MAD [16] is a white box tool designed to assist developers in assessing the ease with which a certain monolithic application can be converted to a microservice architecture, given a certain decomposition of the original domain, by indicating the number and types of data anomalies that will arise from applying a given decomposition.

More specifically, MAD takes as input a representation of a monolithic application and a desired decomposition of its functionalities into microservices, which are then translated to Z3[2] statements that combined with it's anomaly finding engine lead to the discovery of execution sequences that generate anomalies, which are then categorized as Dirty Reads, Dirty Writes, Lost Updates, Write Skews, Read Skews based on their structure or Other if it is too complex to categorize. The use of a SAT solver in this tool allows for a complete analysis, discovering all possible data anomalies in the application with the trade-off of long execution times and the risk that it may not finish within the useful period.

Although MAD does not support invariants, the detection of data anomalies can be useful in the discovery process of invariant violations, as discussed in Section 4.

3.2 Black Box Analysis Tools

3.2.1 HawkEDA

HawkEDA [17] is a black box tool designed to help developers weigh the advantages of microservice implementations and assess the number of data integrity anomalies for different workloads. The user is required to provide a description of the application's interface and the relevant invariants, described in terms of calls to the interface, along with some configuration parameters for the intensity of the scenario, which entail, among others, the skewness and frequency of requests. With that, the tool runs the application in a controlled environment, providing performance metrics such as average latency and throughput, along with any invariant breach it might have encountered. Requests are randomized according to the configuration parameters, so there are no guarantees of the analysis completeness.

[2] Z3 Theorem Prover is a SAT solver developed by Microsoft: https://www.microsoft.com/en-us/research/project/z3-3/

This tool provides insights on a way to represent invariants different from the typical domain driven design approach that might be useful for some applications. Furthermore, the combination of random testing with invariant verification used by this tool is different from those seen before and may be a relevant alternative to the use of SAT solvers or model checkers.

3.2.2 PETIT

PETIT [18] is a tool designed to test microservice applications provided as a black box, for example, to test a third-party API, the implementation of which is unknown. The tool is introduced along with a specification language to annotate APIs called APOSTL which should be used to annotate the API under test with the invariant that defines the correct operation of that API. This tool takes the definition of an API along with rules that define the correct behaviors of each API method and performs its analysis without knowing the internal components of the application, generating inputs randomly or according to some user-provided pattern.

This tool follows a procedure similar to HawkEDA, making it relevant for the same reasons. However, it adds to the previous tool by providing a more rich and structured way to define invariants at the API level using the APOSTL language.

3.3 Testing and Development Tools

3.3.1 Transactional Causal Consistent Simulator

The Transactional Causal Consistent Simulator [19] is a tool designed to allow the development of Transactional Causal Consistent microservice applications in a user-friendly environment, providing a starting point for their implementation and a friendly environment for testing. The simulator provides the boiler plate code for the implementation of a causal consistent aggregate. These aggregates have versioning control and support the definition of methods to merge them in case of version conflicts. However, it is up to the developer to make use of these classes, implement the remaining logic of their application, and the required merge procedures, along with any desired test cases.

The simulator does not make any analysis of the application code and relies on user-generated code and test cases to operate. As such, it does not find any invariant violations on its own, but instead empowers the user to find and resolve violations themselves. Never the less, it exemplifies what useful test cases for a microservice application using a single technology look like.

3.3.2 Jolie & JoT

Jolie [20] is a service-oriented programming language that focuses on providing syntax for the development and composition of services facilitating the interoperability of Services provided in different technologies, while also introducing a new way to build services that is service-oriented. This is done by providing a completely new grammar built on top of Java with database connectivity options, concurrent programming clauses, and other necessary tools for the purpose. In addition, it provides seamless interoperability with other services deployed in different languages.

For this work, a relevant tool built on top of Jolie is JoT [21], which is a testing framework that allows developers to test all components of their microservice applications regardless of their individual technologies, due to its use of Jolie.

This framework does not make any analysis of the application code and is only a tool for users to write test cases that can span across multiple microservices implemented in different technologies, as such it does not find any violations on its own, but instead empowers the user to find and test violations themselves, like the previously presented simulator. However, unlike the simulator, this framework allows the development of test cases involving microservices using different technologies.

3.4 Tool Comparison and Discussion

To sum up the most relevant characteristics of the tools presented in the previous section, their characteristics are presented in Table 1.

Tool	Black/White Box	Invariants	Complete Analysis	Source Code Analysis
HawkEDA	Black Box	API Level	No	Yes
PETIT	Black Box	API Level	No	Yes
Simulator	White Box	Entity Level	No	Yes
JoT	White Box	Entity Level	No	Yes
Ucheck	White Box	Entity Level	Yes	No
Harmony	White Box	Entity level	Yes	No
Alloy	White Box	Entity level	Yes	No
MAD	White Box	Not Supported	Yes	Yes

Table 1: Tool Comparison

By definition, invariants are correctness constraints over the application's domain, which are often defined in terms of domain entities. As such, representing them at the entity level is simpler, albeit they can also be defined at the API level, through methods that interact with the entities. This leads to invariants defined at the API level, requiring several methods to be called to validate the invariant, whereas invariants defined at the entity level can be validated by analyzing the entities directly, making entity level representation more efficient.

Harmony [10], Ucheck [15], and Alloy [11] define invariants at the entity level, while HawkEDA [17] and PETIT [18] define them at the API level. As a consequence, the representation of the same invariant in the first group is closer to the domain representation of the same invariant, also being smaller and simpler, as it can directly access the entities involved.

Black box approaches lack knowledge regarding the application internals, such as entities, and therefore are unable to define invariants at the entity level. This puts them at an advantage compared to white box approaches. Furthermore, black box approaches are unable to achieve a complete analysis, as they are based on testing approaches. Due to both of these factors, black box tools are less interesting for this work as they are inherently further away from its goals than white box tools.

The Simulator [19] and JoT [21] are white box tools. However, as they require the user to manually create all the test cases required to provide a complete analysis, their invariant violation detection capabilities are also not as interesting for this work as the remaining white box tools. Nevertheless, they are relevant for this work for their testing environments and techniques, as is shown later in this Section.

Ucheck stands out from the remaining white box tools because it operates both at a static level, to discover possible invariant violations in the domain model and infer what are correct behaviors during runtime and at runtime to prevent the occurrence of said incorrect behaviors. Meaning that while it shares some similarities with this work, it acts more like an invariant violation countermeasure tool than a detection tool.

With all this in mind, the tools that are more related to the goal of detecting all invariant violations in an application are MAD, Harmony, and Alloy. Within these, it is worth highlighting that MAD is the only one that supports source code analysis and does not require a manually generated model of the application. The capabilities of this tool and potential application as a solution for the problem of this work are discussed in Section 4.

Another relevant aspect for this work is automatic test case generation from the discovered violations. However, depending on the technologies used in the application under test, different testing techniques may be required. One possible solution is to support multiple technologies and generate technology-specific test cases, such as those in Simulator [19], another would be to use a generic framework like Jot [21] that can be used regardless of the technologies used by each microservice. Both approaches are not without limitations. The first allows for more in-depth testing of each individual microservice at the cost of making tests that span multiple microservices complex, while the second makes high-level tests spanning across several microservices simple at the cost of more complex configuration and forcing testing to be made over remote procedure calls, making the tests slower and more complex.

4 Discovering Invariant Violations Using Previous Work

To clarify what Harmony [10], MAD [16], and Alloy [11] (the tools highlighted in the previous Section 3.4) are already capable of achieving, what their limitations are, and which of their features can be repurposed for this work, this section discusses their use to discover violations.

4.1 Discovering Invariant Violations Using Harmony

Harmony [10] is designed to discover scenarios in which invariants are violated in a concurrent programming setting, much like what this work aims to achieve. To use Harmony for the purpose of this work, functionalities have to be translated into HarmonyLang functions, domain invariants represented as Harmony invariants, and aggregates represented as Harmony variables. Performing this translation requires some engineering work due to disparities between the technologies, but the logic of the application remains the same. However, to correctly modulate this work's problem, order within the transactions that make up a functionality has to be enforced. To do so, the created functions call each other in the correct order, ensuring that the functionalities are executed according to the specification. After the whole domain is translated, Harmony requires that the possible input for each function be bounded, methods for random input values are provided and could be used. Furthermore, indications of which function are going to execute in parallel and how many instances of each have to be provided, an adequate value would be two instances of each functionality. Using the procedure just described, Harmony will report any execution in which an invariant can be breached and for what values.

However, Harmony does not possess any syntax to indicate when an invariant should be verified. As such, when Harmony is used, it is not possible to make the distinction between eventual invariants and absolute invariants, as described in Section 2.3. This means that using Harmony for the purpose of this work will lead to the report of a significant number of violations that are not relevant. Harmony will report every intermediate state where an eventual invariant is breached, including scenarios when, it the final state, the invariant is upheld. Given that this tool is not available for expansion, this is not a limitation that could be overcome, and this also means that there would be little space for future improvements.

4.2 Discovering Invariant Violations Using MAD

MAD [16] is not designed for invariant analysis, its goal is the discovery of data anomalies. As such, to discover invariant breaches, a parallel between invariant breaches and data anomalies is required. Our research revealed that looking for invariant breaches in cases where there are data anomalies yields very good results. Therefore, to use MAD for the purpose of this work, the application should be analyzed by MAD, and then the discovered scenarios where there are data anomalies, analyzed. Eventual invariants should be tested at the end of each scenario and the absolute invariant at each step to look for possible invariant violations. Note that invariants only need to be tested in scenarios where they could be affected, that is, in scenarios where the data anomalies involve functionalities that manipulate entities relevant to the invariants. Although this procedure did not automatically discover any invariant breaches, running MAD was useful in narrowing the search. Experimenting with this approach led to the design of a potential approach to this problem, which is presented in Section 5.4. Furthermore, MAD does not require any manual modulation of the application, because it automatically translates the application's source code. This introduces a significant reduction in the analysis time, while also removing space for human error. Contrary to Harmony, MAD's features are available for repurposing or expanding, making meaning that the translation layer and the internal representation used by MAD may be used as ground work for a new tool or to automate the translation of source code to some other tool, like Harmony or Alloy.

4.3 Discovering Invariant Violations Using Alloy

Alloy [11] is designed to validate the integrity of domains. However, it is not designed for direct analysis of source code. So, to use Alloy for the detection of an application's possible invariant violations, it is necessary to translate the aggregates to alloy signatures and represent any relationships as alloy facts. Invariants can then be modulated as Alloy asserts over the created signatures. Lastly, each transaction of the functionalities should be represented as a predicate. This modulation is not trivial, as there are significant differences between the

syntax of Alloy and that of an average programming language. Alloy is designed so that the representation of the consequences of executing a transaction is simple and not to represent the full procedure of a transaction. However, just representing the consequences of a transaction and not the exact procedure can lead to incorrect representation of the application's implementation and miss invariant violations that occur not due to domain design flaws, but due to implementation specifics. For clarification, consider a functionality that updates an entry on a list. The implementation of that functionality may iterate over the entire list performing several validations before finally updating the desired entry. The representation of this functionality in Alloy could miss these validations as they are only a byproduct of the implementation and not feature, however, they may still be relevant for invariant violations.

To sum up, Alloy will produce executions where there are assert breaches, that is, where there are invariant violations, but it will only produce those that arise from domain design flaws, falling short of the objectives of this work, by not covering all invariant violations. At the same time, it also does not provide any clause that could represent the distinction between eventual invariants and absolute invariants, meaning that this would have to be represented some other way. During this study, no representation was discovered that could solve this problem.

4.4 Takeaways

Harmony is able to discover invariant violations in a microservice application. However, it does not provide syntax to distinguish eventual invariants from absolute invariants, and adding such clauses is impossible. That means that even if an automatic way to translate an application's source code to HarmonyLang the tool would still not meet all the invariant violation detection requirements for this work.

MAD is not capable of detecting any invariant violation. Detects anomalies in the data that may lead to invariant violations in some cases. However, its components for automatically translating and modulating source code are very useful to achieve this work's goals.

Alloy is capable of discovering domain inconsistencies. However, it is not fully capable of representing the implementation of that domain and, as such, fails to detect all possible invariant violations that may occur in an application. Furthermore, it also does not provide clauses to represent eventual invariants separately from absolute invariants.

As such, MAD [16], Harmony [10], and Alloy [11] all lack some aspect to achieve the invariant violation detection goals of this work. Furthermore, none of them provides anything related to test case generation, which is also a requirement for this work. In summary, the requirements for the new tool and what these tools already accomplish are as follows.

- *Verify Microservices Invariants*: Alloy and Harmony do this, while MAD does not.
- *Make A Complete Analysis*: All these tools do it.
- *Analyze Source Code*: MAD is the only that performs source code analysis
- *Distinguishing Between Eventual and Absolute Invariants*: None of the tools provide this feature, nor a way to emulate it.
- *Generate Test Cases*: None of the tools generates test cases.

For these reasons, this work will produce a new tool that will achieve all of its goals, using some of MAD's components to automatically analyze source code. It does not rely on the user's ability to model their application to provide accurate results, but instead analyzes the source code directly. Furthermore, taking inspiration from these tools, the new tool will use a formal verification technique to grant a complete analysis of the application. Given that MAD's internal representation is already optimized for SAT solvers, this work will also follow that approach. This tool is presented in the next Section 5.

5 Design

The objective of the tool produced by this work is the discovery and reproduction of invariant violations, making a distinction between eventual invariants and absolute invariants, as defined in Section 2.3.

An overview of the tool is presented in Section 5.1, followed by an in-depth presentation in Section 5.2. Then, Section 5.3 clarifies what technologies will be used during development. Lastly, a potential approach to reduce the search space of the problem along with its trade-offs is discussed in Section 5.4.

5.1 Tool Overview

To discover all possible invariant violations in an application, the tool produced by this work will formulate a satisfiability problem directly from the application source code that can be interpreted as: Can there be any input, initial state, and execution order of the provided functionalities where at least one of the absolute invariants is breached at any point or at least one of the eventual invariants is breached in the final state? This formulation will then be run on an SAT solver and, if any solution is produced, then a violation was discovered. With the discovered violations, the tool will generate a visualization of the violation and a test case skeleton that will reproduce it.

5.2 Tool Design & Operation

The tool will take as input the source code of a microservice application and its invariants, which will then be translated into an internal representation that is ready for direct translation to SAT statements used to build a model in conjunction with the violation discovery engine. The violation discovery engine is the set of SAT statements that do not depend on the application under test but constrain the formulation, essentially defining the problem. Once the analysis is completed, the tool will generate parameterizations for inputs and initial states that trigger that cause the violations and display them in a graphical manner, as well as a skeleton for a test case that can be used to reproduce it.

Given the explored tools, the desire for direct analysis of the application's source code and the introduction of the eventual and absolute invariants, it was decided that the produced tool would use MAD's [16] source code parser, expanding as needed to fit our needs, using their internal representation, which is ready to build SAT statements as ground work to build this work's new SAT formulation. Other possible approaches would be to create a new parser for source code and translate it to HarmonyLang or Alloy, modulating the application in their respective syntax validating if the model could ever lead to an invariant violation. The first approach was chosen because repurposing MAD's parser and internal representation allows this work to explore the newly defined eventual and absolute invariants, a concept which is not supported in Alloy nor Harmony, where invariants are always verified at each step. Another factor that leads to Alloy not being chosen is the need to order the execution of the functionality parts. Furthermore, building on top of MAD allows this work to jump directly to formulating the core problem instead of having to build a new parser and internal representation for HarmonyLang or Alloy which would take up a lot of time from the already limited duration of this project.

This subsection details the design and operation of each of the tools' components. Figure 3 represents the tool pipeline, and the following sections detail each mentioned component.

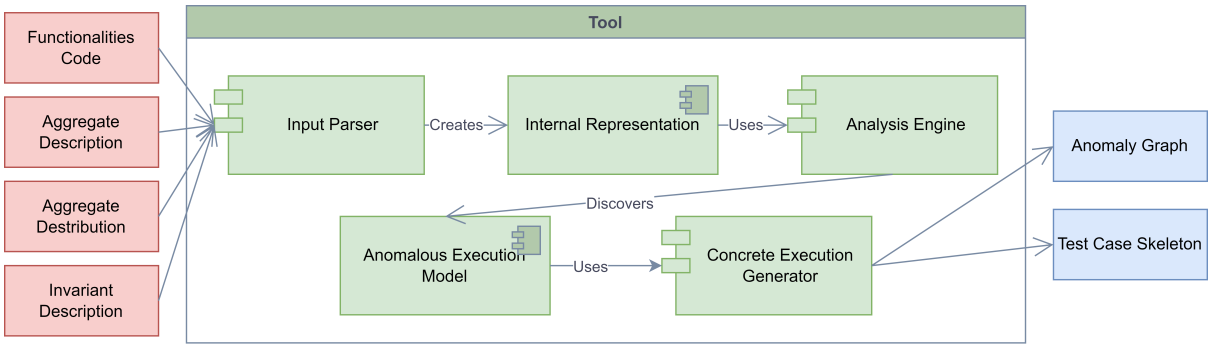

Figure 3: Tool Pipeline

5.2.1 Input

To analyze an application with this tool, the user should know the source code of each of the application's functionalities along with the description of the entities belonging to each of the aggregates and the distribution of the aggregates by the microservices. Furthermore, the applications' invariants should also be provided as logical clauses relating to the aggregate entities and categorized as eventual or absolute. These representations must be complete and cover all of the application logic, as they are the only knowledge of the business logic the tools will have. Often microservice applications make use of different technologies and programming languages in each microservice, however, there is no generic way to interpret all of them. As such, the internal representation of the application will be generic and technology independent, but the input translation will always be technology dependent. To accommodate this, the tool will support the implementation of several parsers to support the required technologies, working in parallel to broaden the range of technologies supported by the tools.

5.2.2 Internal Representation

To facilitate the representation of the application as SAT statements, this tool will adapt MAD's internal representation of functionalities to represent microservice functionalities instead of their current representation as the decomposition of monolith functionalities. Essentially, it will represent the logic of each functionality and how they relate to each other. The internal representation of a functionality will contain an ordered list of its transactions which in turn contain which entity they manipulate and how. Furthermore, this representation will be expanded to accommodate the representation of aggregates: their entities, distribution among the microservices, and which functionalities manipulate them. Lastly, it will also represent invariants as conditions over the entities of the aggregates involved and distinguish eventual invariants from absolute invariants. The internal representation exists not only to simplify the representation of the domain under test to SAT statements, but also to make the source code parsing independent of the SAT representation and vice versa. This allows multiple parsers to be implemented without any change to the core logic of the tool, which is the SAT encoding. One more relevant aspect of the internal representation is that it simplifies the process of matching each invariant with the functionalities that manipulate it. This information is derived, during parsing, from crossing which aggregate's entities are manipulated by each functionality with which entities are involved in each invariant. Depending on the implementation's specifics, several approaches can be used to convert the input into the internal representation, but existing parsing tools will be used whenever possible, both to avoid bugs and because parsing is not the focus of this work.

5.2.3 Engine For Detection Invariant Violations

This engine will attempt to find a possible execution order for the current set of functionalities, along with an initial state and input that will cause an invariant violation. Considering this, the high-level operation will be as follows: There must be a state when the invariant is breached. This state must be the result of the execution of a transaction. There must exist a valid initial state and input combination that make the execution of that functionality generate the inconsistent state. The analyze distinguishes eventual invariants from absolute invariants

by restring the state when an eventual invariant violation is detected to final states, that is, the state after all the involved functionalities have finished. More specifically, the pseudo-SAT statements that make up this engine are as follows:

```
(forall (f1 functionality)
    ;this clause will be specified for each part of the functionalities, to include the requeried preconditions, in addition to removing the option of the starting state being the initial state for the functionality parts where that should not happen
    (and
        (type fp1 some_functionality_part)
        (exists (s1 state)
            (and
                (=>
                    (start fp1 s1)
                    (or
                        (exists (fp2 functionality_part)
                            (result fp2 s1)
                        )
                        (initial s1)
                )
                (;preconditions)
        )
    )
)

(forall (s1 state)
    (or
        (and
            (exists (fp1 functionality_part)
                (and
                    (result fp1 s1)
                    (or
                        ;there will be a clause defining the postconditions for each functionality_part
                        (=> (type fp1 some_functionalit_party)
                            (;postconditions)
                        )
                    )
                )
            )
        )
        (initial s1)
    )
)

//Absolute Invariant violation detection clause
(exists ((s1 State))
    (=>
        (and
            (not (initial s1))
            (;absolute invariant pre conditions)
        )
        (not
            (;absolute invariant post conditions)
        )
    )
)
//Eventual Invariant violation detection clause
(exists ((s1 State))
    (=>
        (and
            (final s1)
            (;eventual invariant pre conditions)
        )
        (not
            (;eventual invariant post conditions)
```

```
            )
        )
)
```

Listing 3: Aproach 2 - Engine pseudo SAT statements

Note that additionally to the presented clauses, there will be, for each functionality, a clause forcing its starting state to be correct and characterizing the resulting state. Furthermore, there will be a clause forcing the order of the functionality parts to be correct by defining a chain of states.

5.2.4 Generating Concrete Executions

The result of the operation of the engine over the representation of the application will be a set of SAT statements that represent an execution in which an invariant was violated, detailing dependencies between transactions and the path taken on any branching conditions. However, these statements require translation into constraints that developers can understand. In other words, this component is responsible for deriving which functionalities were invoked in what order, the parameterization of the inputs required, along with the parameterization of the initial state. The order is obtained directly by matching the resulting state of a functionality with the start state of another. The inputs required for each functionality and the initial state of the aggregates involved are derived from the branching conditions produced by the engine. The conditions will be SAT statements that indicate that the values taken by some entity in some aggregate should obey some logical statement. For clarification, a statement like that could translate to: "entity A must be greater than X or entity C must have the same value as entity B". Crossing all these conditions should bound most variables, forming the parameterization for the inputs and initial state for the scenario. If some initial state or input variable is not bound by any constraint, then the outcome is independent of that variable/entity, and any value can be taken.

5.2.5 Output

The discovered violation is stored in the tool, independently of the way it will be outputted, so several representations can be implemented without changing the logic of the tool. In an initial stage, two will be developed, a graph view and a test case skeleton that attempts to be technology independent.

To display the results in a useful way that makes the violations clear to the developer, the tool will use the constraints generated by the previous module to generate graphical representations of the execution and parameters that constitute the violation. This representation will be done in a graph that represents the initial state of the involved aggregates, the order of execution of the involved transactions, and their inputs. The input and initial states will be presented in terms of parameterizations. It will also include the breached invariants. The following Figure 4 presents an example of what it will look like.

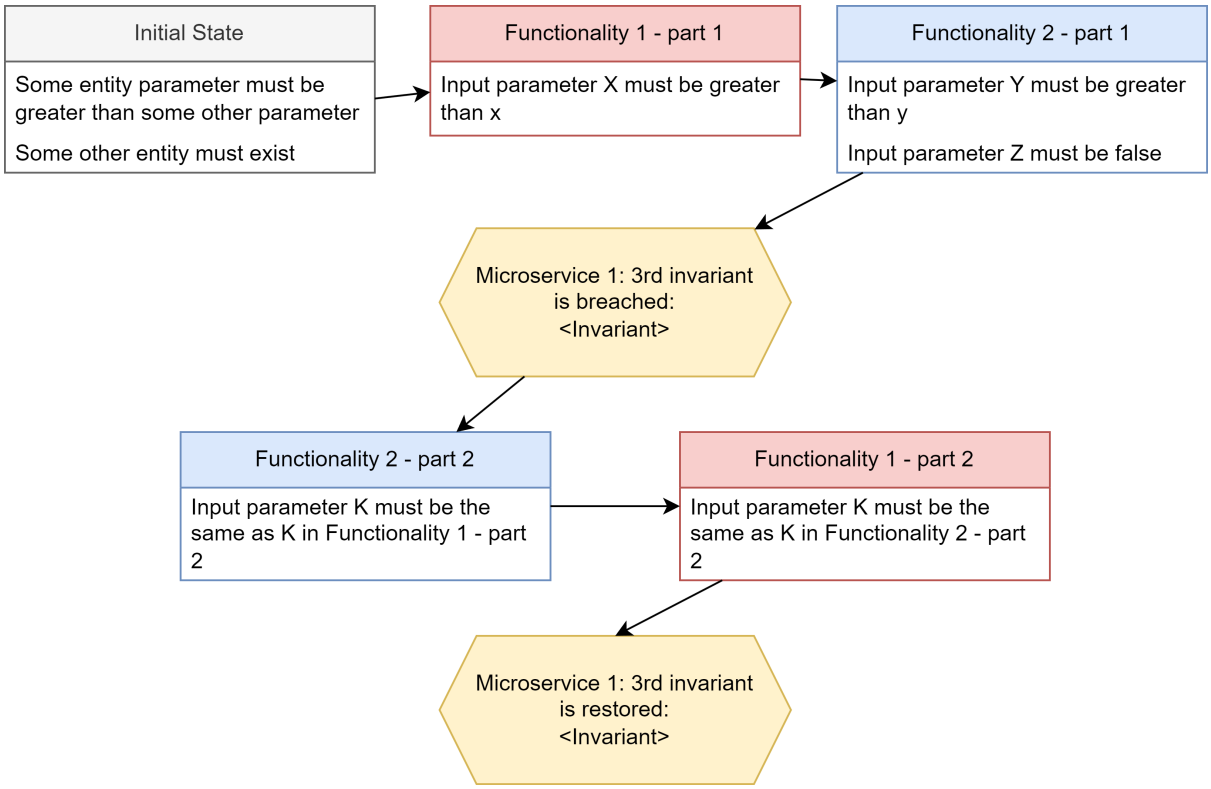

Figure 4: Output - Violation Graph

Note that this Figure 4 shows a violation where the involved invariant was an absolute invariant. This can be identified because the invariant violation is detected in an intermediate step. If the invariant were eventual, there would not be a violation because in the final state the invariant is restored, and eventual invariants only cause violations if they are breached in the final state.

Furthermore, the tool will also generate a skeleton for a test case that reproduces the discovered violation, again using the discovered execution order and parameterizations. Given that the different modules may be implemented using different technologies, it is not possible to generate a concrete test case for the discovered violations. A middle ground could be using JoT, the Jolie test framework presented in the Related Work Section 3, however, this may not interest all users as it requires plenty of configuration to operate, and it may require changes on the implementations of each microservice to allow the individual execution of each involved transaction. Considering this, it was decided that the tool will produce a skeleton for test cases that uses pseudo-code notation to set the initial state and invoke each transaction in order, with the adequate inputs. Listing 4 presents an example of what it will look like.

```
\\Set Initial State
    \\entity A parameter 1 be greater than parameter 2
    \\add entity B to entity list

\\Execute Functionality 1 - part 1
    \\parameter X must be greater than 3

\\Verify that invariant Microservice 1 - 3rd invariant still holds.

\\Execute Functionality 2 - part 1
    \\parameter Y must be greater than 2
    \\parameter Z must be false

\\Verify that invariant Microservice 1 - 3rd invariant still holds.

```

```
\\Execute Functionality 2 - part 2
    \\parameter K must be the same as K in Functionality 1 - part 2

\\Verify that invariant Microservice 1 - 3rd invariant is breached.

\\Execute Functionality 1 - part 2
    \\parameter K must be the same as K in Functionality 2 - part 2
```

Listing 4: Output - Test case Skeleton

5.3 Technologies to Be Used When Developing the Prototype

Some technology choices had to be made for the prototype that will be implemented in this work. Specifically, the code representation of the functionalities will have to be provided in Java. To simplify and avoid bugs in the parsing and translation work, a Java code parser will be used: Soot [3]. Some components of the language may not be fully incorporated, depending on the available time. Most noticeably, classes will not be modulated, and the aggregates should be described in SQL. As such, all interactions over aggregates should be done with SQL statements. Furthermore, aggregate structure rules, apart from invariants, should be included as SQL constraints. The distribution of aggregates by the microservices should be provided in a JSON file that also includes the invariants related to each aggregate and separates them as eventual and absolute, described using SQL-like syntax. A complete example of an input for the tool is as follows:

```
CREATE TABLE Students (
    id                  INT NOT NULL,
    name                VARCHAR NOT NULL,

    PRIMARY KEY (id)
);

CREATE TABLE Tournaments (
    id                  INT NOT NULL,
    creator_id          INT NOT NULL,
    creator_name        VARCHAR NOT NULL,
    start_time          INT NOT NULL,
    end_time            INT NOT NULL,
    course_execution    INT NOT NULL,

    PRIMARY KEY (id)
    Foreign Key (creator_id) REFERENCES Students(id)
);

CREATE TABLE Tournament_Participants (
    tournament_id       INT NOT NULL,
    participant_id      INT NOT NULL,
    participant_name    VARCHAR NOT NULL,
    registration_time   INT NOT NULL,

    CONSTRAINT PK_Tournament_Participants PRIMARY KEY (tournament_id, participant_id)
    Foreign Key (tournament_id) REFERENCES Tournaments(id)
    Foreign Key (participant_id) REFERENCES Students(id)
);
```

Listing 5: Input - Aggregate Description

```
import java.sql.*;

public class qt_example {
    private Connection connect = null;

    public qt_tournament_creator() {
        //Connection to Database
```

[3]Soot is a Java parsing framework: https://soot-oss.github.io/SootUp/v1.1.2/

```
    }

    public void update_name_1(int student_id, String new_name) throws SQLException {
        PreparedStatement stmt1 = connect.prepareStatement("UPDATE Students SET name = ? WHERE
     id = ?");
        stmt1.setString(1, new_name);
        stmt1.setInt(2, student_id);
        stmt1.executeUpdate();
    }
    public void update_name_2(int student_id, String new_name) throws SQLException {
        PreparedStatement stmt2 = connect.prepareStatement("UPDATE Tournaments SET
    creator_name = ? WHERE creator_id = ?");
        stmt2.setString(1, new_name);
        stmt2.setInt(2, student_id);
        stmt2.executeUpdate();

        PreparedStatement stmt3 = connect.prepareStatement("UPDATE Tournament_Participants SET
     participant_name = ? WHERE participant_id = ?");
        stmt3.setString(1, new_name);
        stmt3.setInt(2, student_id);
        stmt3.executeUpdate();

    }

    public void enroll_student_1(int student_id, int tournament_id) throws SQLException {
        PreparedStatement stmt1 = connect.prepareStatement("SELECT name FROM Students WHERE id
     = ?");
        stmt1.setInt(1, student_id);
        ResultSet rs1 = stmt1.executeQuery();
        rs1.next();
        String student_name = rs1.getString("name");
    }
    public void enroll_student_2(int student_id, int tournament_id, String student_name)
    throws SQLException {
        PreparedStatement stmt2 = connect.prepareStatement("SELECT * FROM
    Tournament_Participants WHERE tournament_id = ? AND participant_id = ?");
        stmt2.setInt(1, tournament_id);
        stmt2.setInt(2, student_id);
        ResultSet rs = stmt2.executeQuery();

        if (rs.next()){
            PreparedStatement stmt3 = connect.prepareStatement
                ("Insert INTO Tournament_Participants (tournament_id, participant_id,
    participant_name) VALUES (?, ?, ?)");
            stmt3.setInt(1, tournament_id);
            stmt3.setInt(2, student_id);
            stmt3.setString(3, student_name);
            stmt3.executeUpdate();
        }
    }
}
```

Listing 6: Input - Functionalities Code File

```
{
    "M1": {
        "Tables": [
            "Student"
        ],
        "Absolute Invariants": [
        ],
        "Eventual Invariants": [
        ]
    },
    "M2": {
        "Tables": [
```

```
            "Tournaments",
            "Tournament_Participants"
        ],
        "Absolute Invariants": [
            "WHERE Tournaments.creator_id == Students.id THEN Tournaments.creator_name ==
    Students.name",
            "WHERE Tournament_Participants.participant_id == Students.id THEN
    Tournament_Participants.participant_name == Students.name"
        ],
        "Eventual Invariants": [
            "Tournaments.start_time < Tournaments.end_time",
            "WHERE Tournament_Participants.tournament_id == Tournaments.id AND
    Tournament_Participants.participant_id == Tournaments.creator_id THEN
    Tournament_Participants.participant_name == Tournaments.creator_name"
        ]
    }
}
```

Listing 7: Input - Aggregate Destribution and Invariants

Furthermore, the tool will use Z3 as a SAT solver. Its graphical output will be produced using Graphviz [4].

5.4 Potential Way to Reduce the Search Space

The proposed design covers all possible executions of the functionalities that manipulate the invariants. However, sequential executions of a functionality should, by definition, preserve the invariants. The correctness of the sequential execution of functionalities can be verified by unit testing, which is often done during the development of microservices. An execution in which the functionalities do not interleave or the interleaving does not cause any data anomaly[5] is equivalent to a sequential execution of these functionalities. Therefore, such executions would only violate an invariant if any of the functionalities do, meaning that if the functionalities are assumed to be correct, the only executions where invariants may be breached are those where data anomalies occur. Summing up, if an application's functionalities are presumed correct, then the set of executions where invariants can be violated can be reduced to the executions that suffer from data anomalies. Depending on the application, this can be a significant reduction.

For the Quizzes Tutor example, this procedure would reduce the search space from forty executions (all the possible execution flows of the functionalities, which are twenty, times the number of relevant sets of input initial state combinations, which are four) to only twelve (the number of execution flows which can cause data anomalies, which are three, times the number of relevant sets of input initial state combinations, which are four). Naturally, this is not the case for all domains, and this is a very simple example that only covers two functionalities spanning two microservices and involving three invariants. Nonetheless, it shows that this could be a very good optimization.

Although this reduction comes with two trade-offs. First, it assumes that the sequential or equivalent to sequential execution of functionalities cannot violate invariants, and second, it depends on a tool that will discover all possible data anomalies. One of such tools was covered in the related work 3, MAD [16].

As such, a potential way to reduce the search space at the cost of assuming that the functionalities are correct is to incorporate MAD's ability to discover data anomalies in the tool and only search the discovered executions that cause data anomalies for invariant violations, reducing the number of scenarios to test.

6 Evaluation

The evaluation of the tool produced by this work will cover the time it takes for the tool to analyze a given application and the impact that the invariant complexity has on the tool's performance. Furthermore, it will

[4] Graphviz is an open source graph visualization software: https://graphviz.org/

[5] This work adopts the definition of data anomalies in [22]

evaluate the completeness of the tool's source code translation. For this, real-world applications, like Quizzes Tutor, along with some popular examples, like the eShop[6] will be used.

6.1 Completeness of Source Code Representation

The tool's ability to represent the source code of the application under test without missing any relevant aspect that may lead to disparities in domain logic is key to an accurate analysis. So, to evaluate the completeness of the tool's source code representation, a comparison will be made between the logic of functionalities in the source code and their logic in the SAT formulations generated by the toll to evaluate if all the source code clauses are correctly translated.

6.2 Execution Time

Using a SAT solver can lead to long execution times for the tool, so the evaluation will also measure the execution time, establishing a parallel between the complexity of the application domain and the time it takes for the tool to analyze it. Metrics such as the number of microservices, the number of invariants per microservice, and the average number of transactions that manipulate each invariant will be included in the study. Furthermore, to evaluate the impact of the optimization proposed in Section 5.4, the same application will be verified using the optimization and not using it. Comparing the execution time of both runs will clarify if the optimization is useful despite its trade-offs.

6.3 Invariant Complexity Impact

It is expected that the complexity of an application's invariants will have an impact on the tool execution time. It is important to evaluate this relationship to determine potential focus points to improve the tool. As such, the evaluation will cover the analysis of the same application with invariants varying in complexity, so that the impact of the invariant complexity in the execution time can be measured. The invariant complexity will be measured in terms of how many entities and how many microservices it relates.

7 Scheduling of Future Work

This work will include writing a paper for submission to the Inforum Conference [7], considering this, future work is divided into and scheduled as follows:

- 2024 January 22 to 2024 May 03: Implement the proposed tool
- 2024 May 06 to 2024 June 07: Improve and test the tool
- 2024 May 13 to 2024 June 11: Write a Paper for Inforum
- 2024 June 11 to 2024 October 30: Write the MSc dissertation
- 2024 October 30: Deliver the MSc dissertation

8 Conclusion

The microservice architecture has become popular for the development of applications with business logic rich domains due to its numerous advantages over the more conventional monolithic architecture. However, designing microservice applications that correctly represent all the required business logic without any anomalous behavior is a complex task, due to the hardships of enforcing domain invariants in distributed environments where an application's functionalities may span several microservices.

[6] eShop is a reference .NET application implementing an eCommerce web site using a services-based architecture: https://github.com/dotnet/eShop

[7] https://inforum.org.pt/

The goal of this work is to help developers during the development of microservice applications by automating the detection and reproduction of invariant violations that may occur in their application. To do so, this work covers several existing tools, including both white and black box approaches and ranging from invariant validation to data anomaly discovery. After careful analysis, it was concluded that existing tools are still insufficient to detect all the possible invariant breaches an application may suffer from and that existing tools are not flexible enough to accommodate the developers' need to validate different invariants in specific application states.

Therefore, this work proposes a new tool for invariant violation detection and reproduction in business logic reach microservices that introduces different categories of invariants to be validated at different stages of the application execution.

Acknowledgments We are grateful to Rafael Soares, Tomás Pereira, and Valentim Romão 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).

Bibliography

[1] M. Fowler, "Microservices," Web page: http://martinfowler.com/articles/microservices.html.

[2] J. Thönes, "Microservices," *IEEE Software*, vol. 32, no. 1, pp. 116–116, 2015.

[3] S. Newman, *Building microservices.* " O'Reilly Media, Inc.", 2021.

[4] M. Fowler, "Microservice trade-offs," Web page: https://martinfowler.com/articles/microservice-trade-offs.html.

[5] K. Gos and W. Zabierowski, "The comparison of microservice and monolithic architecture," in *2020 IEEE XVIth International Conference on the Perspective Technologies and Methods in MEMS Design (MEMSTECH)*, April 2020, pp. 150–153.

[6] N. C. Mendonca, C. Box, C. Manolache, and L. Ryan, "The monolith strikes back: Why istio migrated from microservices to a monolithic architecture," *IEEE Software*, vol. 38, no. 05, pp. 17–22, sep 2021.

[7] Y. Abgaz, A. McCarren, P. Elger, D. Solan, N. Lapuz, M. Bivol, G. Jackson, M. Yilmaz, J. Buckley, and P. Clarke, "Decomposition of monolith applications into microservices architectures: A systematic review," *IEEE Transactions on Software Engineering*, vol. 49, no. 8, pp. 4213–4242, 2023.

[8] E. Evans, *Domain-driven design: tackling complexity in the heart of software.* Addison-Wesley Professional, 2004.

[9] P. M. P. Correia, "Development and evolution of e-assessment platform based on multiple choice questions," Ph.D. dissertation, Instituto Superior Tecnico, University of Lisbon, 2020.

[10] R. van Renesse, *Concurrent Programming in Harmony.* Cornell, 2020.

[11] D. Jackson, *Software Abstractions, Revised Edition.* The MIT Press, 2016.

[12] S. M. A. Shah, K. Anastasakis, and B. Bordbar, "From uml to alloy and back again," in *Proceedings of the 6th International Workshop on Model-Driven Engineering, Verification and Validation*, ser. MoDeVVa '09. New York, NY, USA: Association for Computing Machinery, 2009. [Online]. Available: https://doi.org/10.1145/1656485.1656489

[13] K. Anastasakis, B. Bordbar, G. Georg, and I. Ray, "Uml2alloy: A challenging model transformation," in *Model Driven Engineering Languages and Systems*, G. Engels, B. Opdyke, D. C. Schmidt, and F. Weil, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 436–450.

[14] Y. He, "Comparison of the modeling languages alloy and uml." *Software Engineering Research and Practice*, vol. 2, pp. 671–677, 2006.

[15] A. Panda, M. Sagiv, and S. Shenker, "Verification in the age of microservices," in *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, ser. HotOS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 30–36. [Online]. Available: https://doi.org/10.1145/3102980.3102986

[16] 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, Sep. 2023.

[17] P. Das, R. Laigner, and Y. Zhou, "Hawkeda: A tool for quantifying data integrity violations in event-driven microservices," in *Proceedings of the 15th ACM International Conference on Distributed and Event-Based Systems*, ser. DEBS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 176–179. [Online]. Available: https://doi.org/10.1145/3465480.3467838

[18] A. Ribeiro, "Invariant-driven automated testing," Ph.D. dissertation, Universidade NOVA de Lisboa, 02 2021.

[19] P. Pereira and A. R. Silva, "Transactional causal consistent microservices simulator," in *Distributed Applications and Interoperable Systems*, M. Patiño-Martínez and J. Paulo, Eds. Cham: Springer Nature Switzerland, 2023, pp. 57–73.

[20] F. Montesi, C. Guidi, and G. Zavattaro, "Service-oriented programming with jolie," *Web Services Foundations*, pp. 81–107, 2014.

[21] S. Giallorenzo, F. Montesi, M. Peressotti, F. Rademacher, and N. Unwerawattana, "JoT: A Jolie framework for testing microservices," in *International Conference on Coordination Languages and Models*. Springer, 2023, pp. 172–191.

[22] A. Adya, "Weak consistency: A generalized theory and optimistic implementations for distributed transactions," Ph.D., MIT, Cambridge, MA, USA, Mar. 1999, also as Technical Report MIT/LCS/TR-786.