MicroValid: A Validation Framework for Automatically Decomposed Microservices

Michel Cojocaru
Universiteit van Amsterdam
michel.cojocaru@student.uva.nl

Alexandru Uta
Vrije Universiteit Amsterdam
a.uta@vu.nl

Ana Oprescu
Universiteit van Amsterdam
a.m.oprescu@uva.nl

***Abstract*—In a dynamic world of software development, the architectural styles are continuously evolving, adapting to new technologies and trends. Microservice architecture (MSA) is gaining adoption among industry practitioners due to its advantages compared to the monolithic architecture. Although MSA builds on the core concepts of Service Oriented Architecture (SOA), it pushes for a finer granularity, with stricter boundaries. Due to cost rationale, numerous companies choose to migrate from the monolithic style instead of developing from scratch. Recently, semi-automatic decomposition tools assist the migration process, yet a crucial part is still missing: validation. The current study focuses on providing a validation framework for microservices decomposed from monolithic applications and complete the puzzle of architectural migrations. From previous work we select quality attributes of microservices that may be assessed using static analysis. We then provide an implementation specification of the validation framework. We use five applications to evaluate our approach, and the results show that our solution is scalable while providing insightful measurements of the assessed quality attributes of microservices.**

***Index Terms*—microservices, decomposition, validation, coefficient of variation, granularity, coupling, cohesion**

I. Introduction

The current pull in cloud computing is toward the newer paradigms of serverless or function-as-a-service (FaaS) computing [1]. These paradigms are gaining rapid traction in the industry, with the FaaS market value estimated at \$7 billion by 2021[1], and the serverless architecture market at \$20 billion by 2025[2]. Such cloud computing paradigms rely heavily on large-scale applications being expressed as microservices instead of traditional monoliths. As a consequence, both industry and academia are working toward designing tools that automatically decompose large code monoliths into microservices. Although several such tools already exist, the community is still investigating what are the desired properties [2] of the decomposition tools, and also how to automatically validate their achieved results. Bridging this gap, in this article we propose a framework for automatically validating both qualitatively and quantitatively the microservices resulted through automatic decomposition of large-scale code monoliths.

Microservice Architectures (MSA) are constantly increasing in adoption, and are the most chosen architecture by industry practitioners who develop enterprise applications providing advantages such as agility and maintainability [3]. While new large scale enterprise applications are increasingly relying on MSA, there still exist legacy applications. The companies embracing the architectural advantages of MSA are considering migrating their legacy code. Often the migration process starts with adding microservices to the monolithic application and gradually moving functionality to the microservices.

The legacy application is kept in production for the entire time span of the migration, in parallel with the modules in development. Along with this architectural shift, the companies can benefit from code refactorization and new features without incurring extra costs [3]. Companies are more prone to choose migration towards MSA rather than rebuilding the application from scratch mainly due to cost rationale. To aid this migration process from monoliths to microservices, semi-automatic decomposition tools have emerged, an example being the Service Cutter [4]. Using clusterization algorithms, this tool ”cuts” a monolith in microservices suggested to the architect conducting the migration.

Although recent work attempts automation of the decomposition processes [5], the lack of unanimously accepted guidelines for defining a good microservice means there is a crucial need for the validation of such tools. Therefore, this study focuses on providing a validation framework for microservices resulted from migrations from monoliths. Our solution addressed the need for unbiased validation of the results of (semi-)automatic decomposition tools. Our contributions are the following:

- We provide an open-source framework[3] for validating suggested ”cuts” of monoliths obtained via decomposition with a primary focus on static analysis attributes.
- We consider in our implementation the prospect of future decomposition tools, therefore striving to provide a unified format for which the subsequent tools should only write an integration plugin to extend our framework.

Outline. Section II presents the architecture of MicroValid; we describe and discuss in detail the results of our extensive evaluation in Section III, discuss related work in Section IV, and conclude in Section V.

[1]https://www.marketsandmarkets.com/Market-Reports/function-as-a-service-market-127202409.html

[2]https://www.grandviewresearch.com/press-release/global-serverless-architecture-market

[3]https://micro-valid.herokuapp.com/,https://github.com/michelcojocaru/

II. The MicroValid Architecture

We describe the MicroValid architecture from two perspectives: design choices and assessment algorithms. We discuss several design choices typical for frameworks in Section II-A. An important design choice is addressing the need of relative assessments in contrast to absolute values [2]. Our assessment algorithms (Section II-B) focus on calculating the **coefficient of variation** which identifies outliers in the scores our framework produces, thus revealing possible issues of a component relative to its peers inside the same system.

A. Design Choices

Our framework allows any user to assess various quality attributes using various assessment tests for any given OOP project. The input is a system model specification file. The output is a mapping of quality attributes and assessments.

Based on previous work [2], we choose the Service Cutter's JSON format as input. Their "entities" depict classes, while the "nanoentities" depict their respective fields/attributes. Relations depicting the communication paths inside the system are simple to comprehend, yet descriptive enough to completely specify the data exchange flows of the system. It provides an unified format, easily comprehensible by any developer [4].

1) Modularity: The modularity of a software system is a characteristic describing the separation of functionality in independent and interchangeable modules, each containing all the requirements for delivering one aspect of the functionality. We choose to develop our validation framework by decoupling the back-end (microservices assessment logic) from the front-end (user interaction capabilities) and introducing an API via which the two interchange data through HTTP calls.

2) Extensibility: Software extensibility is the design principle which takes into consideration the future growth of the system. There are two types of extensibility: the extension of the system via reutilizing modules to build new functionality, or thorough changes of the codebase that do not alter the existent functionality but enrich it. Both types focus on minimizing the impact of changes to existing functionality.

We provide extensibility at several levels. The input and output formats we choose provide extensibility via JSON's intrinsic extensibility. This design choice allowed incremental development of the assessment strategy by using TAR [6].

Our framework can be extended with other quality attributes assessments. We employed the design pattern "Chain of responsibility" [7] to chain the attribute assessment tests and provide clear guidelines for users who want to extend our work. The framework uses independent *checkers* that can be regarded as links in a chain. Anyone who wants to extend our validation framework would only need to implement their own checker (which may assess a new quality attribute or assess differently an existing quality attribute) and simply register the checker in the **checker chain** with one line of code.

Our framework provides a module allowing users to call external programs or routines irrespective of the programming language. Our framework also provides a module for remote installation of external programs for the cases where the deployment is done on Virtual Machines which allow local installation of external dependencies.

3) Usability: Usability is the ease of use and comprehension by its users. By decoupling the front-end from the back-end we avoid a possible technology "lock-in".

4) Testability: We define testability as the ease of assessing the correct behaviour and functioning of our framework. By providing a loosely coupled architecture where the assessment logic is decoupled from the user interaction (configuration) with the analyzer, our framework uses an API to communicate between the front-end and the back-end. The API was augmented with Swagger Editor [8] for easy design and testing. Our framework is compliant with the OpenApi Specification standard for APIs.

B. Assessment Algorithms

Our validation approach focuses on three quality attributes of microservices: *granularity*, *coupling*, *cohesion*. Each quality attribute is assessed by one *checker* using at least two metrics. We avoid setting absolute thresholds for the assessments [2]. We rely on the **coefficient of variation** to identify outliers in the results produced by our framework. The coefficient of variation (C_v) is defined as the ratio between the standard deviation (σ) and the average (μ).

1) Granularity: The assessment is composed of two tests, both describing the concept of size: the **nanoentities composition** test, which uses as input the system model specification files, and the **Lines of Code** (LoC) test, which uses the code base (possibly in a repository stored on versioning systems such as GitHub) to count the lines of code.

The **nanoentities composition** test calculates the number of nanoentities assigned to each proposed service and stores the result in the form of a floating point parameterized list. The length of the list is equal to the number of services found in the system model specification file. The algorithmic steps for assessing the size of microservices using the system specifications file are:

- Calculate the lengths of the nanoentity lists belonging to each proposed microservice
- Calculate the coefficient of variation for these lengths
- Map result from the [0,1] range to a [0,10] range

The algorithmic steps for assessing the size of microservices using the code base are:

- Download code base repository
- Match the entities of each microservice to classes from repository
- Calculate the **LoC** (of all classes belonging to a microservice) per microservice
- Calculate the coefficient of variation for the **LoC** list
- Map result from the [0,1] range to a [0,10] range

The flow of using this test requires the initial validation of a decomposed system, followed by a re-implementation or automated decomposition of application's code base. Hence this test only serves as a final validation for the implementation decomposition with respect to granularity.

2) Coupling: Low coupling is highly desirable in Microservice Architecture [9]. Similarly to the *granularity* checker, the *coupling* checker contains two tests: **dependencies composition** and **strongly connected components (SCC)**.

The **dependencies composition** test assesses how balanced outward dependencies are across the microservices, by counting the outward dependencies each microservice has toward its peers. The algorithm constructs a "dependency graph" of the system where each dependency represents a communication path utilised for exchanging data between two components of the system. The algorithmic steps for assessing the dependencies composition of microservices are:

- Calculate the number of outward dependencies for each microservice
- Calculate the coefficient of variation for the outward dependencies
- Map result from the [0,1] range to a [0,10] range

The rationale behind counting only outward dependencies (ignoring the inward dependencies) is avoiding cycles containing only two components which might incorrectly flag inconsistencies in the system.

The **SCC** test uses Tarjan's algorithm [10] to identify strongly connected components on the previously mentioned "dependency graph". If cycles are detected in the communication paths, the services in question should be aggregated together into one microservice. The score is calculated as the ratio between the number of identified strongly connected components divided by the total number of microservices inside the system. The algorithmic steps for identifying strongly connected components are:

- Construct "dependency graph" based on services and relations descriptions
- Run Tarjan's algorithm on the graph to obtain the connected components (and their sub-components)
- Calculate score $\in$ [0,1] by dividing the number of components to the number of microservices
- Map result from the [0,1] range to a [0,10] range

Ideally, each microservice would constitute its own strongly connected component, thus obtaining a maximum score.

3) Cohesion: A high cohesion score is desirable in a microservices architecture [9]. The *cohesion* checker contains tests reflecting four metrics, aggregated to assess this quality attribute of microservices: nanoentities composition, relation composition, responsabilities composition, semantic similarity.

The **entities composition** test assesses whether the entities are equally distributed among the proposed microservices and no duplicates, which might break the cohesion, exist. It uses a strategy similar to the one employed in the assessment of *granularity* in **nanoentities composition** test, the difference consisting of the data it is applied on, namely entities instead of nanoentities. We define an *entity* as the class, or action of the service.

We may regard the entity as the class of a Java program and the nanoentities as the attributes (or fields) of that class.

A set of entities is created in form of a floating point value list for which the **coefficient of variation** is calculated. If an entity is detected as belonging to several different microservices, the test fails, implying a break of cohesion at component level.

The **relation composition** test assesses the quantitative variation in published language per relation. It applies the concept of relative assessment to entities shared between the services via their communication paths. The shared entities, named "published language" by Service Cutter [4], are extracted from relations which describe the communication paths in the system model specifications file.

This test is particularly useful for identifying the services which communicate much more data than their peers, signalling a possible communication bottleneck.

The **responsibilities composition** test assesses to which extent the use case responsibilities are equally distributed among the proposed microservices. For scoring, it uses the **coefficient of variation** between the number of use case responsibilities of each microservice.

It is useful to identify the services that have more responsibility compared to others, which might imply low cohesion: a service providing multiple actions contradicts the single responsibility principle [11].

The **semantic similarity** test uses lexical distance assessment algorithms to flag the services that contain unrelated components or unrelated actions hindering *cohesion*. The test is composed of up to eight different assessment algorithms that can be chosen by the user in any desired combination. The similarity algorithms available are: *Hirst & St'Onge*, *Leacock & Chodorow*, *Resnik*, *Jiang & Conrath*, *Lin*, *Path*, *Lesk*, *and Wu Palmer* provided in WS4J package [12] and Jaw-Jaw [13]. If multiple algorithms are selected simultaneously, the overall test score is the average of the selected similarity algorithms scores. The knowledge base used is an English and Japanese WordNet database [14].

We use a Java wrapper for Japanese WordNet instead of using a simple English knowledge base to provide a flexible framework that can be easily adapted to work with languages that use different text encodings.

III. Experimental Evaluation

A. Experimental Workload

We evaluate our validation framework using five applications inspired from real-world systems: three application models from Service Cutter [4], and two from the Dataflow Diagram decomposition research [15]. We focus on data depicting the services and their respective entities and nanoentities, relations with their direction and "published language", and respective use case responsibilities per microservice.

Each of the five input applications has at least two decomposition approaches: *deterministic* and *non-deterministic*. Additionally, two applications were also *manually* decomposed by us according to our best understanding of the Data-flow Diagram decomposition technique, for comparison purposes. For the industrial scale application, an ideal manual decomposition was constructed based on its GitHub repository. All system specifications files depicting the input applications and their respective versions are available online [4].

1) Cargo Tracker - An Industrial Scale Application: a well-known example of applied Domain Driven Design (DDD) [16]. The application consists of an industrial scale system which manages cargo shipments. Besides the non-deterministic and deterministic versions of the system model specifications files, we also created an "ideal" decomposition of the system. This was done in order to showcase how our framework would obtain maximum scores for all the tests, yielding a total of three versions for this input application.

2) Booking System: a synthetic generic example with a variety of applications in industry ranging from public administration tasks to commercial movie/theatre ticket reservations. The system model specifications come from the sample applications proposed by Service Cutter [4] as non-deterministic and deterministic decompositions, yielding two versions.

3) Trading System: a financial application synthesised by the Service Cutter [4] authors both as non-deterministic and deterministic decompositions, yielding two versions.

4) Movie Crawler System: a business logic module, part of a movie information crawling project. The system model comes from a data-flow based decomposition approach [15]. For comparison purposes, we also fed the system model specifications file to Service Cutter, yielding three versions.

5) Ticket Price Comparator: is a business logic module, part of the same movie information crawling project as the Movie Crawler System. The system model come from the data-flow based decomposition approach [15]. For comparison purposes, we also fed the system model specifications file to Service Cutter, yielding three versions of the system.

B. Experimental Results for Static Analysis

We present and discuss the results obtained by employing static analysis on the workload applications. For each application decomposition we assessed three quality attributes using checkers: **granularity**, **coupling**, **cohesion**. Each checker contains at least two tests, as described in Section II-B. We group the results by the decomposition type: deterministic, non-deterministic, manual (according to DFD [15]) and a manually idealised decomposition. We define three intervals which classify the score as follows: *score* $\in [0.0, 5.0] \rightarrow failed$, *score* $\in (5.0, 7.5] \rightarrow acceptable$, *score* $\in (7.5, 10.0] \rightarrow$ *good.*

1) Non-deterministic decompositions: User cannot control the number of microservices yielded by Service Cutter [17].

The **Cargo Tracker System** scores for all attributes are collected in Table I. Granularity scores 5.4. It contains only the *nanoentities composition* test, as no code base is available and thus no *LoC* test can be performed. The framework reveals a high variation in microservices' sizes as the cause of the low score. The disproportionate spread of nanoentities across the microservices is confirmed by the *detailed* results: 16 nanoentities in one service, and 9 in another service.

[4]https://github.com/michelcojocaru/MasterProject/tree/master/test-files/service-cuts

Coupling scores 10.0. The *dependencies composition* test scored 10.0 due to a perfect distribution of communication paths, each microservice having one outward dependency. The *SCC* test flagged no strongly connected component maintaining the average for the coupling assessment at 10.

Cohesion scores 6.3. The *entities composition* scores 4.7 due to an imbalanced entities distribution: 8 entities in one service and 3 in another service. The *responsibilities composition* test scores 4.4 due to a similarly imbalanced use-case responsibility distribution among the microservices: 2 and 7, respectively. The *relations composition* test scores 10.0 due to only one relation facilitating communication between the only two microservices of the system. The *semantic similarity* test scores 6.2 as the average of all available algorithms enabled.

Checker	Checker Score	Test Name	Test Score
Granularity	5.4	Nanoentities	5.4
Coupling	10.0	Dependencies	10.0
		Scc	10.0
Cohesion	6.3	Entities	4.7
		Responsibilities	4.4
		Relations	10.0
		Semantic	6.2

TABLE I
CARGO TRACKER: NON-DETERMINISTIC DECOMPOSITION SCORES.

Table II shows the **Booking System** scores. Granularity scores 5.4. Its only component, the *nanoentities composition* test, scores just below the threshold (5.5) due to an imbalance in nanoentities distribution across services (7 and 9, respectively). With no source code available, the *LoC* test is absent.

The coupling scored 10.0. The *dependencies composition* scored 10 due to the presence of a single relation between the two microservices of the system. Furthermore, the single relation also justifies the perfect score of the *SCC* test due to the mathematical impossibility of having a cycle in graph with two nodes and a single, unidirectional communication path.

Cohesion scores 7.0. The *entities composition* test scores 5.3 due to a slight imbalance in entities distribution across services (1 and 2, respectively). The *relations composition* test scores 10.0 due to the concise "published language", the only relation of the system containing one shared entity. The *responsibilities composition* test scores 4.0 due to its uneven use cases distribution across services: 1 use case for one service and 4 for another service. The *semantic similarity* test score 8.5 due to high entities semantic relatedness, with examples such as "Article" and "Booking" being part of the same service, while "Customer" is part of the other service, thus revealing high cohesion inside the microservices.

The **Trading System** scores are collected in Table III. The granularity received the overall score of only 1.1 as its only component, the *nanoentities composition* test, revealed a severe imbalance in nanoentities distribution across microservices: one has 13 nanoentities, while the rest only 3 nanoentities respectively.

The coupling scored the average of 7.1 with *dependencies composition* test scoring 4.2, due to one service having no

Checker	Checker Score	Test Name	Test Score
Granularity	5.4	Nanoentities	5.4
Coupling	10	Dependencies	10.0
		Scc	10.0
Cohesion	7.0	Entities	5.3
		Responsibilities	4.0
		Relations	10.0
		Semantic	8.5

TABLE II
BOOKING SYSTEM: NON-DETERMINISTIC DECOMPOSITION SCORES.

outward dependencies while its peers have one respectively. The *SCC* test identified no strongly connected components in the system, thus scoring 10.

The cohesion scored only 2.4. The *entities composition* test scored 0.8 as a severe imbalance in entities distribution was identified with 1,7,3 and 1 respectively. The *responsibilities composition* test scored 1.8 due to uneven distribution of use case responsibilities across the microservice with 1,6,1, and 2 respectively. The *relations composition* test failed with 0.0 due to "published language" containing duplicates of the "Stock" shared entity. The only *good* score (7.1) related to cohesion was obtained by the *semantic similarity* test due to related entity names implying acceptable cohesion.

Checker	Checker Score	Test Name	Test Score
Granularity	1.1	Nanoentities	1.1
Coupling	7.1	Dependencies	4.2
		Scc	10.0
Cohesion	2.4	Entities	0.8
		Responsibilities	1.8
		Relations	0.0
		Semantic	7.1

TABLE III
TRADING SYSTEM: NON-DETERMINISTIC DECOMPOSITION SCORES.

Table IV collects all the scores for the **Movie Crawler System**. Granularity received an overall score of 2.9 as its only component, the *nanoentities composition* test, revealed a severe imbalance in nanoentities distribution across microservices: one has 2 nanoentities, while another has 11.

Coupling scored perfectly as the *dependencies composition* test indicated that each non-terminal node has one outward dependency. The *SCC* test scored 10 as no strongly connected components were found inside the system.

Cohesion scored 5.3 with *entities composition* test scoring 3.7 due to an imbalanced distribution of entities across the services, with 1 and 4 respectively. The *responsibilities composition* test scored 0.0 due to the first service having no use case responsibilities while the latter has 3. However, the *relations composition* test scored perfectly as the only relation between the two components of the system is publishing only 1 shared entity. The *semantic similarity* test increases the average of this attribute assessment by scoring 7.3, which implies cohesion between the terms used to represent the entities.

Table V collects the scores for the **Ticket Price Comparator**. The granularity has an overall score of only 2.9 as its only component, the *nanoentities composition* test, reveals an imbalanced nanoentity distribution among microservices, with 2,7 and 11 respectively.

Checker	Checker Score	Test Name	Test Score
Granularity	2.9	Nanoentities	2.9
Coupling	10.0	Dependencies	10.0
		Scc	10.0
Cohesion	5.3	Entities	3.7
		Responsibilities	0.0
		Relations	10.0
		Semantic	7.3

TABLE IV
MOVIE CRAWLER: NON-DETERMINISTIC DECOMPOSITION SCORES.

Coupling scored 10.0, as the *dependencies composition* test shows each non-terminal node having one outward dependency. The *SCC* test also scored 10.0, as no strongly connected components were found in the system.

The cohesion scored the average of 5.1. The *entities composition* test scored 3.0 as an imbalance in entities distribution was identified with 1,3 and 5 respectively. The *responsibilities composition* test scored only 2.9 due to imbalanced distribution of use case responsibilities: one service has none, while the rest 2 respectively. The *relations composition* test passed with a 6.67 as one relation has one shared entity while the other has 2. The *semantic similarity* test scored 7.6 as the average of all similarity algorithms, the results implying cohesion between the names of the entities belonging to the same microservice.

Checker	Checker Score	Test Name	Test Score
Granularity	2.9	Nanoentities	2.9
Coupling	10.0	Dependencies	10.0
		Scc	10.0
Cohesion	5.1	Entities	3.0
		Responsibilities	2.9
		Relations	6.7
		Semantic	7.6

TABLE V
TICKET PRICE COMPARATOR: NON-DETERMINISTIC DECOMP. SCORES.

2) Deterministic decompositions: User in theory controls the number of microservices yielded by Service Cutter [18].

All **Cargo Tracker** scores are collected in Table VI. Granularity received the overall score of 1.2 from the *nanoentities composition* test. The framework identified a high variation between microservices sizes as the cause for the low score. This is further backed by the detailed results, showing: 3 nanoentities in services "Service A", "Service B" and "Service C"; 2 nanoentities in "Service D"; and 4 nanoentities in "Service E" and "Service F"; 5 nanoentities in "Service G"; 1 nanoentity in "Service H", thus proving a disproportionate placement of nanoentities across the microservices.

Coupling scored overall 5.5. The *dependencies composition* test scored only 1.1 due to a high variation in outward dependencies with: 5,0,0,0,3,3,2,0 outward communication paths corresponding respectively to each of the 8 proposed microservices. Similarly to the non-deterministic decomposition of this workload instance, the *SCC* test flagged no strongly connected component increasing the average for the coupling assessment.

Cohesion scored overall 3.1. The *entities composition* test has a score of 1.1 due to an imbalance in entities distribution: 3,2,1,1,1,1,2,1. The *responsibilities composition* test scored 4.4 due to a similarly imbalanced use case responsibility distribution among the microservices: 5,0,0,0,1,1,2,0 use cases corresponding respectively to each of the 8 proposed microservices. The *relations composition* test failed completely (score 0.0) due to the abundance of duplicates in the published language with ”Cargo” and ”Itinerary” appearing in two relations, ”RouteSpecification”, ”Voyage”, ”Location” having 3 duplicates respectively and lastly the ”Delivery” shared entity having 4 duplicates. The *semantic similarity* test scored 6.8 as the average of all available algorithms.

Checker	Checker Score	Test Name	Test Score
Granularity	1.2	Nanoentities	1.2
Coupling	5.6	Dependencies	1.1
		Scc	10
Cohesion	3.1	Entities	1.1
		Responsibilities	4.4
		Relations	0
		Semantic	6.8

TABLE VI
CARGO TRACKER: DETERMINISTIC DECOMPOSITION SCORES.

The **Booking System** scores are shown in Table VII. Granularity scored overall 2.3 as its only component, *nanoentities composition* test, revealed a severe imbalance in nanoentities distribution across the proposed microservices with 2 entities belonging to the first and last service, 5 nanoentities to the second service and 7 to the third service.

Coupling scored only 6.5 as one of its components, *dependencies composition* test, scored 2.9 due to each microservice having 0,1,1 and 2 outward dependencies respectively. Similarly to the non-deterministic decomposition, the *SCC* test flagged no strongly connected component.

Cohesion has an overall score of 3.4, as three out of four component tests did not scored a passing mark (5.5). The *semantic similarity* test is the only one to pass with a score of 9.3 due semantically related entities belonging to the same service. The *entities composition* test scored only 3.1 due to an imbalance in entities distribution across microservices with 1,1,2 and 1 respectively. A similar situation is recorded for the *responsibilities composition* test in which the third service has two more use case responsibilities than its peers leading to a score of 1.3. Moreover, the *relations composition* test failed completely due to the ”Customer” shared entity being published by several microservices.

Checker	Checker Score	Test Name	Test Score
Granularity	2.3	Nanoentities	2.3
Coupling	6.5	Dependencies	2.9
		Scc	10.0
Cohesion	3.4	Entities	3.1
		Responsibilities	1.3
		Relations	0.0
		Semantic	9.3

TABLE VII
BOOKING SYSTEM: DETERMINISTIC DECOMPOSITION SCORES.

The scores for the **Trading System** are shown in Table VIII. Similarly to the non-deterministic version of this workload instance, the granularity checker retrieved an overall score of 1.1 due to one service having 13 nanoentities, while the rest have 3 nanoentities respectively.

Coupling scored perfectly as the *dependencies composition* test as each non-terminal node has one outward dependency. The *SCC* test scored 10 maintaining the perfect average as no strongly connected components were found inside the system.

Cohesion scored 2.4, similarly to the non-deterministic version. For the Trading application, we observe that the deterministic and non-deterministic decompositions yielded similar results (Tables III and VIII), for similar reasons.

Checker	Checker Score	Test Name	Test Score
Granularity	1.1	Nanoentities	1.1
Coupling	10.0	Dependencies	10.0
		Scc	10.0
Cohesion	2.4	Entities	0.8
		Responsibilities	1.8
		Relations	0.0
		Semantic	7.1

TABLE VIII
TRADING SYSTEM: DETERMINISTIC DECOMPOSITION SCORES.

The scores for the **Movie Crawler System** are shown in Table IX Granularity has an overall score of only 3.4 as its only component, *nanoentities composition* test, revealed a severe imbalance in nanoentities distribution across microservices, with 4,7, and 2 nanoentities per microservice.

Coupling scored 7.1 as the *dependencies composition* test scored 4.1 due to the fact that only one service has 2 outward dependencies while its peers have none. The *SCC* test scored 10 as no strongly connected components were found.

Cohesion scored the average of 6.2 with *entities composition* test scoring 6.5 due to a slight imbalance in entities distribution with 2, 2 and 1 respectively. The *responsibilities composition* test scored only 1.8 as one service has 1 use case responsibility while another 2 use cases and the last one, none. However, the *relations composition* test scored perfectly due to each of the two relations publishing 1 shared entity, thus ensuring equal distribution of published language. The *semantic similarity* test scores 6.5 as the terms used for entities are deemed cohesive by the underlying similarity algorithms.

Checker	Checker Score	Test Name	Test Score
Granularity	3.4	Nanoentities	3.4
Coupling	7.1	Dependencies	4.1
		Scc	10.0
Cohesion	6.2	Entities	6.5
		Responsibilities	1.8
		Relations	10.0
		Semantic	6.5

TABLE IX
MOVIE CRAWLER: DETERMINISTIC DECOMPOSITION SCORES.

The scores for the **Ticket Price Comparator** are shown in Table X. Granularity has the overall score of only 1.0 as its only component, *nanoentities composition* test, revealed an imbalanced nanoentity distribution among microservices, one having 5 nanoentities, two having a single nanoentity, 2 nanoentities for each of the two and 3 nanoentities for the rest.

Coupling scored 7.1. The *dependencies composition* test scored 4.1 due to only three out of eight services having outward dependencies. The first and last services have 1 outward dependency respectively, another service has 2 outward dependencies, while the rest have none. The *SCC* test scored 10 as no strongly connected components were found.

The cohesion scores 2.9. The *entities composition* test scores 1.7 as a slight imbalance in entities distribution was identified: one service has 2 entities, while the rest have only one respectively. The *responsibilities composition* test failed completely due to imbalanced distribution of use case responsibilities: only four services out of eight have one use case responsibility respectively. The *relations composition* test failed completely (with 0.0) due to the "Movie" entity being identified as having duplicates. The *semantic similarity* test scored 9.8 as the average of all similarity algorithms toggled on, the results implying high cohesion between the names of the entities belonging to the same microservice.

Checker	Checker Score	Test Name	Test Score
Granularity	1.0	Nanoentities	1.0
Coupling	7.1	Dependencies	4.1
		Scc	10.0
Cohesion	2.9	Entities	1.7
		Responsibilities	0.0
		Relations	0.0
		Semantic	9.8

TABLE X
TICKET PRICE COMPARATOR: DETERMINISTIC DECOMPOSITION SCORES.

3) Data-flow Diagram decompositions (DFD): Table XI shows the scores for the **Movie Crawler System**. Granularity and coupling score 3.4 and 7.1 respectively, similarly to the deterministic version (Table IX), and the same discussion applies here. In contrast, cohesion scores 8.3 due to the *responsibilities composition* test scoring 10.0 as each service has exactly one use case responsibility.

Checker	Checker Score	Test Name	Test Score
Granularity	3.4	Nanoentities	3.4
Coupling	7.1	Dependencies	4.1
		Scc	10.0
Cohesion	8.3	Entities	6.5
		Responsibilities	10.0
		Relations	10.0
		Semantic	6.5

TABLE XI
MOVIE CRAWLER: DFD DECOMPOSITION SCORES.

Table XII shows the scores for the **Ticket Price Comparator**. Granularity scores 3.1, as its only component, the *nanoentities composition* test reveals an imbalanced nanoentity distribution across microservices: 4,4,5 and 7 nanoentities, respectively. Coupling scores 10.0. The *dependencies composition* test scores 10 as each non-terminal node has one outward dependency, thus being evenly distributed. The *SCC* test scores 10 as no strongly connected components were found.

Cohesion scores 7. The *entities composition* test scores 5.5 as a slight imbalance in entities distribution was identified: 2 services contain 2 entities each, while the other two services have 3 entities each. The *responsibilities composition* test scores 10.0 due to each service having one use case responsibility. The *relations composition* test scores 6.5 due to unequally distributed "published language" among the relations: one has 2 shared entities, the rest have only one respectively. The *semantic similarity* scores 6.1 as the average of all similarity algorithms results, implying an acceptable level of cohesion across the entities names of the same microservice.

Checker	Checker Score	Test Name	Test Score
Granularity	3.1	Nanoentities	3.1
Coupling	10.0	Dependencies	10.0
		Scc	10.0
Cohesion	7.0	Entities	5.5
		Responsibilities	10.0
		Relations	6.5
		Semantic	6.1

TABLE XII
TICKET PRICE COMPARATOR: DFD DECOMPOSITION SCORES

4) Manual decomposition (Ideal): The scores are shown in Table XIII. For the **Custom Cargo Tracker Adaptation**, since the input file containing the system model specifications for this validation experiment was manually idealised, we expect perfect scores (10.0) for all the tests contained in granularity checker, except the *LoC* test. This test was applied on a Cargo Tracker DDD example repository [19] and resulted in a low score due to its high variation in microservices' sizes: 314, 1020, and 97 lines of code, respectively. By following the *cargo-ideal*[5] system model we observe that the equal distribution of nanoentities (4) per microservice leads to a score of 10.0 for the *nanoentities composition*.

Coupling scores 10.0. The *dependencies composition* test scored 10.0 due to each non-terminal microservice having only one outward dependency. The *SCC* test flagged no strongly connected components in the system's dependency graph.

Cohesion scores 10.0, the *entities composition* test reporting equal distribution of entities with one per microservice. The *responsibilities composition* test scores 10.0 due to a balanced use case responsibilities distribution with 3 for each microservice respectively. The *relations composition* test scored perfectly with 2 relations distributed among 3 microservices which share only one entity as "published language" for each relation. The *semantic similarity* test scores 10.0, the average of all available algorithms, due to strong semantic closeness in the names the entities and nanoentities.

Checker	Checker Score	Test Name	Test Score
Granularity	6.7	Nanoentities	10.0
		Loc	3.3
Coupling	10.0	Dependencies	10.0
		Scc	10.0
Cohesion	10.0	Entities	10.0
		Responsibilities	10.0
		Relations	10.0
		Semantic	10.0

TABLE XIII
CARGO TRACKER: IDEAL DECOMPOSITION SCORES.

C. *MicroValid Performance Analysis Results*

For each test currently implemented in MicroValid: *nanoentities*, *dependencies*, *entities*, *relations*, *responsibilities*, and *semantic*, we conduct performance analysis experiments. We report the time complexity of the algorithm, possible correlations between the input file sizes (ordered ascending) and runtimes, together with their explanation.

[5]https://github.com/michelcojocaru/MasterProject/blob/master/test-files/service-cuts/cargo-tracker/cargo-tracker_ideal.json

1) Nanoentities composition test: has a low variation in runtimes per input file. The mean runtimes do not exceed $0.05\,(\mathrm{ms})$, except the deterministic decomposition of Cargo Tracker. This workload instance runs up to $0.15\,(\mathrm{ms})$ with the mean approximately $0.1\,(\mathrm{ms})$.

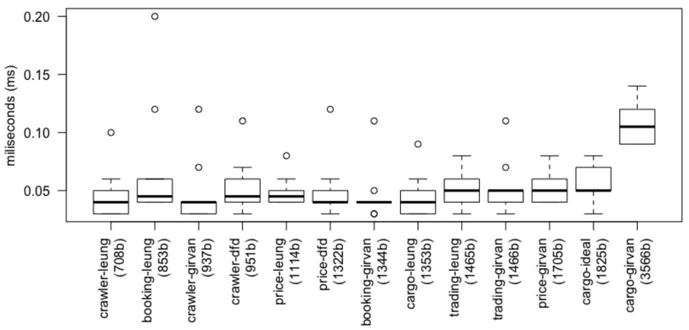

Fig. 1. Nanoentities test runtimes statistics.

The correlation coefficient $R = 0.81$ with $p - values < 0.05$ indicates a slight dependency between the runtimes and the file sizes, which can be explained by the algorithm complexity of $O(N)$, where N = number of microservices. The file size grows with the number of microservices contained in the system specifications model. These two observations indicate that the runtimes are more correlated to the number of microservices in the system, rather than the input file size.

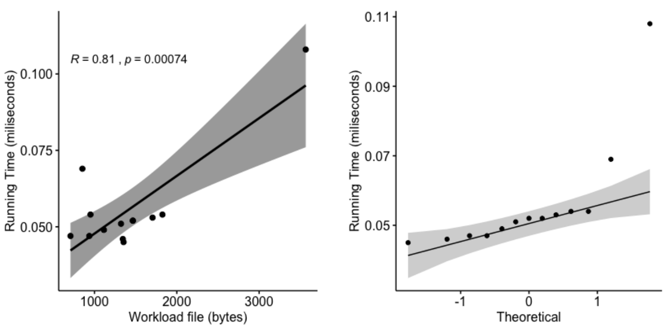

Fig. 2. Nanoentities test run times vs. file sizes correlation.

2) LoC test: We present the statistical analysis of runtimes for the Cargo Tracker application codebase [19]. The mean runtime over 10 iterations is $70\,(\mathrm{ms})$, the values generally varying between $65\,(\mathrm{ms})$ and $75\,(\mathrm{ms})$ with a minimum of $57\,(\mathrm{ms})$ and maximum of $94\,(\mathrm{ms})$. The variation across mean runtime values for this test across the rest of the files is small, between $0.08\,(\mathrm{ms})$ and $0.1\,(\mathrm{ms})$.

3) SCC test: Recorded consistently runtimes under $0.1\,(\mathrm{ms})$, disregarding 4 outliers which are ceiling at $0.13\,(\mathrm{ms})$. Most of the values are between $0.02\,(\mathrm{ms})$ and $0.06\,(\mathrm{ms})$, a correlation between the running times and workload file sizes being identified. This is due to Tarjan's algorithm [10] employed in searching strongly connected components being linear to the number of microservices and their communication paths $O(|N| + |E|)$, where N = number of microservices, and E = number of communication paths.

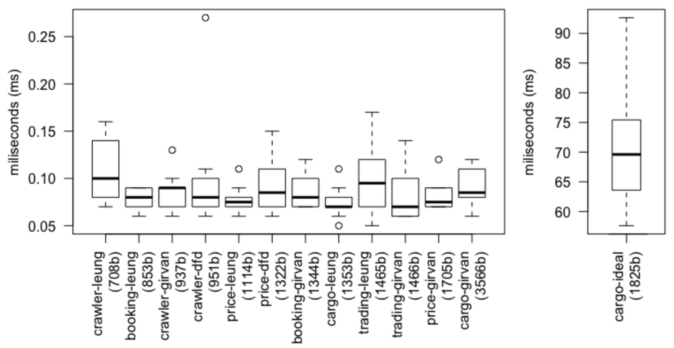

Fig. 3. LoC test running times statistics.

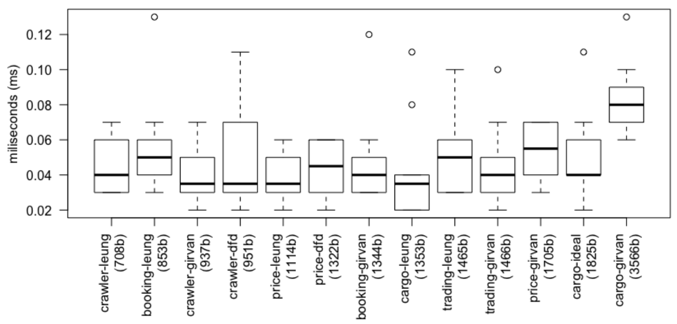

Fig. 4. SCC test running times statistics.

We identified a positive correlation between the runtimes and the input file sizes of $R = 0.85$ with $p - values < 0.05$, which can be explained by algorithm complexity. This relies on the number of microservices and relations specified in the system model, thus implying a stronger linear dependency with the file sizes compared to the *nanoentities composition* test.

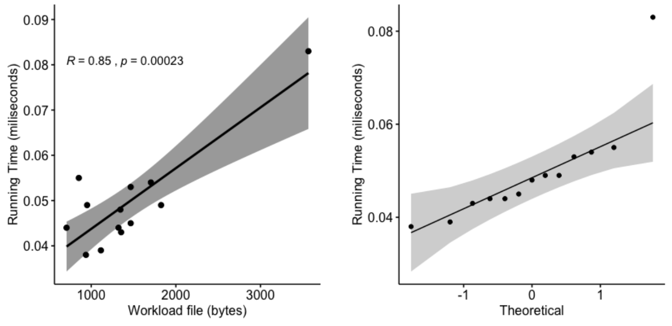

Fig. 5. SCC test run times vs. file sizes correlation.

4) Semantic Similarity test: Recorded runtimes under $450\,(\mathrm{s})$ with 10 out 13 workload files running under $100\,(\mathrm{s})$. The 3 remaining workload files yielded runtimes from $100\,(\mathrm{s})$

to $300\,(\mathrm{s})$, with the non-deterministic decomposition of the Trading application setting the maximum at approximately $450\,(\mathrm{s})$. Per workload application, the runtime is calculated as the sum of all eight semantic similarity assessment algorithms.

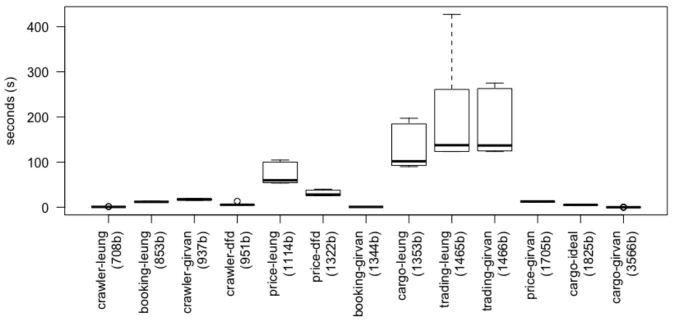

Fig. 6. Semantic test running times statistics.

Without information about the internal logic of the semantic algorithms, we could not identify dependencies between runtimes and the number of microservices, entities, nanoentities and relations. The correlation coefficient between the input file sizes and the runtimes for this test is $R = -0.062$ with $p-values > 0.84$. Since the significance level exceeds 0.05, we assume no correlation between the input file sizes and the semantic similarity test runtimes.

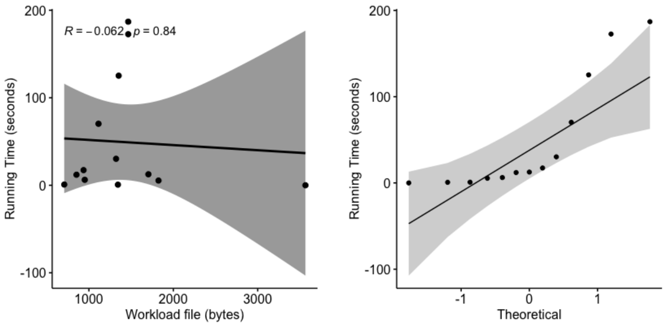

Fig. 7. Semantic test run times vs. file sizes correlation.

D. Discussion

Although they are highly debated, issues such as a high variation in lines of code per microservice that may arise when assessing granularity [20] can be circumvented [21] by using relativity between the components of the same system. We argue that in industry it is often the case that developers adhere to a strict coding style guideline which aids similar writing further supporting the relativity approach.

The *LoC* test is highly dependent on users internet downstream bandwidth due to the actual download of the remote repository locally taking the majority of time for this test. Due to the fact that we managed to gather only one repository, the statistical correlation is not feasible. For calculating the correlation between the run times of this test against the input file sizes would imply to possess repositories for each of the 13 versions of the considered applications, which is not the case. The run times on different codebases might have shown correlations between the running time and the size of the repository (as number of classes) together with the number of microservices and communication paths.

Although we could not identify correlations between the runtimes of the *semantic similarity* test and the system model specifications, we found that eliminating the *Hirst St'Onge* from the semantic algorithms selection leads to a steep correction of runtimes, consistently dropping by 95%. We present no data backing this matter due to our focus being on the performance analysis of our framework and not on semantic similarity algorithms.

IV. Related Work

We identify related research on two main topics: the migration process, and validation techniques or tools.

Migrations. Many cloud-based companies chose to migrate towards MSAs due to their benefits [22]. Additionally, many non-cloud enterprises started similar migrations, ranging from finance critical systems [23], database as service [24], to cloud-based applications [25]. The most adopted migration approach relies on Domain Driven Design (DDD) for partitioning complex models into bounded contexts scoping their relations [25]–[27]. The most employed pattern in migration process is the "Strangler" pattern [28].

Validation. The previously mentioned decomposition techniques produce service "cuts" of the original monoliths. Current work provides only limited and often subjective decomposition validation, such as [4], [15]. Other approaches focus on the core architectural principles of MSA. Characteristics drawn from the definition such as small sized, independent, loosely coupled, cohesive and programming language agnosticism are used in [15] to validate their resulted microservices. Team size reduction and structural improvements are synthesized via the Team Reduction Metric [20] which uses algorithmic contributors information extraction from version control systems.

V. Conclusions

As companies choose to migrate from monolithic applications to Microservice Architecture rather than rebuild the entire application from scratch, the scene of (semi-)automatic decomposition tools is expected to expand. The migration process generally implies decomposition and validation of the result. Yet, a missing piece in this puzzle is automatic validation. We provide an easily extensible microservice validation framework which tests the quality of the microservices resulted from semi-automatic decomposition processes with emphasis on static analysis quality attributes.

In this work we provide a reference implementation together with the methodology for implementing additional tests for a validation framework. We evaluate our reference implementation for a validation framework using the results of a semi-automatic decomposition tool (the Service Cutter [4]) and of an established technique for decomposing monoliths (the Data-Flow Diagram [15]).

References

[1] E. Van Eyk, L. Toader, S. Talluri, L. Versluis, A. Uta, and A. Iosup, "Serverless is more: From paas to present cloud computing," *IEEE Internet Computing*, vol. 22, no. 5, pp. 8–17, 2018.

[2] M.-D. Cojocaru, A. Oprescu, and A. Uta, "Attributes assessing the quality of microservices automatically decomposed from monolithic applications," in *2019 18th International Symposium on Parallel and Distributed Computing (ISPDC)*. IEEE, 2019, pp. 84–93.

[3] P. D. Francesco, P. Lago, and I. Malavolta, "Migrating towards microservice architectures: An industrial survey," pp. 29–2909, April 2018.

[4] G. W. Z. O. Gysel M., Klbener L., "Service cutter: A systematic approach to service decomposition," *Lecture Notes in Computer Science*, vol. 9846, 8 2016.

[5] D. Kruidenberg, "From monoliths to microservices the decomposition process," Ph.D. dissertation, Universiteit van Amsterdam, 2018.

[6] "Technical Action Research," http://rcis-conf.com/rcis2012/document/slides/RCIS12_TechnicalActionResearch.pdf, [Online; accessed 27.03.2019].

[7] "Chain of Responsibility Pattern," https://www.journaldev.com/1617/chain-of-responsibility-design-pattern-in-java, [Online; accessed 25.03.2019].

[8] "Swagger," https://swagger.io, [Online; accessed 25.03.2019].

[9] M. Kalske, N. Mkitalo, and T. Mikkonen, *Challenges When Moving from Monolith to Microservice Architecture*, 02 2018, pp. 32–47.

[10] "Tarjan algorithm," https://github.com/thai321/Algorithm-Problems-Java/tree/master/Strongly\%20Connected\%20Components\%20(SCC), [Online; accessed 11.04.2019].

[11] "Understanding SOLID Principles: Single Responsibility," https://codeburst.io/understanding-solid-principles-single-responsibility-b7c7ec0bf80, [Online; accessed 27.02.2019].

[12] "WordNet Similarity for Java," https://code.google.com/archive/p/ws4j/, [Online; accessed 25.03.2019].

[13] "Java Wrapper for Japanese WordNet," https://code.google.com/archive/p/jawjaw/, [Online; accessed 25.03.2019].

[14] "NICT Japanese WordNet," https://github.com/Sciss/jawjaw, [Online; accessed 25.03.2019].

[15] R. Chen, S. Li, and Z. Li, "From monolith to microservices: A dataflow-driven approach," pp. 466–475, Dec 2017.

[16] "Domain-driven design: Tackling complexity in the heart of software," 2003.

[17] I. X. Leung, P. Hui, P. Lio, and J. Crowcroft, "Towards real-time community detection in large networks," *Physical Review E*, vol. 79, no. 6, p. 066107, 2009.

[18] M. E. Newman and M. Girvan, "Finding and evaluating community structure in networks," *Physical review E*, vol. 69, no. 2, p. 026113, 2004.

[19] "CargoTracker domain-driven design blueprints for Java EE Repository," https://github.com/javaee/cargotracker, [Online; accessed 11.05.2019].

[20] G. Mazlami, J. Cito, and P. Leitner, "Extraction of microservices from monolithic software architectures," pp. 524–531, June 2017.

[21] A. Z. J. Bogner, S. Wagner, "Automatically measuring the maintainability of service and microservice-based systems - a literature review," pp. 107–115, 2017.

[22] D. Francesco, I. Malavolta, and P. Lago, "Research on architecting microservices: Trends, focus, and potential for industrial adoption," pp. 21–30, 4 2017.

[23] M. Mazzara, N. Dragoni, A. Bucchiarone, A. Giaretta, S. T. Larsen, and S. Dustdar, "Microservices: Migration of a mission critical system," *IEEE Transactions on Services Computing*, pp. 1–1, 2018.

[24] A. Balalaie, A. Heydarnoori, and P. Jamshidi, "Migrating to cloud-native architectures using microservices: An experience report," *Springer International Publishing*, pp. 201–215, 2016.

[25] A. Balalaie, A. Heydarnoori, and P. Jamshidi, "Microservices architecture enables devops: an experience report on migration to a cloud-native architecture," *IEEE Software*, vol. 33, pp. 1–1, 05 2016.

[26] J. Thnes, "Microservices," *IEEE Software*, vol. 32, no. 1, pp. 116–116, Jan 2015.

[27] O. Mustafa, J. Marx Gmez, M. Hamed, and H. Pargmann, "Granmicro: A black-box based approach for optimizing microservices based applications," pp. 283–294, 01 2018.

[28] "Strangler pattern," https://docs.microsoft.com/nl-nl/azure/architecture/patterns/strangler, [Online; accessed 03.03.2019].