Detection of Invariant Violations in Microservices

(extended abstract of the MSc dissertation)

João Fitas
Departamento de Engenharia Informática
Instituto Superior Técnico

Advisors: Professor Luís Rodrigues and Professor António Rito da Silva

***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 SMT 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.**

I. Introduction

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

As such, the goal of this work is to automatically identify and help the developer handle all invariant violations in their application. This is a complex task, as it it involves the analysis of all the possible ways the invariants can be affected during the application's execution, especially when the goal is full coverage.

II. Related Work

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. So, tools following both approaches were covered while performing this work. Furthermore tools related to the development and testing of microservices were also analyzed. To sum up the most relevant characteristics of the tools covered, their characteristics are presented in Table I, 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 I: 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.

Harmony [5], Ucheck [6], and Alloy [7] define invariants at the entity level, while HawkEDA [8] and PETIT [9] define them at the API level. As a consequence, the first group does not require, procedure calls to verify the invariants, it can directly verify the state of the attributes, making the invariant verification process simpler and faster.

Furthermore, black-box approaches are unable to achieve a complete analysis, as they are based on testing approaches. Meaning that such an approach could never fulfill this work's completeness goal.

The Simulator [10] and JoT [11] 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, which aims to automate the detection process.

With all this in mind, the tools that are more related to the goal of detecting all invariant violations in an application are MAD [12], Harmony [5], Alloy [7], and Noctua [13]. Within these, it is worth highlighting that MAD [12] and Noctua [13] are the only ones that support source code analysis and do not require a manually generated model of the application.

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 [7] and Harmony [5] 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 [10]. Another would be to use a generic framework like Jot [11] that can be used regardless of the technologies used by each microservice. 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.

Considering all this aspects, Harmony [5], MAD [12], Noctua [13], and Alloy [7] are the tools closer to achieve the goals of this work, however they all lack some aspect. 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 [12] and Noctuas's [13] 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 [12] and Noctua [13] use an internal representation optimized for SMT solvers, this work will also follow that approach. This tool is presented in the next Section.

III. DAVIAC

This section presents a tool produced by this work for the automatic detection of invariant violations in business logic rich applications, 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) [14]. 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.

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 III-A1. 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 III-C1.

A. Architecture

The architecture and execution flow of the DAVIAC tool are represented in Figure 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.

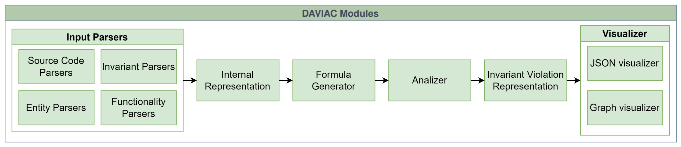


Figure 1: DAVIAC Module Sequence

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

- **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. Entities are usually represented in SQL or ORM) [15]. 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. Currently, only basic types are supported by the tool.

- **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. 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 "! =".
- **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 that support code using popular frameworks, like Spring[2], Django[3], among others that are common throughout the microservice world.
- **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. It receives a JSON file containing the functionalities and their transactions, presented as a list sorted by sequence order.

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.

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

[1]https://javaparser.org/

[2]https://spring.io/

[3]https://www.djangoproject.com/

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 VI.

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 III-C1.

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

Design details on the SMT formulas are covered in Section III-B.

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

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.

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 III-A6, and one that generates a graph of the violation III-A6.

- **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.
- **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 2, 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.

B. 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.

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*

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

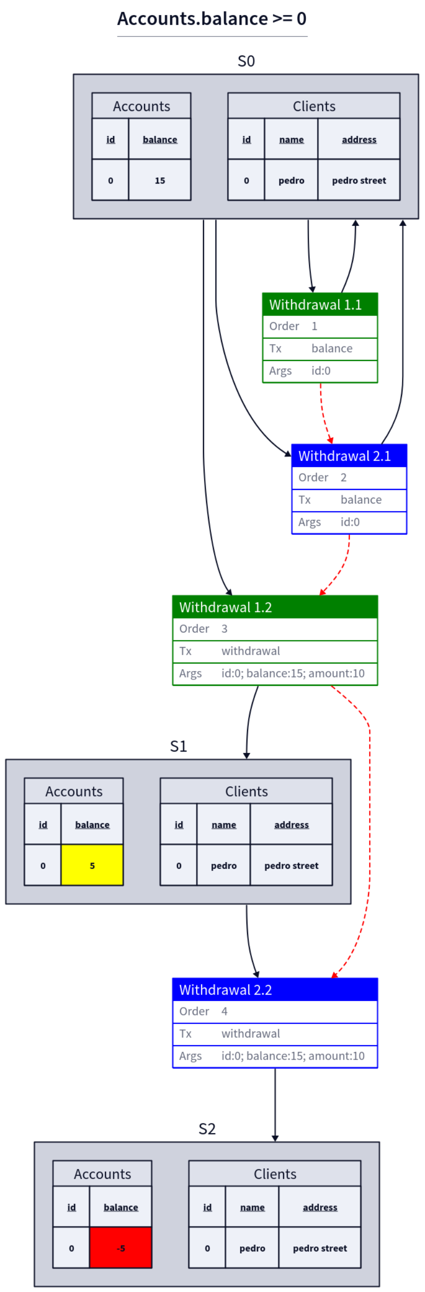


Figure 2: Graph visualization of a Simple Bank Invariant violation

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.

- **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 a simple bank application, an *Accounts* Entity is represented by the Accounts sort, the exists function and the one function for each of its attributes, as shown in Listing 1. Listing 2 contains the unique clause for the *id* attribute of the *Accounts* entity. Lastly, Listing 3 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.

```
(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 1: SMT formula representation of Simple Bank's Accounts Entity

```
(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 2: SMT formula representation of a unique 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)
)))))
```

Listing 3: SMT formula representation of a foreign atribute

- **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_Parts* 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, from the same Simple Bank App, in the formula can be seen in Listing 4

```
(declare-fun balance () Int)
(declare-fun balance_id_account (Functionality_Part) Int)
(declare-fun balance_balance (Functionality_Part) Int)
(assert (forall ((func_part Functionality_Part))
  (=>
    (= (get_func_part_type func_part) balance)
    (exists ((start_state State) (end_state State))
      (ite
        (and
          (= (start_state func_part) start_state)
          (= (end_state func_part) end_state)
        )
        (forall ((clients Clients) (accounts Accounts))
          (and
            (=>
              (and
                (Accounts_exists accounts start_state)
                (=  (Accounts_id accounts start_state)
                    (balance_id_account func_part))
              )
              (=  (Accounts_balance accounts start_state)
                  (balance_balance func_part))
            )
            (=  (Accounts_exists accounts start_state)
                (Accounts_exists accounts end_state))
            (=  (Accounts_id accounts start_state)
                (Accounts_id accounts end_state))
            (=  (Accounts_balance accounts start_state)
                (Accounts_balance accounts end_state))
            (=  (Accounts_name accounts start_state)
                (Accounts_name accounts end_state))
            (=  (Clients_exists clients start_state)
                (Clients_exists clients end_state))
            (=  (Clients_id clients start_state)
                (Clients_id clients end_state))
            (=  (Clients_address clients start_state)
                (Clients_address clients end_state))
            (=  (Clients_name clients start_state)
                (Clients_name clients end_state))
        ))
        false
)))))
```

Listing 4: SMT formula representation of Simple Bank's balance transaction

- **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 5 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 2.

```
;; 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)
    (= (get_func_part_type func_part_2) withdrawal)
    (= (get_func_part_type func_part_3) withdrawal)
    (= (get_func func_part_0) (get_func func_part_2))
    (= (get_func func_part_1) (get_func func_part_3))
    [...]
)))
```

Listing 5: SMT formula representation of Simple Bank's Execution Order Example

- **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 6.

```
;; 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
    [...]
    ;; Initial State Invariants
    (forall ((accounts Accounts) (clients Clients))
      (=>
        (=  (Accounts_id accounts state_0)
            (Clients_id clients state_0))
        (=  (Accounts_name accounts state_0)
            (Clients_name clients state_0))
    ))
    (forall ((accounts Accounts))
      (>= (Accounts_balance accounts state_0) 0)
    )
    [...]
))
```

Listing 6: SMT formula representation of Simple Bank's Initial State Invariants

- **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 7 and the absolute in Listing 8.

```
;; 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
    [...]
    ;; Enventual Invariant
    (exists ((state State) (accounts Accounts) (clients Clients))
      (and
        (or
          (= state state_4)
        )
        (Accounts_Exists accounts state)
        (=  (Accounts_id accounts state)
            (Clients_id clients state))
        (not
          (=  (Accounts_name accounts state)
              (Clients_name clients state))
))))))
```

Listing 7: SMT formula representation of Simple Bank's Eventual 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 8: SMT formula representation of Simple Bank's Absolute Invariant

C. Limitations

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.

```
//Invariant: Accounts.balancce < 100
func1_T0(){
    assert Accounts.balancce < 50
}
func1_T1(){
    Accounts.balance += 10
}
```

Listing 9: Tool limitation sample application

Consider the simple application introduced in Listing 9 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 VI-A

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 IV-A. 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 VI-A.

IV. Evaluation

A. 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[5] application, whose microservice implementation is available in [10], 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 II, while the results of directly running each f_{max} value are presented in Table III.

[5]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 II: Quizzes Tutor incremental analysis results

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

Table III: Quizzes Tutor non incremental analysis results

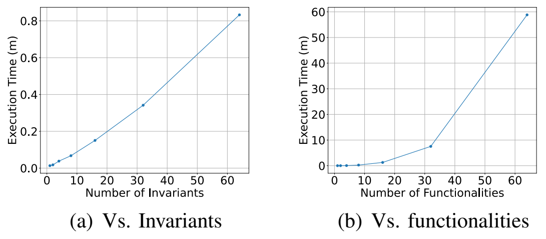


Figure 3: Execution time in terms of number of Invariants and Functionalities

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 VI-A.

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.

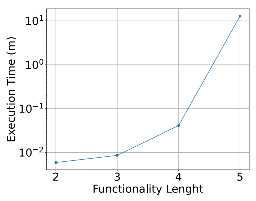
B. Tool Performance

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 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.

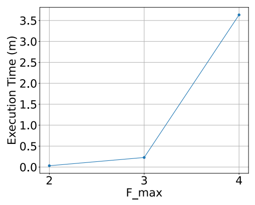
First, the impact of the number of invariants on an application was measured. For that, the number of invariants in the original application was increased while maintaining the number of functionalities that interact with each invariant unchanged, as well as the number of transactions in each functionality (two). The results of this experience are shown in figure 3(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 VI.

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 3(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 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 4(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.



(a) Vs. Functionality Length



(b) Vs. f_{max}

Figure 4: Execution time in terms of Functionality Length and f_{max}

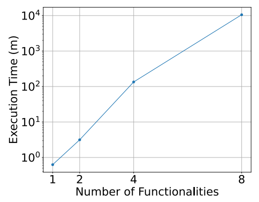


Figure 5: Execution time in terms of Number of Functionalities with constant Total Transaction Number

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 4(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 was analyzed using f_{max} values such that the analysis involve all functionalities in each formula. The results are presented in Figure 5, 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.

V. Conclusions

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.

VI. Future Work

Future work for this project is divided into two categories: Addressing Current Limitations, in Section VI-A, where proposed solutions for the current limitations of DAVIAC are addressed; and Expanding The Tool, in Section VI-B, where we address possible extensions to the tool.

A. 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.

1) 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 III-C1 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.

2) 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.

3) 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 [6] 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 [13].

4) 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.

B. 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.

1) 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.

2) 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.

Acknowledgments

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). Parts of this work have been performed in collaboration with other members of the Distributed Systems Group at INESC-ID, namely, Rafael Soares.

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

References

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

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

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

[8] 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.

[9] A. Ribeiro, "Invariant-driven automated testing," Master's thesis, Universidade NOVA de Lisboa, 2021.

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

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

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

[13] 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.

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

[15] 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.