TÉCNICO LISBOA

Detection of Invariant Violations in Microservices

João Quelhas Guterres Serrão Fitas

Thesis to obtain the Master of Science Degree in

Computer Science and Engineering

Supervisors: Professor António Manuel Ferreira Rito da Silva
Professor Luís Eduardo Teixeira Rodrigues

Examination Committee

Chairperson: Pedro Tiago Gonçalves Monteiro
Supervisor: Professor António Manuel Ferreira Rito da Silva
Member of the Committee: Prof. Carla Maria Gonçalves Ferreira

October 2024

Declaration
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.

Acknowledgments

I would like to thank my parents for their friendship, encouragement, and caring over all these years. They have always been there for me through thick and thin, and without them, this project would not be possible. I would also like to thank my grandparents, aunts, uncles, and cousins for their understanding and support throughout all these years.

I would also like to acknowledge my dissertation supervisors Professor António Rito Silva, Professor Luís Rodrigues and my colleague, Rafael Soares, for their insight, support, and sharing of knowledge, which have made this Thesis possible.

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

To each and every one of you – Thank you.

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

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, generating states that violate correctness invariants. This work studies techniques to: 1) detect automatically executions that may cause invariants to be violated, and 2) automatically present concrete executions that illustrate those violations. We have built a tool, named DAVIAC, that achieves these goals. The tool encodes the application code and the invariants as *Satisfiability Modulo Theories* formulas and then uses a *Satisfiability Modulo Theories* solver to explore the space of possible interleavings and input parameters. When the violation of an invariant is found, the tool captures the exact interleaving that causes the violation. We have evaluated the tool by applying it to different microservice applications.

Keywords

Microservices, Concurrency, Invariants, Anomaly Detection, Formal Verification

Resumo

Num monólito, as funcionalidades são executadas como transações, isoladas umas das outras. Numa arquitetura de microsserviços, as funcionalidades podem ser compostas por múltiplas transações, cada uma executada num microsserviço diferente. Quando as funcionalidades são executadas concorrentemente, essas transações individuais podem se intercalar, gerando estados que violam os invariantes de consistencia. Este trabalho estuda técnicas para: 1) detectar automaticamente execuções que podem causar violação de invariantes, e 2) apresentar automaticamente execuções concretas que ilustram essas violações. Construímos uma ferramenta, chamada DAVIAC, que atinge esses objetivos. A ferramenta codifica o código da aplicação e os invariantes como fórmulas *Teorias do Módulo de Satisfiabilidade* e então usa um solucionador *Teorias do Módulo de Satisfiabilidade* para explorar o espaço de possíveis intercalações e parâmetros de entrada. Quando a violação de um invariante é encontrada, a ferramenta captura a intercalação exata que causa a violação. Avaliamos a ferramenta aplicando-a a diferentes aplicativos de microsserviços.

Palavras Chave

Microsserviços, Concorencia, Invariantes, Deteção de Anomalias, Verificação Formal

Contents

List of Figures

List of Tables

Listings

Acronyms

DDD Domain-Driven Design

SMT Satisfiability Modulo Theories

ORM Object Relational Mapping

AST Abstract Syntax Tree

1

Introduction

Contents

Microservices are an architectural style that promotes the development of applications through the composition of small loosely coupled services, in contrast to the traditional monolithic architecture in which all functionalities are provided by a centralized 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 deployment and execution.

Unfortunately, managing the effect of concurrency becomes more challenging in a microservice architecture compared to a centralized environment. In a monolith, functionalities are executed as isolated ACID transactions [4]. In a microservice architecture, by design, functionalities are composed of multiple transactions, each possibly executed in a different microservice. When functionalities execute concur-

rently, the resulting interleavings may generate global states that violate the application's invariants: applications' correctness rules that must be enforced throughout execution.

1.1 Motivation

Our goal is to facilitate the development of correct microservice applications by simplifying the detection of correctness violations that may occur during the execution. Detecting these violations via testing is notably hard and impractical due to the sheer number of possible interleaving scenarios. So, we aim to develop a tool that can automatically detect interleavings that can cause violations of invariants *at design time* and that can represent the interleaving in a way that helps the programmers understand the root cause of the violation, making it easier to resolve.

1.2 Contribution

This work analyzes, implements, and evaluates techniques to discover and represent invariant violations that can occur in microservices applications. As such, this thesis' main contributions are the following:

- A formulation to represent microservice executions that can be used in formal verification contexts, in the form of SMT formulas.
- A tool that automatically generates the aforementioned formula from a microservice application and uses it to detect invariant violations.
- Automatic procedures to visualize the violations, as concrete executions.

1.3 Results

This work has produced the following results:

- A tool named DAVIAC capable of discovering invariant violations in microservice applications and visualizing them in various ways.
- An experimental evaluation of DAVIAC, that assesses its ability to analyze real applications and its performance and scalability.

1.4 Research History

This work was developed under the DACOMICO (Data Consistency in Microservices Composition) project, which has the objective of guiding programmers in the effort of decomposing a monolith into microservices. A tool such as DAVIAC, which is capable of detecting and simplifying the process of handling invariant violations, in other words, application correctness violations, is a relevant contribution to this project.

Early results from this thesis have been published as:

> *Deteção de Violação de Invariantes em Microserviços. J. Fitas, R. Soares, A. Silva and L. Rodrigues. Actas do décimo quinto Simpósio de Informática (Inforum), Lisboa, Portugal, Setembro 2023.*

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

1.5 Organization of the Document

The document is organized as follows: Chapter 2 provides the required background. Chapter 3 describes the related work. Our tool, DAVIAC, is presented in Chapter 4 and Chapter 5 describes its experimental evaluation. Chapter 6 presents the conclusion and gives directions for future work.

2

Background

Contents

2.1 Example: Simple Bank

In this document, we use a toy application, named **Simple Bank**, to illustrate the different concepts and mechanisms introduced in the thesis. The domain of this application includes *client*s that have an *id*, a *name*, and an *address*, and *account*s that are linked to clients by their *id* and have the *client*'s *name* and a *balance*, as shown in Figure 2.1. There is a functionality that allows *client*s to withdraw money, and another that allows the *name* to be changed. Lastly, there are two correctness invariants in this domain: the first is that no *account* can have a negative *balance* at any point, and the second is that the *name* of a *client* should match the *name* on their *account* once the update name process is completed.

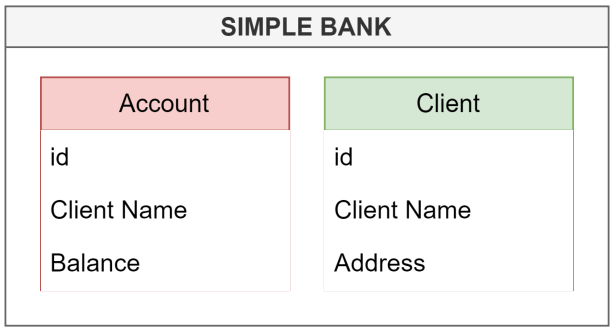


Figure 2.1: Simple Bank Domain

2.2 The Microservices Architecture

In a microservices 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 deployable and scalable independently, potentially having its own storage. Contrary to the traditional monolithic architecture where a centralized piece of software is responsible for handling all the functionalities of the application.

Microservice architectures have become popular and are often considered when developing new systems. Some of the most appealing aspects of breaking up an application's functionalities into microservices are [3, 5, 6]:

- *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 and 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 multiple loosely coupled services poses several challenges, such as [3, 5, 6]:

- *Operation Costs:* Without proper development infrastructure and monitoring tools, the increased number of components of this architecture may be unmanageable for some organizations.
- *Data Synchronization 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, functionalities may span multiple microservices, creating space for interleavings. With increased business logic complexity, functionalities tend to increase in size, making the correctness verification of invariants for all possible interleavings more difficult, leading to possible incorrect behavior being unnoticed during development.

Given the downsides of microservice compositions, it is important for the developer to weigh the pros and cons carefully, as there have been cases where microservice applications needed to be reimplemented as monoliths to solve these problems [7].

2.3 Domain-driven Design

The design of microservice applications is a difficult task when the application has complex business logic and tightly coupled functionalities [8]. One common way to do so is through a Domain-Driven Design (DDD) approach [9]. 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.

Aggregates capture relevant aspects of the domain by including data types, 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 [9]. 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 [9]. In order to avoid circular dependencies.

To clarify these concepts, let us expand on the previously 2.1 introduced Simple Bank application. Listing 2.1 presents the entities of the *Clients* and *Accounts* aggregates, where the *name* is an example of duplicate information kept in both aggregates.

Listing 2.1: Simple Bank Aggregates

```
1  Aggregate Clients {
2      Root Entity Client {
3          Long id;
4          String name;
5          Address address;
6      }
7
8      Entity Address {
9          Long id;
10         String street;
11         String city;
12     }
13 }
14
15 Aggregate Accounts {
16     Root Entity Account {
17         Long id;
18         Client holder;
19         Float balance;
20     }
21
22     Entity Client {
23         Long id;
24         String name;
```

```
25        }
26    }
```

Listing 2.2 expands the *Accounts* aggregate by introducing its invariants. There is one intra-invariant stating that the *balance* of the *account* entities must be greater than or equal to zero. And there is one inter-invariants that states that the *name* of the *client* entities in the *Accounts* microservice must correspond to the *name* of that *client* in the *Clients* microservice.

Listing 2.2: Simple Bank Accounts aggregate's Invariants

```
1  Aggregate Accounts {
2      IntraInvariants {
3          root.balance >= 0;
4      }
5
6      InterInvariants {
7          Clients.get(root.client.id).name == root.client.name;
8      }
9  }
```

Figures 2.2 and 2.3 present two functionalities of Simple Bank: *Update Name* and *Withdrawal*, which are responsible for updating the *client* entities' *name* and withdrawing money, respectively.

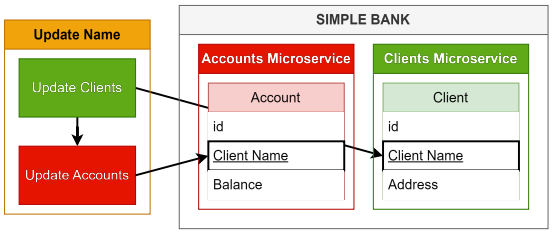


Figure 2.2: Update Name Functionality

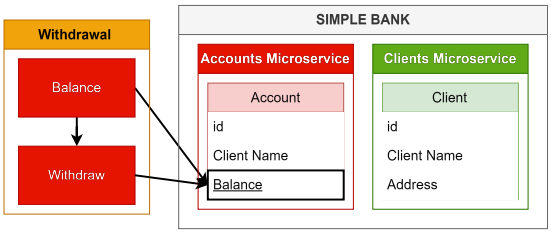


Figure 2.3: Withdrawal Functionality

2.3.1 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 isolated execution of the functionality. The second is the occurrence of anomalies during the concurrent execution of functionalities. Both of these cases are relevant to our work, and it is our goal to detect both occurrences.

Furthermore, not all invariant violations are the same. As mentioned in the previous Section 2.3, 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 be upheld at all times during execution. The second may be violated during intermediate states, but must be upheld after all the functionalities manipulating entities relevant to the invariant 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.

3

Related Work

Contents

There are two main distinct approaches to analyze the execution of an application, namely following a black-box approach or following a white-box approach. The first approach focuses only on the effects of an execution, having no knowledge of the application's inner workings, while the second approach also covers the internal operation of the application. 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 the white-box approach are presented in Section 3.1 and tools following the black-box approach in Section 3.2. All addressed systems are designed for or related to invariant analysis. Furthermore, Section 3.3 covers tools related to the development and testing of microservices. After this, Section 3.1 compares the tools and highlights their pertinence for this work. Lastly, Section 3.5 makes an in dept analysis of the tools, closer to achieving the goals of this work.

3.1 White-Box Analysis Tools

3.1.1 Harmony

Harmony [10] is a white-box tool designed to detect errors in multi-threaded 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, a Python-like language designed to simplify the analysis of concurrent programming applications. As such, the application under test must be translated manually into HarmonyLang to use this tool. 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 shortest path from the application's 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 many programming languages; for instance, it does not support classes. Lastly, since this tool uses a model checker, the time required to run complex scenarios is significant, or it may not finish within a useful period.

3.1.2 Alloy

Alloy [11] is a modeling language and solver 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 constraints, Alloy supports the validation of invariants and the analysis of methods; that is, it can verify if the execution of a method or methods can lead to invariant breaches.

The versatility of this language and solver 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. Furthermore, this tool also, provides graphical representation of sample instances which violate the domain constraints.

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 whether the models verify the invariants, and

second, it uses them to determine validity criteria for the messages exchanged between the microservices during runtime.

Ucheck operates in three stages. The first two statically validate 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.

3.1.4 MAD

MAD [16] is a white-box tool designed to assist developers in assessing the costs associated with the migration of a monolith to a given microservice decomposition, indicating the number and types of data anomalies that will arise from applying a decomposition.

More specifically, MAD takes as input a representation of a monolith application and a desired decomposition of its functionalities into microservices, which are then translated to Z3[1] statements that, combined with its anomaly finding engine, lead to the discovery of execution sequences that generate transactional anomalies [17]. The use of an SMT 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 a useful period.

3.1.5 Noctua

Noctua [18] is a white-box tool designed to verify the consistency of distributed web applications by indicating execution instances where consistency is not preserved.

To do so, Noctua receives as input the source code of a web server (supporting only Python Django web servers), and automatically converts it into SMT formulations representing each possible functionality execution of the web server. By analyzing said formulas, Noctua can discover which execution or which combination of executions causes inconsistent states in the web server. The aforementioned inconsistent states originate from data anomalies which are not handled by the web server.

3.2 Black-Box Analysis Tools

3.2.1 HawkEDA

HawkEDA [19] 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

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

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. The tool runs the application in a controlled environment, providing performance metrics such as average latency and throughput, along with any invariant breaches that might have encountered. Requests are randomized according to the configuration parameters, so there are no guarantees of the completeness of the analysis.

3.2.2 PETIT

PETIT [20] is a tool designed to test microservice applications provided as a black-box, for example, to test a third-party API, of which the implementation is unknown. The tool is introduced along with a specification language for annotating APIs called APOSTL, which should be used to annotate the API under test with the invariants that define the correct operation of that API. In essence, this tool takes the definition of an API annotated with rules that define the correct behaviors of each API method and performs tests with randomly generated inputs or according to some user-provided pattern. This means that while the analysis can reveal some invariant violations, there are no guarantees that it discovers all of them.

3.3 Testing and Development Tools

3.3.1 Transactional Causal Consistent Simulator

The Transactional Causal Consistent Simulator [21] 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 boilerplate 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 perform 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.

3.3.2 Jolie & JoT

Jolie [22] 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 are 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 [23], 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, 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 Comparison and Discussion

To sum up the most relevant characteristics of the tools presented in the previous section, their characteristics are presented in Table 3.1, along with the new tool produced by this work, DAVIAC.

Tool	Black/White-Box	Invariants	Microservice Oriented	Complete Analysis	Source Code Analysis
HawkEDA	Black-Box	API Level	✓	✗	✓
PETIT	Black-Box	API Level	✓	✗	✓
Simulator	White-Box	Entity Level	✓	✗	✓
JoT	White-Box	Entity Level	✓	✗	✓
Ucheck	White-Box	Entity Level	✓	✓	✗
Harmony	White-Box	Entity level	✗	✓	✗
Alloy	White-Box	Entity level	✗	✓	✗
MAD	White-Box	Not Supported	✓	✓	✓
Noctua	White-Box	Not Supported	✗	✓	✓
DAVIAC	White-Box	Entity level	✓	✓	✓

Table 3.1: 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. In contrast, invariants defined at the entity level can be validated by analyzing the entities directly, making the entity-level representation more efficient.

Harmony [10], Ucheck [15], and Alloy [11] define invariants at the entity level, while HawkEDA [19]

and PETIT [20] 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 of application internals, such as entities, and are therefore unable to define invariants at the entity level. This gives them a disadvantage 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. Albeit they are still useful for testing services whose code-base cannot be accessed.

The Simulator [21] and JoT [23] 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. However, they are relevant for this work in terms of testing environments and techniques, as is shown later in this Chapter.

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 correct behaviors are during runtime; and at runtime to prevent the occurrence of said incorrect behaviors. That is, 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, Alloy, and Noctua. Within these, it is worth highlighting that MAD and Noctua are the only ones that support source code analysis and do not require a manually generated model of the application. The capabilities of these four tools and their potential application as a solution to the problem of this work are discussed in Section 3.5.

Another relevant aspect of this work is addressing the discovered violations. From a high-level perspective, this is done in three stages: understanding the violation, reworking the problematic functionalities, and testing the new code. In terms of addressing the first step, Alloy and Harmony are the only tools with specific components for the understating of the results, by providing a graphical representations of the results. The second stage could be addressed with suggestions of possible ways to alter the code to avoid violations. However, that is outside the scope of this work and will not be covered. Lastly, when testing the code, different testing techniques may be required depending on the technologies used in the application under test. One possible solution is to support multiple technologies and generate technology-specific test cases, such as those in the Simulator [21]. Another would be to use a generic framework like Jot [23] 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. In

contrast, Jot 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. Given the various approaches to this issue in the industry, it was decided not to include concrete test case generation in this work and instead to focus on providing the information required to create test cases in a useful way.

3.5 Discovering Invariant Violations Using Previous Work

To clarify what Harmony [10], MAD [16], Noctua [18] and Alloy [11] (the tools highlighted in the previous Section 3.4) are already capable of achieving and what their limitations are this section discusses their use to discover violations.

3.5.1 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 translating the whole domain, Harmony requires that the possible input for each function be bounded, and methods for random input values are provided and could be used. Furthermore, indications of which functions may be executed in parallel and how many instances of each have to be provided. Defining this value is not a trivial matter, as this problem is again encountered in DAVIAC, and discussed in Section 4.4.1. 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 impossible to make the distinction between eventual invariants and absolute invariants, as described in Section 2.3.1. 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, in other words to a very significant number of false positives. Harmony will report every intermediate state where an eventual invariant is breached, including scenarios when, in the final state, the invariant is upheld, cases which do not constitute a violation.

3.5.2 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 in the discovery of breaches that are caused by the interleaving of correct functionalities. However, if the invariant breach occurs due to a bug in the functionality, using MAD is not sufficient, as it occurs outside a data anomaly. 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 must be analyzed by hand. However, as previously mentioned, for a complete analysis, the functionalities' sequential executions have be tested as well to cover the missing cases, meaning that a lot of the work still has to be done outside MAD. Eventual invariants should be tested at the end of each scenario and 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. 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.

3.5.3 Discovering Invariant Violations Using Noctua

Much like MAD [16], Noctua [18] is not designed for invariant analysis. Its goal is the discovery of inconsistencies in distributed web applications. To do so, Noctua probes the executions of the web server for data anomalies, meaning that, like MAD, the results would then have to be verified for invariant violations. Essentially, the use of this tool for the goals of this work is very similar to the use of MAD, having the same pros and cons. Still, Noctua's methods for translating Python source code to SMT syntaxes were relevant for this work, especially the ability to translate from Object Relational Mapping (ORM) syntax.

3.5.4 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 an 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 a feature; however, they may still be relevant for invariant violations.

In summary, 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.

3.5.5 Takeaways

Harmony can 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 were available, the tool would still not meet all the invariant violation detection goals of this work.

MAD and Noctua are not capable of detecting invariant violations. Detecting anomalies in the data that may lead to invariant violations in some cases. But it does not cover all the cases and requires a lot of manual work. However, their components for automatically translating and modulating source code were very insightful 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, does not 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, Harmony [10], MAD [16], Noctua [18], and Alloy [11] all lack some aspect to achieve the invariant violation detection goals of this work. In summary, the requirements for the new tool and what these tools have already accomplished are as follows.

- *Verify Microservices Invariants*: Alloy and Harmony do this, while MAD and Noctua do not.

- *Make A Complete Analysis*: All these tools do it.
- *Analyze Source Code*: MAD and Noctua perform source code analysis
- *Distinguishing Between Eventual and Absolute Invariants*: None of the tools provide this feature, nor a way to emulate it.

For these reasons, this work produced a new tool, DAVIAC, that achieved all of its goals, using some of MAD and Noctuas's concepts on automatic source code analysis. 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 uses formal verification techniques to grant a complete analysis of the application. Given that both MAD and Noctua use an internal representation optimized for SMT solvers, this work will also follow that approach. This tool is presented in the next chapter.

4

DAVIAC

Contents

This chapter presents a tool produced by this work for theautomatic detection of invariant violations in business logic rich applications, using static analysis and without the use of code instrumentalization, DAVIAC (from the Portuguese Deteção Automática de Violação de Invariantes em Aplicações de Domínio Complexo). To detect possible invariant breaches in an application, the tool encodes the application's invariants and transactions in Satisfiability Modulo Theories (SMT) [24]. Using an SMT solver, the tool explores the universe of possible functionality interleavings and arguments in search of combinations that cause invariant violations. Upon discovery, the initial state, functionality interleaving, and arguments are recorded, and all violations are presented at the end of the tool's execution.

4.1 Usage Overview

To use DAVIAC the user first provides a description of the application under test, specifically the description of the application's entities, invariants, transactions, and functionalities. Details of these descriptions are presented in Section 4.2.1. Then, to run the tool, the user must define the number of concurrent functionalities to consider (f_{max}); naturally, the user is free to choose any number. However, an incremental approach is recommended. In other words, it is recommended that the user begins with a f_{max} value of one and resolves any discovered violation test. Once resolved, the user should run once again with the same f_{max} value, ensuring no other violation remains and, if solved, moves to f_{max} of two and so on until no violations are discovered. This approach is not without limitations, as is discussed in Section 4.4.1.

4.2 Architecture

The architecture and execution flow of the DAVIAC tool are represented in Figure 4.1. The tool follows this execution: first, the source code, the entities, and the information about the functionalities and transactions are supplied to the tool, processed, and transformed into an internal representation, independent of the specific technologies used in the provided specification.

This internal representation is compiled into an SMT formula and analyzed by an SMT solver to detect invariant violations. Lastly, the discovered violations are presented to the user using several visualization strategies. The following sections describe each DAVIAC module in detail.

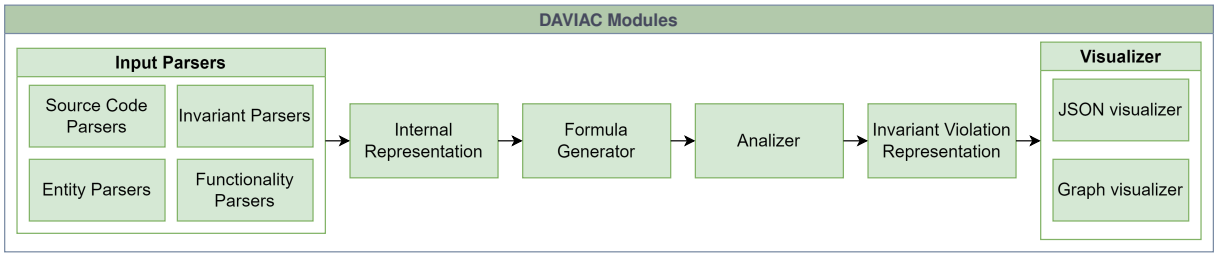


Figure 4.1: DAVIAC Module Sequence

4.2.1 Input Parser

DAVIAC uses four parser classes: I) Entity Parser, II) Invariant Parser, III) Transaction Code Parser, and IV) Functionality Parser.

Listing 4.1: Sample entity description file

```
1 CREATE TABLE accounts (
2     id          INTEGER,
3     balance     INTEGER,
4     PRIMARY KEY (id)
5 );
```

4.2.1.A Entity Parser

The entity parser is responsible for translating the systems entities (including their attributes and respective types) into the internal representation used by the tool. The details of the internal representation are presented in Section 4.2.2. Entities are usually represented in SQL or ORM) [25]. The tool can extract attributes and their types using the database schemes used in both techniques. Currently, the DAVIAC prototype only supports entities encoded as SQL, considering each table as an entity and its comprising columns the entities' attributes. Information on database schemas can be obtained from SQL database creation commands, whether manually written by the programmer or automatically generated by the ORM. The listing 4.1 presents an example of the entity representation currently supported by the tool. It consists of an SQL file with the commands for the creation of a table containing data referring to the entity *accounts*, made up of two attributes, *id* and *balance*, both of type integer. Currently, only basic types are supported by the tool.

4.2.1.B Invariant Parser

Invariants capture the domain's consistency rules, which, according to DDD, should be formalized for each aggregate during its design. This formalization is supplied to the tool, including information related to the type of each invariant (i.e. eventual or absolute).

For clarification, an eventual invariant is breached if it is not upheld in an application's quiescent state. This means that it can be breached during the execution of a functionality as long as it is restored before the execution finishes. Conversely, absolute invariants must be upheld at all times during the application's execution.

The Invariant Parser is responsible for translating these high-level specifications to an internal representation, connecting each invariant with the entities and functionalities involved.

Concretely, the current parser in the prototype takes a JSON file that contains the invariant description in an SQL-like syntax, as shown in the Listing 4.2. Extending the previously presented domain, the file contains two invariants: an absolute invariant, indicating that at no point the *balance* attribute of any *accounts* entity be less than 0, and an eventual invariant, indicating that any *accounts* entity for which the *id* attribute is the same as that of a *clients* entity must have the same value for the *name* attribute on

Listing 4.2: Sample Invariant description file

```
{
    "Absolute Invariants": [ "accounts.balance >= 0" ],
    "Eventual Invariants": [ "WHERE accounts.id == clients.id THEN
        accounts.name == clients.name" ]
}
```

Listing 4.3: Invariant Grammar

```
Invariant := Where Conditions Then Conditions | conditions

Conditions := Condition AND Condition | Condition OR Condition | Condition

Condition := Entity.Attribute Operator Entity.Attribute
| Entity.Attribute Operator Literal

Operator := < | > | <= | >= | == | !=
```

both entities.

Currently, the prototype supports invariants that affect all entities of a type or subset of entities indicated by a given condition, presented as a clause WHERE. The prototype supports logic changing of conditions using conjunctions and disjunctions as well as the following operators: “<”, “>”, “<=”, “>=”, “==” e “! =”. Concretely the Invariant Grammar can be defined as shown in Listing 4.3.

4.2.1.C Transaction Code Parser

The Transaction Code Parser is responsible for translating the entity accesses performed by the application transactions into the internal representation. These accesses are represented as an execution graph containing the operations performed over the accessed entities and the conditions required to perform the said operations. This graph is translated to an internal representation that is agnostic to the programming language used by the application, allowing the implementation of multiple parses, each supporting a different input technology.

The current prototype includes a parser for Java code, which uses SQL statements to manipulate the entities. This parser uses the Java Parser[1] Library to build and traverse an Abstract Syntax Tree (AST), which represents the transactions that make up the functionalities of the application under test. As it traverses the tree, the parser generates the corresponding nodes in the internal representation, which will store the types of the entities in the code, the arguments of each transaction, and all expressions that are used in conditions or writes to entities. In the future, the tool can be expanded with parsers

[1] https://javaparser.org/

Listing 4.4: Sample source code file (trimmed)

```
public void balancce(int id_account) throws SQLException {
    PreparedStatement stmt1 = connection.prepareStatement("SELECT balance" +
                                                          "FROM accounts WHERE id = ?");
    stmt1.setInt(1, id_account);
    ResultSet rs = stmt1.executeQuery();
    rs.next();
    int balance = rs.getInt("balance");
}
```

Listing 4.5: Functionality description file sample

```
{
    "Withdrawal": [
        "balance",
        "withdraw"
    ],
    "Update Name": [
        "update_clients",
        "update_accounts"
    ]
}
```

that support code using popular frameworks, like Spring[2], Django[3], among others that are common throughout the microservice world.

The Listing 4.4 presents the “balance” method of a simple banking application, written as Java code in the style currently supported by DAVIAC.

4.2.1.D Functionality Parser

The Functionality parser is responsible for translating the sequences of transactions into functionalities. A functionality corresponds to a sequence of transactions executed on one or more microservices and by an order defined by the functionality. Information about these sequences allows the tool to narrow its search universe, avoiding the exploration of interleavings of transactions that are impossible in the application.The current prototype of DAVIAC only supports linear sequences of transactions, which were enough to completely analyze our test case.

The Listing 4.5 presents an example of the information currently used by the prototype of the tool. It receives a JSON file containing the functionalities and their transactions, presented as a list sorted by sequence order. The sample only contains one functionality, *Withdrawal*, which is composed of two transactions, *balance* and *withdraw*, executed in this order.

[2] https://spring.io/
[3] https://www.djangoproject.com/

4.2.2 Internal Representation

The internal representation allows the decoupling of the generation of the SMT formula from the codification of the system under test, which makes the tool's expansion easier. It stores the information extracted by the input parsers required for the SMT formulation created by DAVIAC. Concretely, the internal representation is composed of three lists of objects: entities, invariants, and functionalities. The entities hold information related to their attributes, specifically related to their type. The invariants define the restrictions imposed on the possible coherent entity attribute states and are categorized as absolute or eventual. The functionalities keep a sequence of transactions, where each transaction is represented by an AST and captures the entities and attributes manipulated by itself. The AST representation is generic in the sense that it is agnostic to the input programming language. Yet, it is capable of capturing all its relevant aspects, namely the chaining of entity accesses and all branching and conditions relevant to these accesses.

The internal representation also connects the invariants, the relevant attributes to maintaining their correctness, and the transactions that interact with these attributes. These connections are essential to the generation of the SMT formulation, allowing the search universe to be reduced to include only the functionalities relevant to each invariant.

Figure 4.2 presents the internal representation of the sample banking application, highlighting the connections pertaining to the absolute invariant "*+Accounts.balance ¿= 0+*". In this example, the absolute invariant restricts the range of possible values for the attribute *balance* of the entity *Accounts*, being this attribute manipulated by the transactions *balance* and *withdraw* of the functionality *Withdrawal*. In this way, the tool is capable of identifying the relevant functionalities for a given invariant, restricting the search of executions that breach the invariant only to executions that involve these transactions.

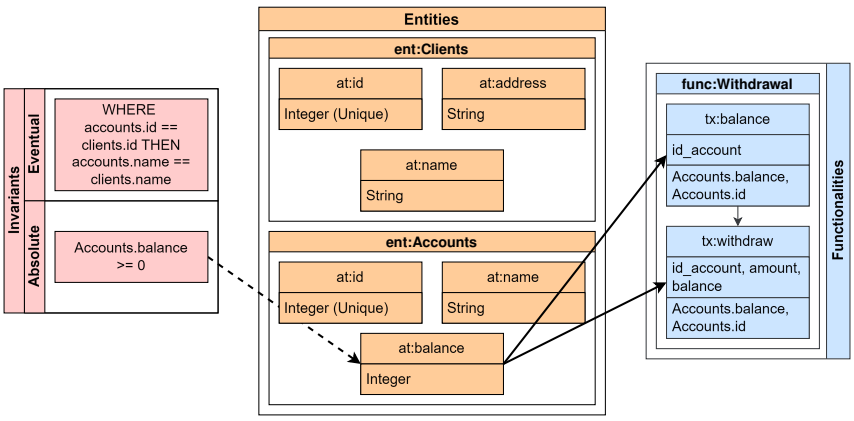


Figure 4.2: Simple Bank's internal representation on DAVIAC (Update Name Functionality is omitted)

4.2.3 Formula Generator

This component is responsible for compiling the internal representation to SMT by generating the formulas that the solver will verify to discover invariant violations. The tool explores, for each invariant, all the possible functionality executions that interact over the relevant entities to the invariant. For that purpose, the tool covers all the possible functionality interleavings, not only among different functionalities but also among concurrent instances of the same functionality. In addition, for each interleaving, DAVIAC covers all possible inputs for each transaction and possible initial application states. DAVIAC only explores initial states which verify all of the application's invariants. Using the information contained in the AST, the tool can explore all branchings of accesses to entities that occur from conditional accesses within a transaction.

Taking advantage of the fact that the search of violations for each invariant is independent, DAVIAC generates separate formulas for each invariant. This separation allows not only the parallelization of the violation search, making concurrent invocations of the solver possible, but it also reduces the complexity of the SMT formulations, improving the search performance. The possibility of parallelizing the analysis is discussed further in Section 6.2.

To limit the maximum search depth of the tool, the maximum number of concurrent functionality instances in a given execution is a predetermined value f_{max}, defined in the tool configuration. For a more complete analysis, multiple incremental values of f_{max} should be tested. More details on the impact of the choice of this value are discussed in Section 4.4.1.

Lastly, DAVIAC's current prototype assumes that all transactions are serializable, although this will be expanded in the future, as discussed in Section 6.2.2.

Design details on the SMT formulas are covered in Section 4.3.

4.2.4 Analyzer

This component verifies the satisfiability of the formulas produced by the generator. DAVIAC uses Z3[4] as a solver due to its efficiency and popularity in the area. The analyzer is invoked for each generated formula and, if a formula is satisfiable, the solution model is used to represent the violation. Otherwise, it is guaranteed that the execution represented by that formula cannot lead to an invariant violation and, as such, can be discarded. Concretely, approaching the problem as an SMT problem provides the guarantee of total coverage of violations in the applications caused by the simultaneous execution of up to f_{max} functionalities because DAVIAC analyzes all the executions for all the possible functionality combinations of length up to f_{max} as SMT formulas which are mathematically proven satisfiable or not. In case no violations are found for a certain f_{max}, it is guaranteed that no violations can occur when up

[4] https://github.com/Z3Prover/z3

to f_{max} concurrent functionalities are executed in the application.

4.2.5 Invariant Violation Representation

This representation is generated by extracting information from the solution model created by the analyzer, and contains all the relevant data to identify and reproduce the discovered violation. Namely, it contains which invariant was breached, the involved functionalities, the execution order, the application's initial state, and the inputs that caused the violation. The purpose of this representation is to separate the SMT solver's output from the result presented to the programmer. This allows the violation to be reported in several ways and makes it possible for the programmer to create multiple visualizers for the violation according to their use cases.

4.2.6 Visualizers

These components are responsible for meaningfully presenting the invariant violations to the programmer. DAVIAC is prepared to use several visualizers and encourages the user to augment it by implementing visualizers that best suit their needs, such as a visualizer that automatically creates test cases for their specific environment. The prototype is equipped with two visualizers, one that generates a JSON description of the violation 4.2.6.A, and one that generates a graph of the violation 4.2.6.B.

4.2.6.A JSON Visualizer

This visualizer creates a JSON file containing a data flow description of the execution where the invariant violation occurs. This description comprises a list of the states of the application's entities, with the functionalities that originate each state between them, along with their respective arguments. A concrete example is Listing 4.6, where the interleaving of two instances of the *Withdrawal* functionality makes it so that, in the last state, the account with *id* 0 has a negative balance, breaching the invariant.

Listing 4.6: JSON visualization of a Simple Bank Invariant violation

```
1 [
2     "Accounts.balance >= 0",
3     {
4         "Accounts": [
5             {
6                 "balance": 15,
7                 "id": 0
8             }
```

```
        ]
    },
    {
        "Functionality": "Withdrawal",
        "Functionality instance": 1,
        "Transaction number": 1,
        "Transaction name": "balance",
        "Arguments": {
            "id_account": 0
        }
    },
    {
        "Accounts": [
            {
                "balance": 15,
                "id": 0
            }
        ]
    },
    {
        "Functionality": "Withdrawal",
        "Functionality instance": 2,
        "Transaction number": 1,
        "Transaction name": "balance",
        "Arguments": {
            "id_account": 0
        }
    },
    {
        "Accounts": [
            {
                "balance": 15,
                "id": 0
            }
        ]
    },
    {
        "Functionality": "Withdrawal",
```

```
        "Functionality instance": 1,
        "Transaction number": 2,
        "Transaction name": "withdraw",
        "Arguments": {
            "id_account": 0,
            "balance": "15",
            "amount": "10"
        }
    },
    {
        "Accounts": [
            {
                "balance": 5,
                "id": 0
            }
        ]
    },
    {
        "Functionality": "Withdrawal",
        "Functionality instance": 2,
        "Transaction number": 2,
        "Transaction name": "withdraw",
        "Arguments": {
            "id_account": 0,
            "balance": "15",
            "amount": "15"
        }
    },
    {
        "Accounts": [
            {
                "balance": -10,
                "id": 0
            }
        ]
    }
]
```

4.2.6.B Graph Visualizer

This visualizer creates a graph that contains a description of the data flow of the execution where the invariant violation occurs. This description comprises a sequence of entities states that compose the application, with the functionalities that originate each state linked to them, along with their respective arguments. The red arrows indicate the sequence of transactions, whereas the links between the initial and final states of the transactions are annotated in black. Furthermore, any changes from one state to the next are highlighted in yellow or red, and the latter also indicates that the changes in that state caused a violation.

A concrete example is in Figure 4.3, where, as in the previous section, the interleaving of two instances of the *Withdrawal* functionality makes it so that, in the last state, the account with *id* 0 has a negative balance, breaching the invariant.

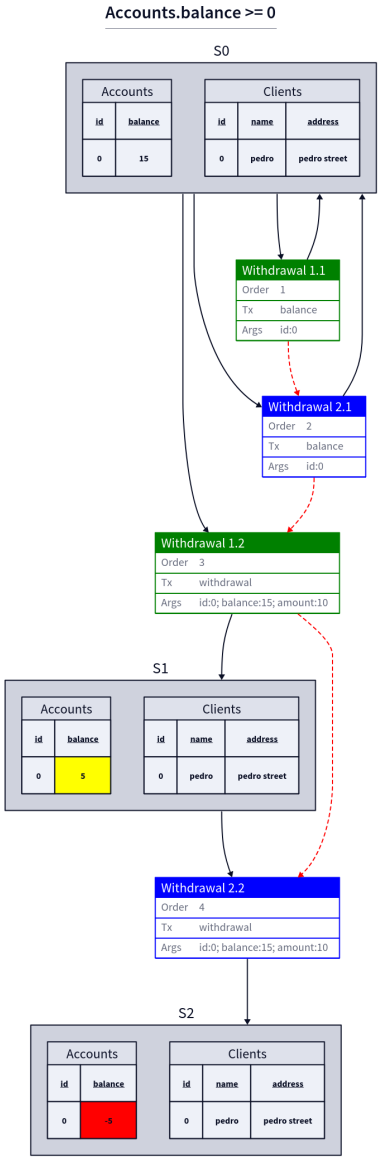


Figure 4.3: Graph visualization of a Simple Bank Invariant violation

4.3 SMT Formula Specifics

The goal of the SMT formulas is to represent the execution of an interleaving of functionalities in order to discover Invariant breaches. The execution of the application is represented as a series of chained states, and moves from one state to the next due to the execution of transactions which manipulate entities, upon which the Invariants are defined.

DAVIAC generates separate SMT formulas for all possible functionality interleavings for each invariant. However, much of the information contained in this formulas is the same, and is recycled from one formula generation to the other in order to save resources.

The application starts from an initial state respecting all invariants, while the final state is the system state after the execution of all transactions. Lastly, given that DAVIAC assumes transactions to be serializable, each transaction has a start state and an end state.

4.3.1 Formula Components

Concretely, each formula contains: the encoding of the domain *Entities* into SMT sorts; the encoding of the application's *Transactions* into SMT clauses; the *Initial State Constraints*, i.e. all the invariants to be upheld on the initial state; the *Execution Order* under test; the *Invariant* which is under test. From this list, only the last two need to be generated again for each formula. The rest are generated once and reused in all formulas.

In this section we now breakdown each component of the formula. The Execution Order, Initial State Constraints and Invariants are all contained within the body of the *Exist* clause. The *Exist* clause header, defines how many states and functionality parts should exist, while the Entities and Transactions are introduced as a set of separate clauses. These separate clauses remain the same for the formula of each execution while the Exists clause is adapted depending on the transaction/functionality types and their execution order.

4.3.1.A Entities

Entities are modeled in the formula as SMT sorts. These sorts are then used in functions, as arguments, to obtain the values of each entities' attributes. Each function takes as an argument the state and entity which the attribute is being accessed on and returns the value in the correct type. Furthermore, to represent whether an entity exists in a state or not (in case it is created or deleted in the middle of an execution), an additional function is created to represent its existence in a given state, receiving the entity and state in question, and returning a boolean. Lastly the formula also needs to capture the behavior of both unique and foreign attributes. Unique attributes are represented by clauses indicating that in any state, if two entities of the same type have the same unique attribute, then they are the same entity. Foreign attributes are represented by clauses stating that if in a state an entity holds a foreign attribute, then the entity reponsible for the foreign attribute must also exist, similar to SQL foreign keys. For example, in the bank application, the *Accounts* Entity is represented by the Accounts sort, the exists function and the one function for each of its attributes, as shown in Listing 4.7. Listing 4.8 contains the unique clause for the *id* attribute of the *Accounts* entity. Lastly, Listing 4.9 introduces a foreign clause

for the *name* attribute of the *Accounts* entity. For the purpose of this example, assume that this attribute is a foreign attribute linked to the *name* attribute of the *Clients* entity.

Listing 4.7: SMT formula representation of Simple Bank's Accounts Entity

```
(declare-sort Accounts)

(declare-fun Accounts_exists (State) Bool)

(declare-fun Accounts_id (Accounts State) Int)
(declare-fun Accounts_name (Accounts State) String)
(declare-fun Accounts_balance (Accounts State) Int)
```

Listing 4.8: SMT formula representation of a unique atribute

```
(assert (forall ((accounts_1 Accounts) (accounts_2 Accounts) (state_1 State)
    (state_2 State))
  (=> (= (Accounts_id accounts_1 state_1) (Accounts_id accounts_2 state_2))
      (= accounts_1 accounts_2)
   )
))
```

Listing 4.9: SMT formula representation of a foreign atribute

```
(assert (forall ((accounts Accounts) (state State))
  (=>
    (Accounts_exists accounts state)
    (exists ((clients Clients))
      (and
        (=  (Accounts_id accounts state) (Clients_id clients state))
        (Clients_exists clients state)
      )
    )
  )
))
```

4.3.1.B Transactions

Unlike entities, transactions are modeled as the sort *Functionality_Part* and are identified by an id. Each transaction in the application is assigned a distinct integer id by a function named after the transaction, that takes no arguments and returns an *Int*. This id can be retrieved by the use of the *get_func_part_type* function which takes as an argument an instance of a *Functionality_Part* and returns its *Int* id. The id is used for matching the *Functionality_Part*s to their transaction types.

Each *Functionality_Part* (transaction) is part of a functionality, represented by the *Functionality* sort, which can be obtained by using the *get_func* function, which takes as an argument an instance of a *Functionality_Part* and returns an instance of *Functionality*, corresponding to its parent *Functionality*.

The arguments of each transaction, as well as any value that is propagated from one transaction to the next is represented by a function named as <*transaction name*>_<*argument name*> which receives as an argument the *Functionality_Part* instance and returns the argument value.

Lastly, the effect of executing the transaction is represented as an assert. These asserts enforce conditions to all *Functionality_Part* instances of the same type, translating the operations performed by the transaction on the Entities values in terms of the *Functionality_Part*'s start and end states.

The start and end states can be obtained using the *start_state* and *start_state* functions, which take as input the *Functionality_Part* instance and return the corresponding *State* instance.

The representation of the *balance* transaction (introduced in Listing 4.4) in the formula can be seen in Listing 4.10

Listing 4.10: SMT formula representation of Simple Bank's balance transaction

```
1  (declare-fun balance () Int)
2  (declare-fun balance_id_account (Functionality_Part) Int)
3  (declare-fun balance_balance (Functionality_Part) Int)
4
5  (assert (forall ((func_part Functionality_Part))
6    (=>
7      (= (get_func_part_type func_part) balance)
8      (exists ((start_state State) (end_state State))
9        (ite
10         (and
11           (= (start_state func_part) start_state)
12           (= (end_state func_part) end_state)
13         )
14         (forall ((clients Clients) (accounts Accounts))
```

```
15              (and
16                (=>
17                  (and
18                    (Accounts_exists accounts start_state)
19                    (=  (Accounts_id accounts start_state)
20                        (balance_id_account func_part))
21                  )
22                  (=  (Accounts_balance accounts start_state)
23                      (balance_balance func_part))
24                )
25
26                (=  (Accounts_exists accounts start_state)
27                    (Accounts_exists accounts end_state))
28                (=  (Accounts_id accounts start_state)
29                    (Accounts_id accounts end_state))
30                (=  (Accounts_balance accounts start_state)
31                    (Accounts_balance accounts end_state))
32                (=  (Accounts_name accounts start_state)
33                    (Accounts_name accounts end_state))
34
35                (=  (Clients_exists clients start_state)
36                    (Clients_exists clients end_state))
37                (=  (Clients_id clients start_state)
38                    (Clients_id clients end_state))
39                (=  (Clients_address clients start_state)
40                    (Clients_address clients end_state))
41                (=  (Clients_name clients start_state)
42                    (Clients_name clients end_state))
43              )
44            )
45            false
46          )
47      )
48    )
49  ))
```

4.3.1.C Execution Order

The execution order is what represents each execution in the SMT formula, as it introduces the chaining of transactions, what type of transaction they are and which functionality they belong to.

First, it indicates the execution flow, setting the start and end states of each *Functionality_Part* using the *start_state* and *end_state* functions declared in the *Exists* clause. Then, it sets the type of each *Functionality_Part* using the *get_func_part_type* function, according to the given interleaving the formula is validating.

Finally, instances of *Functionality_Part* that belong to the same functionality are also matched using the *get_func* function which receives an instance of *Functionality_Part* and returns the corresponding instance of *Functionality*.

To exemplify this, the Listing 4.11 contains part of the *Exists* clause, corresponding to an execution of two instances of the *Withdrawal* functionality, where the execution order is: *Withdrawal* 1.1 (tx:*balance*), *Withdrawal* 2.1 (tx:*balance*), *Withdrawal* 1.2 (tx:*withdrawal*), *Withdrawal* 2.2 (tx:*withdrawal*); like the example on Figure 4.3.

Listing 4.11: SMT formula representation of Simple Bank's Execution Order Example

```
;; Exists Clause
(assert (exists ((state_0 State) (state_1 State) (state_2 State)
    (state_3 State) (state_4 State) (func_part_0 Functionality_Part)
    (func_part_1 Functionality_Part) (func_part_2 Functionality_Part)
    (func_part_3 Functionality_Part))
  (and
    [...]

    ;; Execution Order
    (= (start_state func_part_0) state_0)
    (= (end_state func_part_0) state_1)
    (= (start_state func_part_1) state_1)
    (= (end_state func_part_1) state_2)
    (= (start_state func_part_2) state_2)
    (= (end_state func_part_2) state_3)
    (= (start_state func_part_3) state_3)
    (= (end_state func_part_3) state_4)

    (= (get_func_part_type func_part_0) balance)
    (= (get_func_part_type func_part_1) balance)
```

```
21      (= (get_func_part_type func_part_2) withdrawal)
22      (= (get_func_part_type func_part_3) withdrawal)
23
24      (= (get_func func_part_0) (get_func func_part_2))
25      (= (get_func func_part_1) (get_func func_part_3))
26
27      [...]
28    )
29  ))
```

4.3.1.D Initial State Constraints

Executions must start from a correct initial state (i.e. one which upholds all Invariants), otherwise one could detect invariant breaches raised due to already invalid initial states. As such, the *Exists* clause contains a clause for each invariant, indicating that it must be upheld in the initial state. The Simple Bank's invariants would be represented as shown in Listing 4.12.

Listing 4.12: SMT formula representation of Simple Bank's Initial State Invariants

```
1  ;; Exists Clause
2  (assert (exists ((state_0 State) (state_1 State) (state_2 State)
3      (state_3 State) (state_4 State) (func_part_0 Functionality_Part)
4      (func_part_1 Functionality_Part) (func_part_2 Functionality_Part)
5      (func_part_3 Functionality_Part))
6    (and
7      [...]
8
9      ;; Initial State Invariants
10     (forall ((accounts Accounts) (clients Clients))
11       (=>
12         (=  (Accounts_id accounts state_0)
13             (Clients_id clients state_0))
14         (=  (Accounts_name accounts state_0)
15             (Clients_name clients state_0))
16       )
17     )
18     (forall ((accounts Accounts))
19       (>= (Accounts_balance accounts state_0) 0)
```

```
20      )
21
22      [...]
23    )
24  )
```

4.3.1.E Invariants

The last component of the SMT formula is the representation of the invariant under test. This is represented by another exist clause inside the main *Exists* clause. The internal clause makes use of all the clauses defined so far to represent if there is a state X where the invariant under test is not verified. To distinguish between eventual and absolute invariants, the state X must either be the final state or any state (other than the initial), respectively. To illustrate this, the representation of both Simple Bank's invariants is presented, the eventual invariant in Listing 4.13 and the absolute in Listing 4.14.

Listing 4.13: SMT formula representation of Simple Bank's Eventual Invariant

```
1  ;; Exists Clause
2  (assert (exists ((state_0 State) (state_1 State) (state_2 State)
3      (state_3 State) (state_4 State) (func_part_0 Functionality_Part)
4      (func_part_1 Functionality_Part) (func_part_2 Functionality_Part)
5      (func_part_3 Functionality_Part))
6    (and
7      [...]
8
9      ;; Enventual Invariant
10     (exists ((state State) (accounts Accounts) (clients Clients))
11       (and
12         (or
13           (= state state_4)
14         )
15
16         (Accounts_Exists accounts state)
17         (=  (Accounts_id accounts state)
18             (Clients_id clients state))
19         (not
20           (=  (Accounts_name accounts state)
21               (Clients_name clients state))
```

```
          )
        )
      )
    )
))
```

Listing 4.14: SMT formula representation of Simple Bank's Absolute Invariant

```
;; Exists Clause
(exists ((state_0 State) (state_1 State) (state_2 State)
    (state_3 State) (state_4 State) (func_part_0 Functionality_Part)
    (func_part_1 Functionality_Part) (func_part_2 Functionality_Part)
    (func_part_3 Functionality_Part))
  (and
    [...]

    ;; Absolute Invariant
    (exists ((state State) (accounts Accounts) (clients Clients))
      (and
        (or
          (= state state_1)
          (= state state_2)
          (= state state_3)
          (= state state_4)
        )

        (Accounts_Exists accounts state)
        (not
          (>= (Accounts_balance accounts state) 0)
        )
      )
    )

    [...]
  )
)
```

Listing 4.15: Tool limitation sample application

```
1 //Invariant: Accounts.balancce < 100
2
3 func1_T0(){
4     assert Accounts.balancce < 50
5 }
6
7 func1_T1(){
8     Accounts.balance += 10
9 }
```

4.4 Limitations

4.4.1 Choosing f_{max}

To ensure the analysis is finite, DAVIAC imposes a limit, f_{max}, to the number of concurrent functionalities to be considered when analyzing an application. As previously stated, it is recommended that the user takes an incremental approach when analyzing their application, that is to run the tool several times with increasing f_{max} values. However, deciding the highest f_{max} value to consider is difficult because there might be a gap between two f_{max} values that discover invariant violations. For example, there may be violations for f_{max} of 1 and 2 but then only again for f_{max} 20.

Consider the simple application introduced in Listing 4.15 that has only one functionality *func1*, composed of two transactions: *T0*, that verifies whether the attribute *balance* of an *Accounts* entity is less than 50 and aborting the functionality in case the restriction does not hold. And another transaction *T1* that increments the *balance* attribute by 10. Note that *T1* only executes if the verification in *T0* passes. In this application, there is an invariant on the *balance* attribute, which states that the value of this attribute is never greater than or equal to 100. In this example, with only one invocation of *func1*, in other words, f_{max} of one, it is impossible to violate the invariant because only one increment 10 is performed to the *balance* attribute and that increment only occurs in the case where the attribute had an initial value lower than 50. However, in the case where, initially, the value of the *balance* attribute is 49, and there are six instances of the functionality executing with f_{max} of six, there are several executions where all the instances of *T0* execute before all the instances of *T1*. In this case, all the *T0* instances will validate that the *balance* attribute is less than 50, and all six instances of *T1* will execute, resulting in a combined increase of 60 to the value of the *balance* attribute making it 109 and violating the invariant. For f_{max} values up to five, no violations would be detected, and similar examples can be given with even larger intervals of f_{max} values without violations, followed by values that yield violations.

By default, a maximum f_{max} value corresponding to the number of functionalities that affect each invariant is recommended, so that the tool explores interleavings that involve at least one instance of

each functionality that affects the invariant under analysis; however, when dealing with cases such as the one presented above, this may not be sufficient. Another possible approach would be to choose a f_{max} value higher than the number of concurrent functionalities that the deployed application allows. However, such value is not easily determined in a microservice application due to its distributed nature.

An automatic approach to determine the required value would be most useful. This is discussed further in Section 6.2.1

4.4.2 Duplicate Violations

During the search for invariant violations, the same violation can be found in several different executions. However, the number of such occurrences is severely reduced by the use of the proposed incremental analysis, as shown in Section 5.1. Currently, DAVIAC presents all the discovered executions that cause violations, which can result in different versions of the same anomaly being presented to the user. This limitation will be addressed in future work, as discussed in Section 6.2.1.

5

Evaluation

Contents

This Chapter discusses the experiences performed to evaluate DAVIAC. The machine used in said experiences has a Intel Core i7-9750H CPU @ 2.60GHz and 43GB of DDR4 memory.

5.1 Pertinence of the Results

To evaluate DAVIAC's capacity to detect invariant violations in real applications and validate the use of the proposed incremental analysis, we ran DAVIAC on the Quizzes Tutor[1] application, whose microservice implementation is available in [21], which was translated to a syntax supported by DAVIAC. This application is composed of four functionalities and four invariants (two absolute and two eventual), where, on average, each invariant is accessed for writing or reading by three functionalities. Applying the incremental analysis, the results are as presented in Table 5.1, while the results of directly running each f_{max} value are presented in Table 5.2.

[1] https://quizzes-tutor.tecnico.ulisboa.pt/

f_{max}	#Violations	Runtime	#Violations after Fix	Runtime after Fix	Total Runtime
1	1	0.557s	0	0.501s	1.058s
2	9	7.048s	0	8.626s	15.674s
3	0	3725.432s aprox 1h	–	–	3725.432s aprox 1h

Table 5.1: Quizzes Tutor incremental analysis results

f_{max}	#Violations	Runtime
1	1	0.684s
2	27	10.529s
3	464	3973.447s aprox 1h

Table 5.2: Quizzes Tutor non incremental analysis results

Looking at these results, two things become clear: one is that DAVIAC is indeed capable of analyzing real applications in useful time, the combined analysis time for f_{max} up to three is little over an hour, using the incremental analysis (excluding the time to fix the invariant violations) and it discovers relevant and real invariant violations. The other is that the use of incremental analysis removes 254 duplicate violations that the programmer would have to manually check if they opted to directly analyze the application with f_{max} value of three.

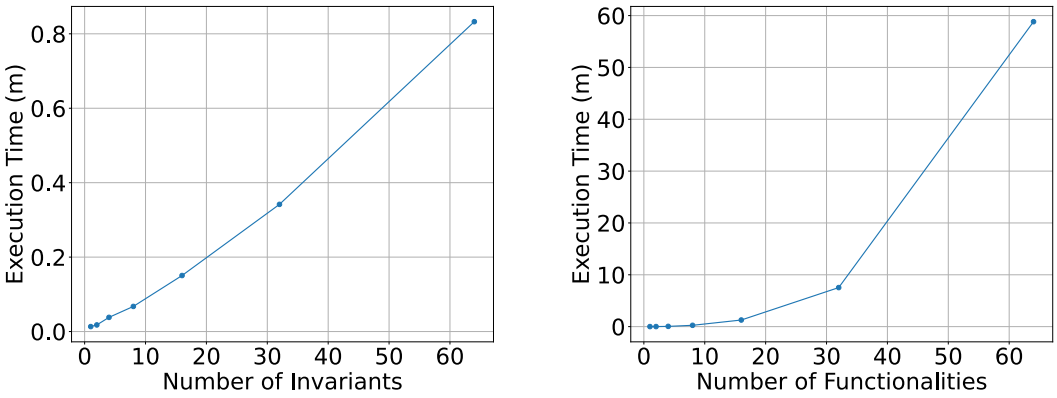
The reduction of duplicates caused by the use of incremental analysis is expected, as violations are detected using the minimal number of required functionalities, stopping the propagation of the bug to executions with more functionalities. By doing so, the developer can discover the same violations without having to analyze as many cases.

Even though Quizzes Tutor is a small application when compared to some of the microservice compositions that exist in the industry, its functionalities are of similar complexity, showing the capability of DAVIAC handling real, complex applications. While larger applications may take more time to analyze (as we will explore in the next section), we believe the parallelization of the analysis may significantly reduce the analysis time complexity, as discussed in Section 6.2.1.

These results confirm DAVIAC's ability to analyze real applications and detect invariant violations within an acceptable time, and prove that the incremental analysis is effective in fighting the occurrence of duplicate violations.

5.2 Tool Performance

Beyond the ability to detect invariant violations, analysis time is critical for a development aid tool such as DAVIAC. As such, to evaluate the execution time scaling with the application size, a synthetic application with a varying number of functionalities, varying functionality length, and varying number of invariants



(a) Vs. Invariants

(b) Vs. functionalities

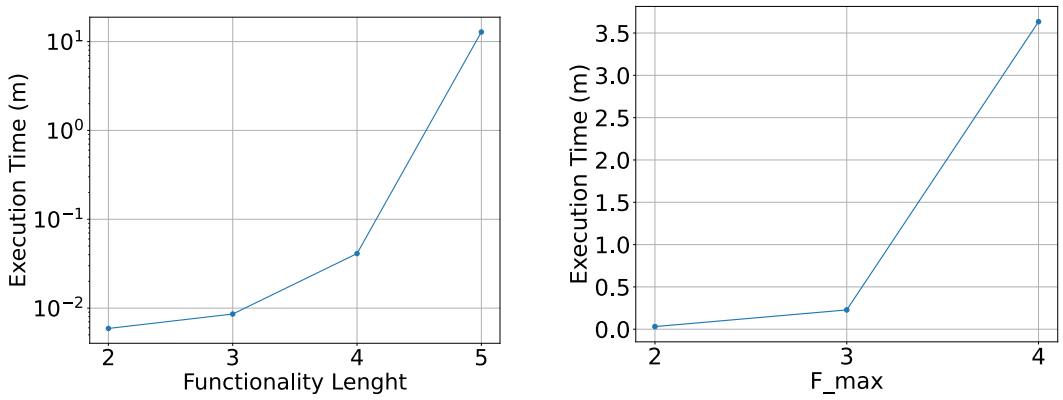
Figure 5.1: Execution time in terms of number of Invariants and Functionalities

was analyzed with DAVIAC, and the analysis times were recorded. For the first three experiments, the value of f_{max} is fixed at two, and the application has no violations reported for f_{max} value of one. The initial synthetic application consists of one absolute invariant, which relates two attributes, and a functionality composed of two transactions, each updating one of the invariant's attributes.

First, the impact of the number of invariants on an application was measured. For that, the number of invariants in the original application was increased while maintaining the number of functionalities that interact with each invariant unchanged, as well as the number of transactions in each functionality (two). The results of this experience are shown in figure 5.1(a), where the tool's execution time is measured in terms of the number of invariants, which increase up to sixty-four. The performance of the tool grows linearly with the increasing number of invariants, as is to be expected given that the analysis of each invariant is independent of the previous and given the number of functionalities that affect each invariant is the same, the analysis time for each invariant is approximately the same. Given the possibility of running the analysis for each invariant in parallel, the analysis time can be reduced linearly with the increase of logical processors, as discussed in Section 6.2.

Second, the impact of the number of functionalities that interact with each invariant on performance was measured. For that, we fix our application to our single original invariant and increase the number of functionalities by introducing extra functionalities with the same behavior as that of the original, each composed of two transactions. The results of this experience are shown in figure 5.1(b), where the tool's execution time is measured in terms of the number of functionalities, which increase up to sixty-four. As expected, the execution time displays approximately exponential growth due to the increase of possible functionality combinations and consequentially executions that DAVIAC has to cover.

Third, the impact of the length of the functionalities was measured. For that, we maintain a single



(a) Vs. Functionality Length

(b) Vs. f_{max}

Figure 5.2: Execution time in terms of Functionality Length and f_{max}

invariant and two functionalities, both affecting the same invariant, increasing the number of transactions in each functionality. The results of this experience are shown in figure 5.2(a), where the tool's execution time is measured in terms of the length of the functionalities, which increase up to five. The execution time displays approximately exponential growth due to increased possible functionality interleavings and consequentially executions that DAVIAC has to cover. However, this growth is expected to be less significant than the one introduced by increasing the number of functionalities, as the interleavings between transactions must still respect the order of execution within a functionality, while functionalities may be interleaved in any order. We will further explore this difference in growth later in the section.

We now evaluate the impact of the f_{max} value in the analysis time. For that, we use an application with four functionalities and two transactions each, all affecting the same unique invariant. As expected and as can be observed in Figure 5.2(b), the execution time shows an approximate factorial growth due to the increasing number of functionalities and transactions involved in each execution, causing a significant increase in the possible functionality interleavings that DAVIAC has to cover. This result has a very direct impact on the analysis; it means that more extensive analyses, in other words, those that cover higher f_{max} values, will see a significant increase in execution time for each analysis step as the value of f_{max} increases.

Finally, we test whether the number of functionalities causes a greater increase in execution time than their length while maintaining the number of transactions involved in each formula constant. Which was not clear from the previous experiments given that the number of transactions involved in each formula was not constant.

To do so, we use four applications, each composed by the same eight transactions but evenly distributed over a varying number of functionalities, increasing from 1 to 8 functionalities. Each application

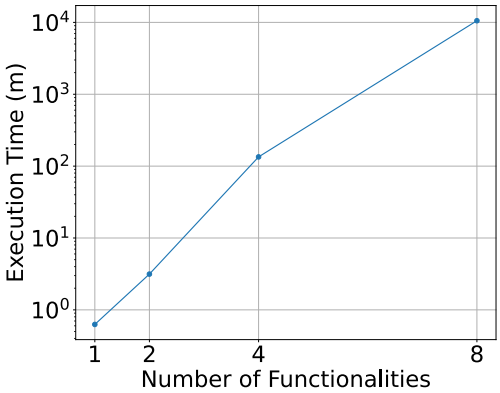


Figure 5.3: Execution time in terms of Number of Functionalities with constant Total Transaction Number

was analyzed using f_{max} values such that the analysis involve all functionalities in each formula. The results are presented in Figure 5.3, and as expected, as the number of transactions are increasingly distributed along more functionalities, the increased space for interleavings leads to an increase in time complexity. This verifies that increasing the number of functionalities has a more significant time complexity impact than increasing the number of transaction in each functionality.

6

Conclusion and Future Work

Contents

6.1 Conclusion

This thesis describes the design, implementation, and evaluation of DAVIAC, a tool for formal verification of microservice applications that discovers all possible invariant violations. By mapping JAVA code to SMT statements, the tool is capable of analyzing real applications and does so in useful time, presenting graphic visualizations of the results. As far as we know, DAVIAC is the first tool capable of performing a complete invariant analysis of a microsservice application directly from its source code.

6.2 Future Work

Future work for this project is divided into two categories: Addressing Current Limitations, in Section 6.2.1, where proposed solutions for the current limitations of DAVIAC are addressed; and Expanding

The Tool, in Section 6.2.2, where we address possible extensions to the tool.

6.2.1 Addressing Current Limitations

Currently, it is recognized that DAVIAC has four limitations that need to be addressed in future work: picking the highest f_{max} value to consider, the inability to recognize different versions of the same violation, the reduced set of allowed inputs, and the analysis time.

6.2.1.A Picking the Highest f_{max} to Consider

In order to guarantee termination of the SMT solver execution, the number of concurrent functionalities in each formulation must be finite. However, this means that the user must decide on which f_{max} value to stop the analysis which, as previously mentioned, may result in some invariant violations with higher concurrent number of functionalities to remain undetected. As such, future studies on automated approaches to analyze the content of transactions and look for clauses similar to those presented in Section 4.4.1 in order to indicate the ideal f_{max} would be most advantageous. Another possible approach to this problem, albeit not ideal, could be to analyze many real-world microservice applications and determine the average max f_{max} value for which invariant violations are still discovered and use that value as a reference. Lastly, some related work [26, 27] presents approaches to determine how certain transactions can potential violate some invariant or to prove that they cannot. Future work should evaluate if such approaches could be applied to the Microservices Architecture, where the impact of the whole functionality has to be considered, not that of a single transaction.

6.2.1.B Recognizing Different Versions of the Same Violation

Naturally, the same invariant violation can occur in different executions. However, DAVIAC lacks the ability to recognize which invariant violation instances correspond to the same violation, which leads to duplicate violations being counted as new violations. Currently, DAVIAC groups violations that are discovered for the same set of invariant and functionalities. During the development of the tool, it was observed that in most cases, the detected violations were in fact duplicates. However, some cases still presented distinct violations for the same invariant and functionalities. To address this, a mechanism to detect equivalent violations should be developed in future work. A good start for this mechanism would be to group equivalent executions.

6.2.1.C Augmenting the Allowed Inputs

Currently, DAVIAC only supports the analysis of applications in which the entire code base is developed in JAVA and makes direct use of SQL statements to manipulate its entities. However, most microservice

applications are not developed exclusively using the JAVA programming language, and even those that use JAVA tend to use ORM frameworks. To address this, future work on DAVIAC will add a parser for JAVA code that supports the use of the Spring framework in JAVA. This expansion will allow further testing of the tool on popular benchmark microservice applications such as the Train Ticket application [1] or other real-world code bases that are publicly available. Further work on this aspect should include parsers for other popular programming languages and frameworks. This process is expected to be simple and mechanical, not involving much, if any, engineering work. This is due to the existence of the internal representation, which makes the input parsers independent from the analysis, and similar parses have been done in related work [18].

6.2.1.D Parallelization of the Analysis

One of the concerns with using SMT solvers is the high temporal complexity of solver. To address this, the formulas used by DAVIAC are as simple as possible, meaning that the solver execution time for each formula is short. However, as the application grows in size, this leads to an increase in the number of formulas to analyze. Given that several instances of the solver can be run concurrently, DAVIAC's next step should include implementing a pool of solver instances to analyze the formulas concurrently. This would reduce the analysis time linearly with the increase of workers. Further improvements could be gained by paralyzing the formula generation process, namely the process of computing all the possible functionality executions.

6.2.2 Expanding the Tool

To make DAVIAC even more appealing to the microservice community, we believe two particular features should be added to the tool. The first is the ability to support different isolation levels. The second is the ability to support more complex microservice orchestrations than linear sequences.

6.2.2.A Support for Varying Isolation Levels

Presently, DAVIAC assumes all transaction executions are serializable; however, this is not industry standard. In fact, due to efficiency concerns, developers often opt to execute some transactions with weaker isolation levels, such as read committed. A useful feature for DAVIAC would be the ability to indicate the desired isolation level for each transaction. This would allow the tool to evaluate the impact of different transaction isolation levels on the number of invariant violations.

[1] https://github.com/FudanSELab/train-ticket

6.2.2.B Support for Other Microservice Orchestrations

DAVIAC only supports functionalities that are a linear sequence of transactions. Corrective and branching transactions are not currently supported and are often used in microservice deployments. While this feature is not critical for the tool to yield useful results, it would be a useful addition as it would make the tool more complete.

Bibliography

[1] M. Fowler, “Microservices,” Web page: http://martinfowler.com/articles/microservices.html, accessed: 2024-06-25.

[2] J. Thönes, “Microservices,” *IEEE Software*, vol. 32, no. 1, Jan. 2015.

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

[4] T. Harder and A. Reuter, “Principles of transaction-oriented database recovery,” *ACM Computing Surveys*, vol. 15, no. 4, 1983.

[5] M. Fowler, “Microservice trade-offs,” Web page: https://martinfowler.com/articles/microservice-trade-offs.html, accessed: 2024-05-23.

[6] K. Gos and W. Zabierowski, “The comparison of microservice and monolithic architecture,” in *16th MEMSTECH*, Lviv, Ukraine, Apr. 2020.

[7] N. 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, Sep. 2021.

[8] 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, Jun. 2023.

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

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

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

[12] S. Shah, K. Anastasakis, and B. Bordbar, “From uml to alloy and back again,” in *MoDeVVA 2009*, Denver, Colorado, USA, Oct. 2009.

[13] K. Anastasakis, B. Bordbar, G. Georg, and I. Ray, “Uml2alloy: A challenging model transformation,” in *MODELS 2007*, Nashville, Tennessee, USA, May 2007.

[14] Y. He, “Comparison of the modeling languages alloy and uml.” *Software Engineering Research and Practice*, vol. 2, Jun. 2006.

[15] A. Panda, M. Sagiv, and S. Shenker, “Verification in the age of microservices,” in *16th HotOS*, Whistler, BC, Canada, May 2017.

[16] V. Romão, R. Soares, V. Manquinho, and L. Rodrigues, “Deteção automática de anomalias em arquiteturas de microsserviços,” in *14th Inforum*, Porto, Portugal, Sep. 2023.

[17] A. Adya, “Weak consistency: A generalized theory and optimistic implementations for distributed transactions,” Ph.D. dissertation, MIT, Mar. 1999.

[18] K. Ma, C. Li, E. Zhu, R. Chen, F. Yan, and K. Chen, “Noctua: Towards automated and practical fine-grained consistency analysis,” in *EuroSys 2024*, Athens, Greece, Apr. 2024.

[19] P. Das, R. Laigner, and Y. Zhou, “Hawkeda: A tool for quantifying data integrity violations in event-driven microservices,” in *15th DEBS*, Virtual Event, Italy, June 2021.

[20] A. Ribeiro, “Invariant-driven automated testing,” Master’s thesis, Universidade NOVA de Lisboa, 2021.

[21] P. Pereira and A. Silva, “Transactional causal consistent microservices simulator,” in *23rd DAIS*, Lisbon, Portugal, Jun. 2023.

[22] F. Montesi, C. Guidi, and G. Zavattaro, “Service-oriented programming with jolie,” *Web Services Foundations*, 2014.

[23] S. Giallorenzo, F. Montesi, M. Peressotti, F. Rademacher, and N. Unwerawattana, “JoT: A Jolie framework for testing microservices,” in *26th COORDINATION*, Jun. 2023.

[24] L. Moura and N. Bjørner, “Z3: An efficient smt solver,” in *14th TACAS*, Budapest, Hungary, Apr. 2008.

[25] S. Rogers, “The pros and cons of object relational mapping (orm),” Web page: https://midnite.uk/blog/the-pros-and-cons-of-object-relational-mapping-orm, 2019, accessed: 2024-06-25.

[26] F. Houshmand and M. Lesani, “Hamsaz: replication coordination analysis and synthesis,” *PACMPL*, vol. 3, no. 74, Jan. 2019.

[27] A. Gotsman, H. Yang, C. Ferreira, M. Najafzadeh, and M. Shapiro, "'cause i'm strong enough: Reasoning about consistency choices in distributed systems," *SIGPLAN Not.*, vol. 51, no. 1, Jan. 2016.